ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

IMPLEMENTACIÓN DE UN PROTOTIPO DE UNA RED DEFINIDA POR SOFTWARE (SDN) EMPLEANDO UNA SOLUCIÓN BASADA EN HARDWARE

PROYECTO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y REDES DE INFORMACIÓN

JUAN CARLOS CHICO JIMÉNEZ
jcchicoj@yahoo.com

DIRECTOR: ING. DAVID MEJÍA, MSc.
david.mejia@epn.edu.ec

Quito, Agosto 2013

DECLARACIÓN

Yo, Juan Carlos Chico Jiménez, declaro bajo juramento que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedo mis derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.

JUAN C. CHICO JIMÉNEZ

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Juan Carlos Chico Jiménez bajo mi supervisión.

______________________________	______________________________
Ing. David Mejía, MSc.	**Iván Bernal, Ph.D.**
DIRECTOR DEL PROYECTO	**CODIRECTOR DEL PROYECTO**

AGRADECIMIENTO

Agradezco a todas las personas que me apoyaron durante toda la carrera y en especial, en el periodo de tiempo en el cual realice el presente Proyecto de Titulación. Particularmente, agradezco a mi director de proyecto, Master David Mejía y a mi codirector Dr. Iván Bernal, por el tiempo que dedicaron para la revisión del proyecto y por toda la ayuda y orientación recibida.

A mis padres Patricio y María Soledad, por sus consejos y ayuda que me impulsaron a salir adelante y a mi hermana María Belén, por su invaluable ayuda y orientación en todo momento.

A mi enamorada Sofía, por todo su cariño, paciencia y comprensión.

A mi mejor amiga Anna, por todo el apoyo a lo largo de todos estos años.

Y a todas esas personas que colaboraron de una u otra manera con el desarrollo del proyecto, para ellos mis más sinceros agradecimientos.

DEDICATORIA

A mis padres, enamorada, amigos y personas que confiaron en que podía salir adelante y concluir el presente Proyecto. Gracias a ustedes he podido culminar esta etapa de mi vida. Se cierra un ciclo de mi vida y en el siguiente ciclo espero seguir contando con su amistad y apoyo, así como ustedes cuentan con el mío.

CONTENIDO

ÍNDICE DE FIGURAS .... **VIII**
PRESENTACIÓN .... **XI**
RESUMEN .... **XII**

CAPÍTULO I

1. MARCO TEÓRICO **1**
1.1 CARACTERÍSTICAS GENERALES DE LAS SDN (SOFTWARE DEFINED NETWORKS) 1
1.1.1 DEFINICIÓN 1
1.1.2 ARQUITECTURA SDN 2
1.1.2.1 Ventajas de la arquitectura SDN 3
1.1.2.2 Servidor controlador SDN 4
1.1.2.2.1 NOX 5
1.1.2.2.2 POX 7
1.1.2.2.3 Beacon 11
1.1.2.2.2 Floodlight 13
1.1.2.3 Dispositivos de conectividad 14
1.1.3 COMPARACIÓN ENTRE LA ARQUITECTURA DE RED TRADICIONAL Y LA ARQUITECTURA SDN 16
1.2 EL PROTOCOLO OPENFLOW 18
1.2.1 CONMUTADOR OPENFLOW 20
1.2.2 MENSAJES OPENFLOW 22

1.2.2.1 Mensajes del controlador al conmutador 22
1.2.2.2 Mensajes asincrónicos...23
1.2.2.3 Mensajes simétricos 24

1.3 MININET 24

CAPÍTULO II

2. SIMULACIÓN DE LA RED 27

2.1 COMPONENTES USADOS EN LA SIMULACIÓN 27

2.1.1 COMPONENTES DE HARDWARE..27
2.1.2 COMPONENTES DE SOFTWARE 28

2.2 TOPOLOGÍA USADA EN LA SIMULACIÓN 29

2.2.1 CREACIÓN DE TOPOLOGÍAS POR DEFECTO EN MININET 31
2.2.2 CREACIÓN DE TOPOLOGÍAS PERSONALIZADAS EN MININET 33

2.2.3 CREACIÓN MEDIANTE MINIEDIT DE TOPOLOGÍAS PERSONALIZADAS EN MININET..37

2.3 CONTROLADOR NOX DE MININET 39

2.4 COMANDO DPCTL...40
2.5 PRUEBAS Y RESULTADOS 49

2.5.1 PRUEBAS CON EL CONTROLADOR NOX USANDO EL COMPONENTE PYSWITCH 50
2.5.2 PRUEBAS CON EL CONTROLADOR NOX USANDO UN COMPONENTE PERSONALIZADO..57

CAPÍTULO III

3. IMPLEMENTACIÓN DEL PROTOTIPO...68

3.1 TOPOLOGÍA....68
3.1.1 PLATAFORMA DE HARDWARE....70
3.1.1.1 Servidor Controlador....70
3.1.1.2 Conmutadores....71
3.1.1.3 Hosts....74
3.2 SOFTWARE DEL SERVIDOR CONTROLADOR....74
3.2.1 NOX....75
3.2.2 POX....78
3.2.3 BEACON....83
3.2.4 FLOODLIGHT....92
3.3 PRUEBAS Y RESULTADOS....97
3.3.1 PRUEBAS DE MENSAJES OPENFLOW....97
3.3.2 PRUEBAS CON LOS COMPONENTES PERSONALIZADOS....105
3.3.2.1 POX....106
3.3.2.2 Beacon....112
3.3.2.3 Floodlight....116
3.3.2.4 Análisis de resultados....120
3.4 PRESUPUESTO REFERENCIAL DEL PROTOTIPO....122

CAPÍTULO IV

4. CONCLUSIONES Y RECOMENDACIONES....125
4.1 CONCLUSIONES....125
4.2 RECOMENDACIONES....130

REFERENCIAS BIBLIOGRÁFICAS....135
ANEXOS

ÍNDICE DE FIGURAS

CAPÍTULO I

Figura 1.1 Arquitectura SDN 3
Figura 1.2 Rendimiento de NOX vs POX 8
Figura 1.3 Comparación entre NOX, Maestro y Beacon con 16 conmutadores....12
Figura 1.4 Comparación entre NOX, Maestro y Beacon con 32 conmutadores....12
Figura 1.5 Diagrama de operación de Floodlight..14
Figura 1.6 Dispositivo de conectividad SDN..15
Figura 1.7 Comunicación mediate OpenFlow..15
Figura 1.8 Componentes de una red SDN ejecutando OpenFlow.........................18
Figura 1.9 Encabezados soportados por un conmutador tipo 0 22

CAPÍTULO II

Figura 2.1 Topología usada para la simulación de la red......................................29
Figura 2.2 Topología lineal...31
Figura 2.3 Topología árbol, con los parámetros M=2 y N=2 33
Figura 2.4 Creación de la topología en Mininet...37
Figura 2.5 Interfaz gráfica de Miniedit 38
Figura 2.6 Ejecución del controlador NOX 39
Figura 2.7 Tabla de flujos vacía del conmutador s3..43
Figura 2.8 Flujos ingresados en la tabla de flujos del conmutador s3...................45
Figura 2.9 Flujos ingresados en la tabla de flujos del conmutador s4...................46
Figura 2.10 Ejecución del comando *mod-port* y su resultado en la interfaz 47
Figura 2.11 Envío de paquetes ICMP fallido, puerto 2 de s3 en estado noflood...48
Figura 2.12 Configuración de los hosts 50

Figura 2.13 Envío de mensajes ICMP entre dos hosts de la VLAN 10 51

Figura 2.14 Captura de paquetes...52

Figura 2.15 Detalle de un paquete OpenFlow..55

Figura 2.16 Captura de mensajes OpenFlow...56

Figura 2.17 Envío de mensajes ICMP entre dos hosts de la VLAN 20 56

Figura 2.18 Prueba *pingall* 65

CAPÍTULO III

Figura 3.1 Topología usada para la implementación de la SDN 69

Figura 3.2 Acceso mediante telnet al conmutador 73

Figura 3.3 Ayuda de NOX 75

Figura 3.4 Ejecución de NOX...77

Figura 3.5 Ejecución de POX 80

Figura 3.6 Ejecución del controlador Beacon...84

Figura 3.7 Interfaz web de Beacon..84

Figura 3.8 Adición de conmutadores en Beacon...85

Figura 3.9 Componentes de Beacon...86

Figura 3.10 Detalle de los flujos en Beacon 86

Figura 3.11 Campo Wildcards...92

Figura 3.12 Ejecución de Floodlight 93

Figura 3.13 Identificación de los conmutadores en Floodlight..............................94

Figura 3.14 Comando *dpctl show tcp*...98

Figura 3.15 Comando *dpctl dump-ports*...99

Figura 3.16 Captura de mensajes *Hello*..100

Figura 3.17 Captura del mensaje *Features Request*..100

Figura 3.18 Detalle de un paquete *Features Reply*..101

Figura 3.19 Resultado del comando *mod-port* (apagar puerto)..........................102

Figura 3.20 Detalle de un mensaje *Port Mod* 103

Figura 3.21 Resultado del comando *mod-port* (*noflood*) 103

Figura 3.22 Detalle de un mensaje *Port Mod* 104

Figura 3.23 Detalle del comando *Flow Mod* 105

Figura 3.24 Ejecución de POX con el componente of_tutorial 109

Figura 3.25 Reglas de flujos en el conmutador S1 110

Figura 3.26 Envío de mensajes ICMP entre V11 y V21 110

Figura 3.27 Comunicación HTTP entre V11 y V21 111

Figura 3.28 Comunicación telnet entre V12 y V22 111

Figura 3.29 Captura de un *streaming* de video originado en V23 112

Figura 3.30 Ejecución de Beacon con el componente personalizado 115

Figura 3.31 Reglas de flujos en el conmutador S1 116

Figura 3.32 Ejecución del *script* para la creación de reglas de flujos 119

Figura 3.33 Visualización en Floodlight de las reglas añadidas en S1 119

Figura 3.34 Reglas de flujos en el conmutador S1 120

PRESENTACIÓN

El mundo ha cambiado mucho en las últimas décadas; principalmente, este cambio ha sido impulsado por el avance de las telecomunicaciones y las redes de información. Hoy en día transacciones, que se demoraban un largo período de tiempo, se hacen prácticamente en un segundo, entre puntos separados geográficamente por grandes distancias.

La red más importante durante este proceso e inclusive en la actualidad y en el futuro es Internet, que posibilita la conexión de miles de usuarios, compañías, etc. Esta red se ha desarrollado en base a una arquitectura TCP/IP, que fue creada en principio como una red militar pero que se ha ido adaptando a las necesidades del medio y que hoy en día probablemente se acerca al final de su vida útil. Esto se debe a que las formas de comunicación han evolucionado: hoy en día se usan muchos servicios en la nube, la información que atraviesa la red ya no es solo datos de aplicaciones, sino también paquetes de voz, video, etc. con una calidad igual que a través de canales dedicados.

Hace algunos años se creó una nueva arquitectura para dar soporte a las necesidades del mundo actual; esta arquitectura permite tener redes más personalizables y eficientes, que permiten dar calidad de servicio a las comunicaciones prioritarias, como es el caso de voz y video y que permiten además brindar una amplia cobertura en cuanto a seguridad y control de acceso. Estas redes se denominan Redes Definidas por Software (SDN *Software Defined Networks)* y su uso se ha incrementado paulatinamente, hasta acaparar la atención de grandes compañías como Google, Microsoft, entre otras.

En el presente Proyecto de Titulación se estudiará esta arquitectura y cómo implementarla en una red.

RESUMEN

El presente Proyecto de Titulación se encuentra dividido en cuatro capítulos, organizados de la siguiente manera: el primer capítulo corresponde al marco teórico básico de las SDN, el segundo capítulo a la simulación de una SDN, el tercer capítulo a la implementación de una SDN y finalmente se presentan las conclusiones y recomendaciones obtenidas de la realización del proyecto.

En el marco teórico se presentan los fundamentos que se requieren para poder conocer y comprender las SDN, comenzando por las características generales de este tipo de red, su arquitectura y los dispositivos que se pueden encontrar dentro de una SDN; posteriormente, se habla del protocolo OpenFlow, que se encarga de la comunicación de los dispositivos de la SDN, haciendo especial énfasis en el dispositivo inteligente de la red, que se denomina controlador, y los mensajes que se envían durante el proceso de comunicación. Finalmente se explica la herramienta que se usa para realizar la simulación de una SDN denominada Mininet.

En la simulación de una SDN se explica la topología y componentes de hardware y software requeridos para llevar a cabo la simulación; posteriormente, se presentan las opciones para crear una topología en la simulación, a continuación se habla del controlador que se usa para la simulación en Mininet y finalmente se presentan las pruebas y resultados obtenidos de la realización de la simulación. Especialmente, se programa un componente personalizado para el controlador que envía instrucciones o reglas a los dispositivos de conmutación de la red.

En la implementación se procede de manera similar a la simulación: en primer lugar se discute la topología de la red a utilizarse, a continuación los componentes de hardware que la componen; posteriormente, las opciones de controladores utilizados y finalmente se presentan las pruebas y resultados de la programación

de tres componentes para tres controladores diferentes, para el envío de reglas a los dispositivos de conmutación y se realiza un análisis de sus similitudes y diferencias.

El cuarto capítulo presenta las conclusiones y recomendaciones de este Proyecto de Titulación y finalmente se incluyen varios anexos que facilitan la instalación y configuración de todas las herramientas de software que se usan en el transcurso de la simulación e implementación del proyecto.

En el CD adjunto se puede encontrar el código del componente NOX creado en la simulación del proyecto, así como el código de tres componentes creados: POX, Beacon y Floodlight, en el capítulo correspondiente a la implementación del prototipo. Además se encuentran incluidos los archivos de configuración para cada componente.

CAPÍTULO I

1. MARCO TEÓRICO

1.1 CARACTERÍSTICAS GENERALES DE LAS SDN (*SOFTWARE DEFINED NETWORKS*)

En el primer capítulo se profundizará acerca del marco teórico referente a las SDN para tener una base que sirva como referencia para interpretar los siguientes capítulos, que se enfocan en cuestiones netamente prácticas. Las SDN surgen como alternativa de las redes actuales, debido a la necesidad de tener una mayor escalabilidad y tener un manejo de tráfico más personalizable y eficiente, ya que la capacidad y eficiencia de las redes actuales ha sido sobrepasada por la cantidad y tipo de tráfico que se maneja.

Hoy en día este nuevo concepto de redes ha tenido una buena aceptación en empresas que manejan una gran cantidad de datos, como es el caso de Google.

Para entender más acerca de las SDN, primero se hará una definición de ellas, posteriormente se analizará su arquitectura y se realizará una comparación con las redes tradicionales. Posteriormente se hablará del protocolo OpenFlow, que es el encargado de la comunicación entre el controlador y los nodos de la red y finalmente se hablará de la herramienta de simulación Mininet, la cual se usará en el siguiente capítulo.

1.1.1 DEFINICIÓN [1]

Una Red Definida por Software (SDN*, Software Defined Network*) tiene como objetivo fundamental separar el plano de control del plano de datos, cosa que no

se da en las redes de datos tradicionales, con lo cual se pretende disociar la infraestructura de la red de las aplicaciones y servicios presentes en ella, los cuales ven a la red como una entidad virtual o lógica. Una SDN permite tener redes más escalables y flexibles que se adaptan fácilmente a cambios, al poder el administrador de la red intervenir sobre los flujos de datos que circulan a través de la red.

En lo referente al control sobre la red, esto se da básicamente a través de la programación de reglas que definen patrones y comportamientos de tráfico mientras este transita por la red. Estas reglas son programadas en un dispositivo denominado servidor controlador SDN. La gestión de la red se encuentra altamente centralizada sobre este dispositivo que maneja toda la inteligencia de la red.

Al implementar este tipo de red, se simplifica el diseño y control en comparación a una red normal, debido a que se obtiene independencia del fabricante del equipo y se simplifican los equipos, ya que estos no necesitan interpretar varios protocolos, solo aceptar las instrucciones del dispositivo controlador.

1.1.2 ARQUITECTURA SDN [1], [2]

La arquitectura SDN está basada en un servidor controlador, el cual inspecciona los patrones del flujo de datos, de acuerdo a las reglas definidas en él, y los dispositivos de interconectividad, los cuales se encargan de recibir órdenes del controlador y aplicarlas.

En la Figura 1.1 se puede observar la arquitectura SDN básica. En ella se muestra que el servidor controlador es el encargado del manejo del plano de control y se comunica con los dispositivos de conectividad a través del protocolo OpenFlow, que se explicará posteriormente.

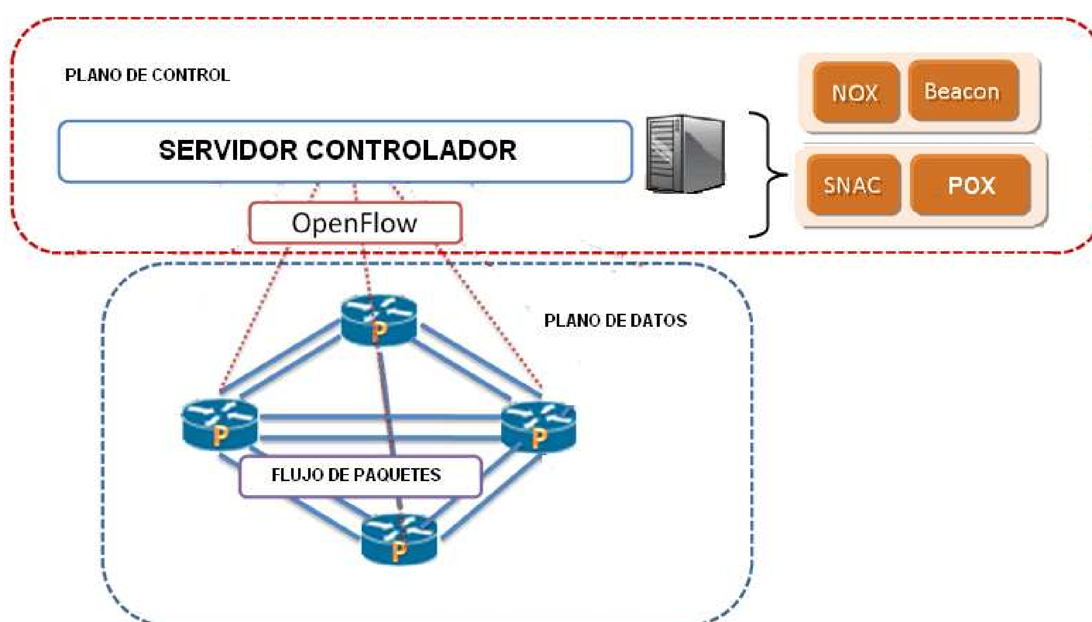


Figura 1.1 Arquitectura SDN [2]

1.1.2.1 Ventajas de la arquitectura SDN [1]

La implementación de una SDN tiene varios beneficios en comparación a los esquemas de red actuales, los mismos que se detallan a continuación:

- Una administración centralizada de la red, a través de un solo dispositivo denominado servidor controlador SDN, en el cual se programan todas las reglas para la circulación de tráfico a través de la red.
- Se obtiene una red con soporte de diferentes marcas de equipos de conectividad, con solo hacer pequeñas modificaciones al *firmware*. El administrador es capaz de manipular la red e implementar mejoras por sí mismo, sin la necesidad de esperar a que los fabricantes de sus equipos introduzcan nuevas tecnologías. Como resultado se obtiene una independencia de los vendedores de equipos de conectividad, al tener un control absoluto sobre la red.
- Se obtienen mejoras tanto en la automatización así como en la gestión de la red a través de Interfaces de Programación de Aplicaciones (API) que son las encargadas de la abstracción de la red subyacente.

- La facilidad de innovación crece al tener la capacidad de proveer nuevas capacidades y servicios a la red sin tener que configurar los dispositivos de manera individual o esperar el software nuevo o actualizado del proveedor para poder implementar nuevas características.
- El dispositivo controlador facilita la programación por parte del administrador o los fabricantes mediante la utilización de lenguajes de programación comunes y conocidos por la mayoría de personas que trabajan en el medio.
- La seguridad de la red se incrementa debido a una mejora en la confiabilidad provocada por el manejo centralizado de dispositivos, al impedir o hacer menos frecuentes errores de configuración que pudiesen generar brechas de seguridad provocadas por un manejo inconsistente y no integrado de los dispositivos de la red. Esto permite una aplicación uniforme de políticas y menos errores en la configuración de seguridad de los dispositivos.
- Se brinda un mejor servicio y experiencia al usuario final, ya que las redes se adaptan fácilmente a los cambios y requerimientos que se necesiten para una determinada aplicación.

1.1.2.2 Servidor Controlador SDN [10]

El controlador SDN es la parte central de la arquitectura SDN, debido a que es el que mantiene toda la inteligencia de la red. En el controlador, el administrador se encarga de definir reglas para administrar el flujo de datos en la red. Esto permite una configuración rápida, que supone una ventaja frente a las redes tradicionales, en las cuales se debe esperar a que el fabricante lo haga, al igual que la implementación de nuevas aplicaciones y servicios.

El controlador ejecuta un software que le permite llevar a cabo sus tareas, que puede ser: NOX, POX, Beacon, Floodlight u otro, diferenciándose uno de otro solamente por el lenguaje de programación con el cual se definan las reglas para el flujo de datos.

1.1.2.2.1 NOX [3]

NOX es un software que se instala y ejecuta en el controlador, para definir reglas que se usarán para administrar el flujo de datos que atraviesa la red SDN con el fin de tener control sobre la misma.

NOX es fabricado por Niciara Networks y fue desarrollado a la par del protocolo OpenFlow. A continuación se presentan las principales características de NOX:

- Incluye una API en C++.
- Posee una entrada y salida rápida y asincrónica, es decir que al software controlador puede ingresar y salir un gran flujo de datos en cualquier instante de tiempo.
- Está diseñado para ser instalado y configurado sobre sistemas operativos Linux, sobre todo Ubuntu en sus versiones 11.0 y 12.04, aunque también puede implementarse fácilmente sobre Debian, Red Hat o CentOS.

Actualmente se encuentran disponibles dos versiones de NOX:

- NOX *Classic*: es una versión antigua que ofrece soporte para Python y C++; sin embargo, la versión fue descontinuada y ya no existe actualmente desarrollo en esta línea. No se recomienda implementar esta versión; sin embargo, cuenta con un rango de módulos más amplio que la nueva versión.
- NOX: es la nueva versión de este software. Ofrece soporte exclusivamente para C++. Sin embargo, ofrece un rango de aplicaciones reducido en comparación a la versión clásica. Su funcionamiento es más rápido y cuenta con un código más depurado. Esta versión, actualmente, se encuentra en continuo desarrollo.

Existen versiones de NOX activas, en las cuales se añaden nuevos elementos que son desarrollados en el transcurso del tiempo, y otras versiones en las que ya no se incluyen nuevos elementos y las actualizaciones permiten solamente

corregir errores del software. La versión activa de NOX se llama "*verity*" y las versiones de NOX *classic* son "*destiny*" y "*zaku*".

Los objetivos de este software controlador son:

- Proveer una plataforma que permita a los desarrolladores realizar innovaciones en el manejo de la red.
- Tener un manejo centralizado de dispositivos de conectividad, como por ejemplo conmutadores, un control de admisión a nivel de usuarios y un motor de políticas de seguridad para la red.

El control de los dispositivos se lo realiza mediante el protocolo OpenFlow, que se detallará en la sección 1.2. NOX ofrece métodos de bajo nivel para interactuar con la red en su forma más básica. Las funciones de alto nivel y los eventos son creados o provistos por aplicaciones de red que se denominan componentes. Los componentes son encapsulados que ofrecen un cierto tipo de funcionalidad. Por ejemplo, las funciones de enrutamiento se encuentran encapsuladas en un componente. Si se requiere agregar enrutamiento a una red, se debe declarar el encapsulado de enrutamiento como dependencia para poder hacer uso de esta característica.

Las aplicaciones se dividen en varios grupos, dependiendo de sus funcionalidades. Las más importantes son las denominadas de *core* o núcleo. Estas se encargan de dar ciertas funcionalidades para las aplicaciones de red y servicios web[1]. Los componentes de núcleo se los encuentra bajo el directorio src/nox/coreapps. A continuación se presentan ejemplos de los mismos:

- *Messenger*: incluye las funcionalidades de sockets TCP (*Transmission Control Protocol*) o SSL (*Secure Socket Layer*) para la comunicación del controlador con otros dispositivos de la red.

[1] Un servicio web es una tecnología que se utiliza para intercambiar datos entre aplicaciones. El servicio web se compone de protocolos y estándares que facilitan y rigen el intercambio de información. Su objetivo es tener una interoperabilidad de las aplicaciones, sin importar el lenguaje o plataforma en la que han sido desarrolladas [4].

- SNMP (*Simple Network Management Protocol*): permite el manejo de *traps* SNMP usando un *script* escrito en el lenguaje Python. El manejo de *traps* se realiza mediante Net-snmp.

Otro tipo de componentes son las aplicaciones de red, que como su nombre lo indica, brindan funcionalidades de red. Los componentes de red pueden ser encontrados bajo el directorio src/nox/netapps. A continuación se presentan ejemplos de los mismos:

- *Discovery*: Permite dar seguimiento a los diversos enlaces que existan entre conmutadores dentro del dominio del controlador.
- *Topology*: Permite mantener un registro de los enlaces activos de la red.
- *Authenticator*: Permite mantener un registro con la ubicación de los *hosts* y conmutadores de una red.
- *Routing*: Es el componente encargado de calcular el mejor camino hacia una red.
- *Monitoring*: Este componente consulta periódicamente a los conmutadores de la red para obtener estadísticas, que posteriormente son presentadas. Así se obtiene una visión del funcionamiento de la red y se pueden detectar posibles problemas.

Adicionalmente, se tienen aplicaciones web que permiten el manejo de NOX a través de servicios web. Esto se da mediante un servidor web que aloja la interfaz de control y servicios web que se comunican con las aplicaciones.

1.1.2.2.2 POX [5], [6]

POX es una herramienta muy similar a NOX, pero ambas se diferencian en que POX usa el lenguaje Python para programar reglas que indican al controlador que acciones llevar a cabo, mientras que NOX puede usar tanto Python como C++. Esta herramienta fue creada con el fin de permitir un desarrollo rápido y dinámico de reglas de flujo para un controlador de una SDN.

Se usa, principalmente, para crear reglas de flujo dentro de una SDN, para comprobar la distribución de flujo de un determinado prototipo y para verificar, depurar y corregir errores de un prototipo de controlador de red. Actualmente, POX se encuentra bajo desarrollo constante. El objetivo de sus desarrolladores es brindar una API bastante estable.

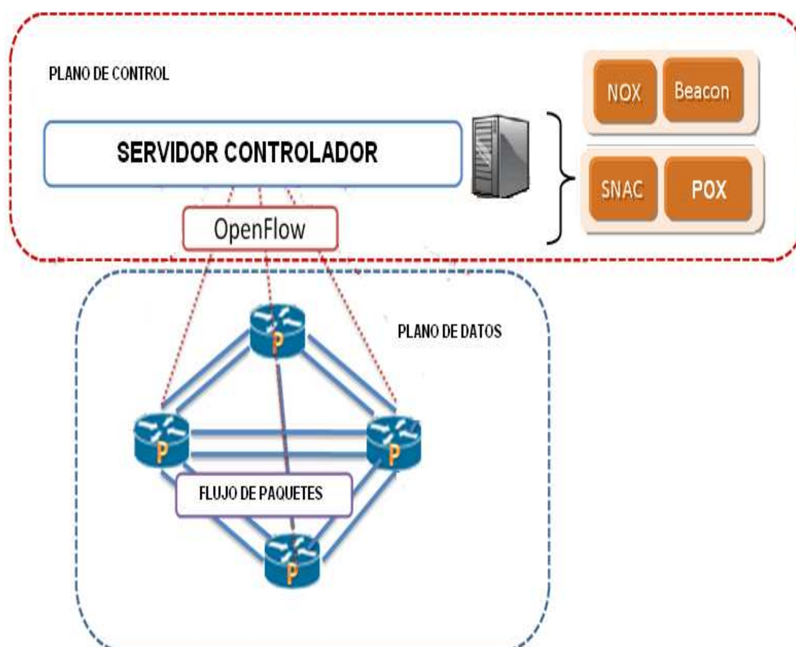


Figura 1.2 Rendimiento de NOX vs POX [5]

POX presenta las siguientes características:

- Posee una interfaz OpenFlow denominada “*Pythonic*” por encontrarse en el ambiente de desarrollo de Python.
- Puede correr en cualquier plataforma: Linux, Windows, Mac OS o cualquier otra, ya que se puede combinar con “PyPy”, que es un entorno de ejecución que permite soportar la ejecución de programas escritos en Python. Este entorno de ejecución tiene implementaciones básicas de comandos de bajo nivel y puede ejecutar comandos de alto nivel.
- Soporta la misma GUI (Interfaz Gráfica de Usuario) que NOX y tiene la misma visualización, debido a que son líneas de desarrollo prácticamente paralelas.

- Tiene un mejor rendimiento que los programas NOX escritos en Python, por lo que ha reemplazado a la versión descontinuada NOX *Classic*. En la Figura 1.2 se presenta un gráfico del rendimiento de un equipo manejando POX, y otros equipos manejando NOX con controladores escritos en C++ y Python, en la cual se observa que el mayor rendimiento lo tiene la herramienta NOX con un controlador escrito en C++, ya que presenta una baja latencia y gran *throughput* medido en flujos controlados por segundo. Sin embargo, de la figura, se puede inducir que si se desea programar el controlador en Python, la herramienta más efectiva y de mayor rendimiento es POX.
- Posee ciertos componentes de muestra que pueden ser reutilizables para la selección del mejor camino, descubrimiento de topologías, etc.

Para la invocación de POX, se debe correr pox.py, especificando qué componentes se desean incluir en la ejecución. Existen componentes ya desarrollados y otros que pueden ser creados por el administrador. Entre los componentes existentes se encuentran:

- Forwarding.l2_learning: permite una funcionalidad similar al de un conmutador capa dos en estado de aprendizaje; es decir, reconocer las direcciones MAC de los dispositivos conectados y llenar la tabla de direcciones MAC.
- Forwarding.l3_learning: funciona de manera similar al componente anterior, salvo que incluye funcionalidades de capa tres, como por ejemplo el manejo de ARP (*Address Resolution Protocol*).
- Samples.spanning_tree: brinda las funcionalidades del protocolo *spanning tree*, es decir que permite crear el árbol *spanning tree* y luego bloquea los puertos redundantes que no van a ser utilizados para evitar posibles lazos a nivel de capa dos.
- Web.webcore: es el encargado de iniciar un servicio web dentro de los procesos POX.
- Messenger: es una interfaz que permite al controlador interactuar con procesos externos; sin embargo, este componente es solo una API, ya

que la comunicación es implementada por los componentes denominados de transporte. Este servicio actualmente se encuentra disponible tanto para transporte TCP como para HTTP (*HyperText Transfer Protocol*).

- Openflow.of_01: este componente permite la comunicación con el protocolo OpenFlow.
- Openflow.discovery: se encarga de enviar mensajes para las tareas de descubrimiento de topologías.
- Samples.pong: Es un componente que implementa un ejemplo de comunicación a través de mensajes ICMP. El componente escucha las peticiones ICMP y, posteriormente, las responde para comprobar el funcionamiento de este protocolo en una arquitectura SDN.
- Log: permite manejar la información de *logging,* así como asignar niveles de alerta de estos elementos.

Además de estos componentes estándar, el usuario puede programar sus propios componentes para que brinden la función que se requiera. Estos componentes pueden ser almacenados para mayor facilidad dentro del directorio “ext”, ya que Python automáticamente agrega estos componentes y existe una mayor facilidad al momento de trabajar con ellos.

Adicionalmente POX cuenta con varias API, como por ejemplo:

- API para el manejo de direcciones (pox.lib.addresses): permite el manejo tanto de direcciones MAC como IP con las clases EthAddr e IpAddr respectivamente.
- API para el sistema de eventos (pox.lib.event): para clases que deseen incorporar el manejo de eventos.
- API para el manejo de paquetes (pox.lib.packet): permite manejar varios tipos de paquetes, como por ejemplo: Ethernet, IPv4, ICMP, TCP, UDP, ARP, DHCP, DNS, entre otros.
- API para el manejo de hilos de ejecución, tareas y contadores (pox.lib.recoco).

1.1.2.2.3 Beacon [6]

Beacon es un software para la programación de reglas de flujo en controladores SDN rápido, escrito en Java, multiplataforma y modular. Posee las siguientes características:

- Estable: este software se considera estable ya que ha sido probado en varios proyectos de investigación y en *data centers* sin sufrir ninguna interrupción en sus servicios por largos periodos de tiempo.
- Multiplataforma: gracias a las características de Java de poder correr sobre cualquier dispositivo debido a que se ejecuta sobre una máquina virtual propia de Java denominada JVM (Java *Virtual Machine*).
- Código Abierto: el software se encuentra en el mercado bajo la licencia GPLv2 y la licencia de la Universidad de Stanford FOSS[2] v1.0.
- Rápido funcionamiento: debido a la utilización de múltiples hilos de ejecución.

En las Figuras 1.3 y 1.4 se presenta una comparación entre Beacon, NOX y otro software controlador llamado Maestro, en donde se puede observar el alto rendimiento del controlador con varios conmutadores, a pesar de que con pocos dispositivos, NOX es más eficiente.

En la Figura 1.4 se muestra el alto rendimiento del software Beacon expresado en el número de flujos por segundo que es capaz de manejar en una red con un controlador Beacon que maneja 32 conmutadores.

---

[2] FOSS: *Free and Open-Source Software*.

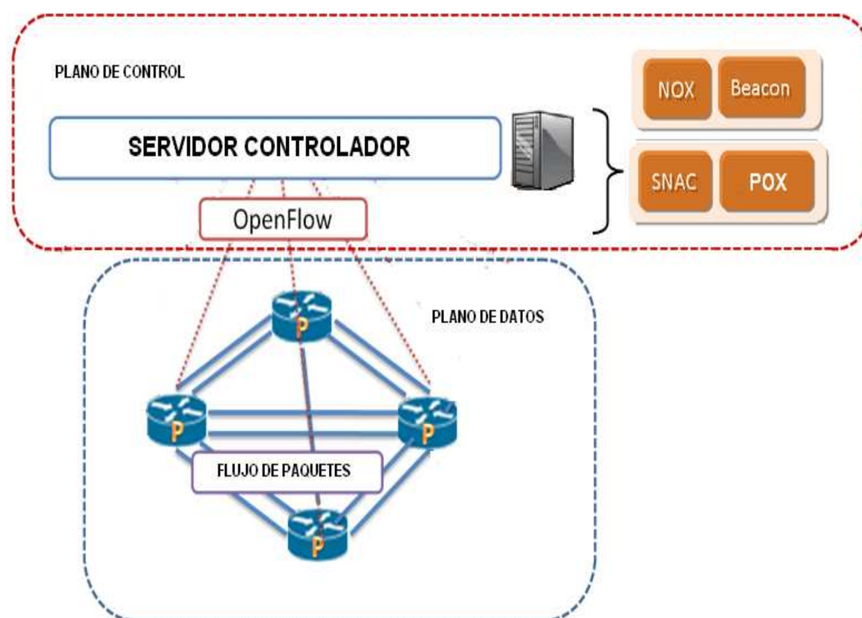


Figura 1.3 Comparación entre NOX, Maestro y Beacon con 16 conmutadores [2]

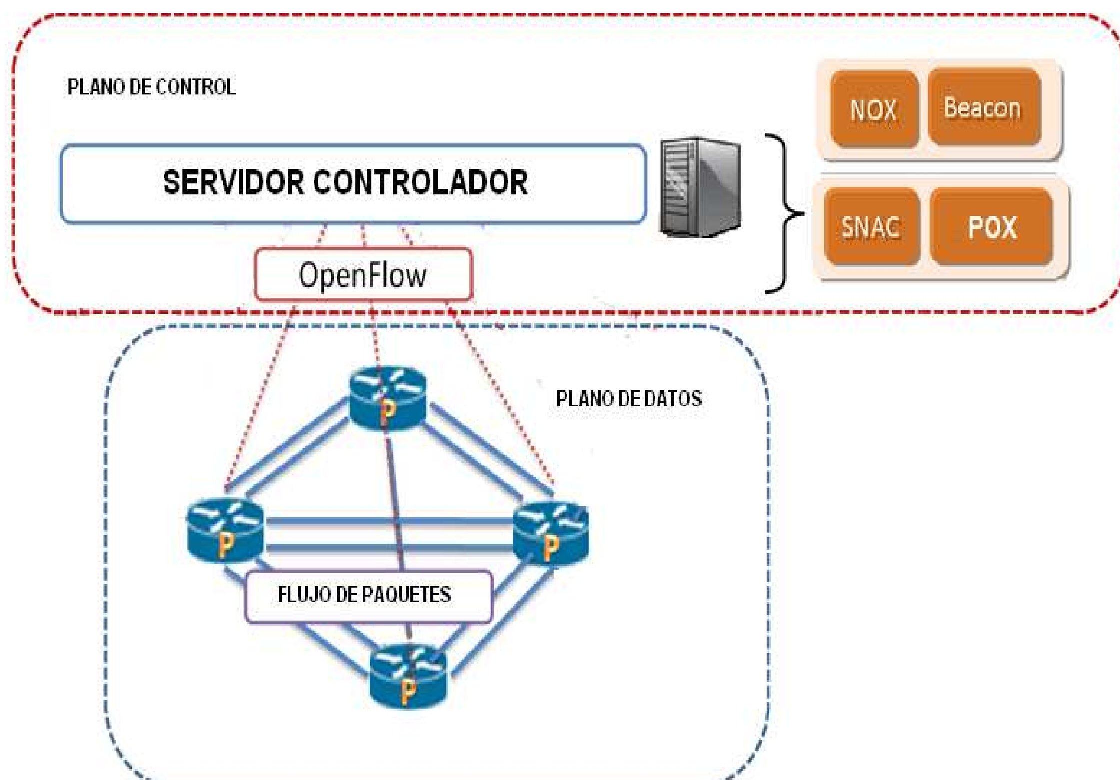


Figura 1.4 Comparación entre NOX, Maestro y Beacon con 32 conmutadores [2]

- Dinámico: permite manipular librerías dinámicamente, es decir parar, iniciar o refrescar la ejecución de ellas, e instalar nuevas librerías en tiempo de ejecución sin interrumpir el funcionamiento de otras librerías independientes.
- Desarrollo rápido: gracias a las facilidades que brinda Java y al entorno Eclipse, que ofrecen una programación más rápida y amigable con el usuario.
- Posee una interfaz web: manejada de manera opcional, que incluye el servidor web *Jetty Enterprise*.

1.1.2.2.4 Floodlight [12][13]

Floodlight es un software controlador SDN que está escrito en Java y que corre sobre una máquina virtual de Java JVM. El software se caracteriza por:

- Trabajar con OpenFlow tanto en conmutadores físicos como virtuales.
- Es un controlador basado en módulos que permiten obtener ciertas funcionalidades como ruteo, conmutación, etc. El módulo de conmutación se denomina Learning Switch.
- Se puede instalar en Ubuntu 10.04 o superior o MacOS 10.6 o superior.
- Para su funcionamiento se requiere el software Java Developement Kit (JDK) y ANT[3] de Apache, que es una herramienta usada en programación para realizar tareas repetitivas en Java.
- Permite tener una isla OpenFlow, es decir una agrupación de dispositivos que entienden este protocolo, conectado a una isla con una red tradicional que no entiende este protocolo mediante un único enlace.

En la Figura 1.5 se muestra un diagrama general de como Floodlight opera.

Floodlight es un software controlador SDN que mantiene una comunicación con los dispositivos de red. Mediante esta comunicación el controlador envía órdenes

[3] ANT: *Another Neat Tool.*

o reglas de flujo que permiten al dispositivo conmutar el tráfico entrante y enviarlo por el camino destinado para un determinado flujo de datos. Sobre Floodlight se pueden implementar varios tipos de aplicaciones que brindarán cierto tipo de funcionalidad al controlador. Entre ellas *Virtual Switch*, que es una aplicación de virtualización de redes o ACL, que permite definir listas de control de acceso en la red OpenFlow.

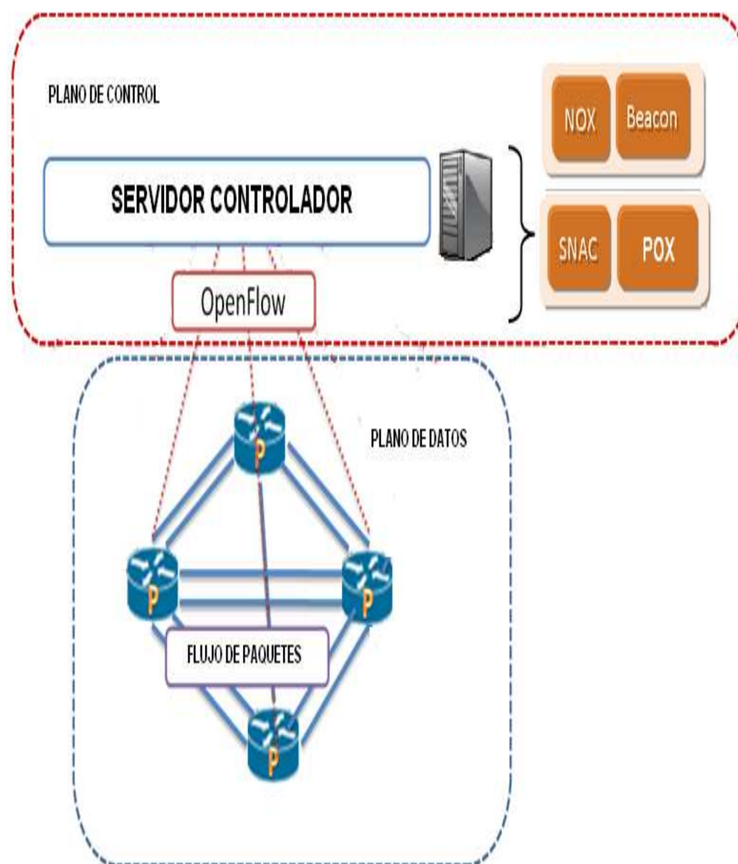


Figura 1.5 Diagrama de operación de Floodlight [12]

1.1.2.3 Dispositivos de conectividad [2]

Un dispositivo de conectividad en una SDN se puede conceptualizar como un elemento conformado por dos componentes, una parte de hardware y otra de software. El hardware permite realizar tareas como conmutación de paquetes, de manera similar a un equipo de conectividad tradicional; mientras que el software

se comunica con el servidor controlador y obtiene órdenes que definen su comportamiento en respuesta a la llegada de un tipo de tráfico determinado.

En la Figura 1.6 se presentan las partes constitutivas de un equipo de conectividad.

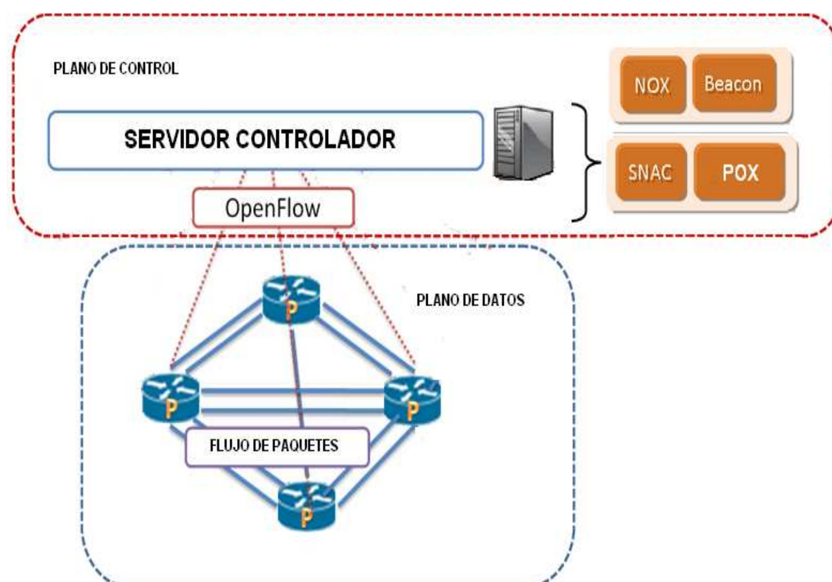


Figura1.6: Dispositivo de conectividad SDN [2]

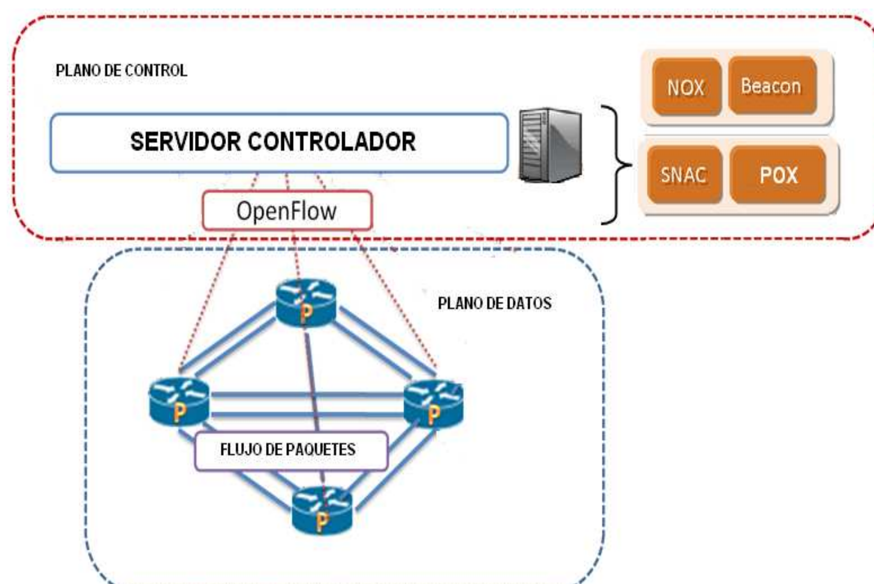


Figura 1.7 Comunicación mediante OpenFlow [2]

De la Figura 1.6 se puede observar que el administrador es capaz de definir el tipo de procesamiento que ejecuta el dispositivo de conectividad. Las órdenes para el manejo del dispositivo se dictan por el usuario a través del controlador y viajan a los dispositivos de conectividad a través de un canal OpenFlow (TCP o SSL), como se muestra en la Figura 1.7.

En la Figura 1.7 se muestra que el dispositivo de conectividad, en este caso un conmutador que maneja el protocolo OpenFlow, tiene una tabla de flujos muy similar a una tabla de ruteo o de conmutación. Esta tabla obedece a las reglas que el controlador ha definido y de esta manera se maneja el tráfico de la red.

Para los siguientes capítulos cabe recalcar que se usará el término conmutador para referirse a un dispositivo de conectividad cualquiera, ya que en una SDN no se tienen conmutadores o enrutadores, solo dispositivos capaces de realizar las funciones que el usuario desee a través de la conmutación de flujos, por lo que el término conmutador puede tener mayor significado que un conmutador tradicional en una SDN.

1.1.3 COMPARACIÓN ENTRE LA ARQUITECTURA DE RED TRADICIONAL Y LA ARQUITECTURA SDN [1]

Las redes que se usan actualmente fueron diseñadas hace varios años para cubrir requerimientos que han ido evolucionando a lo largo del tiempo. La arquitectura de estas redes ha servido eficazmente hasta la actualidad, pero últimamente han surgido nuevos requerimientos que no pueden ser satisfechos eficazmente, por lo que se han desarrollado arquitecturas alternativas, como SDN, para cubrir estos requerimientos.

A continuación se presenta una comparación entre la arquitectura de red tradicional y la arquitectura SDN.

- En los últimos años, los patrones de tráfico han cambiado. El modelo cliente-servidor ha sido desplazado por un modelo consistente de varios

servidores que realizan varias transacciones entre sí para finalmente responder al cliente, por lo que se requiere una conexión en cualquier ubicación en cualquier momento, tanto entre los servidores así como con los clientes para brindar un servicio eficiente. Las SDN permiten crear reglas que admiten dar prioridad al tráfico entre servidores y así tener transacciones más rápidas.

- Últimamente, los denominados *Cloud Services* han aumentado considerablemente. Estos servicios necesitan agilidad al acceder a las aplicaciones y recursos, además de una gran seguridad, escalabilidad y facilidad de adaptación a cambios. Las SDN permiten incorporar estas características en la red de una manera eficiente, al manejar seguridad centralizada y tener una gran facilidad de adaptación a los cambios de la red.
- El volumen de datos manejados actualmente es mucho mayor que el de unos años atrás, por lo que hoy en día se requiere más capacidad de la red, conjuntamente con una alta disponibilidad, cosas que SDN ofrece a través del manejo de reglas para los flujos de datos.
- Los protocolos en la arquitectura tradicional se manejan de manera aislada y resuelven problemas específicos, sin embargo, no realizan una abstracción de la red subyacente, lo que no permite separar las aplicaciones de consideraciones que se deben tener en cuenta en el envío de paquetes por la red y que suman complejidad a toda la red. Esto provoca una demora en el tráfico al atravesar dispositivos, actualizar listas de control de acceso, realizar enrutamiento, entre otros. Las SDN permiten manejar redes mucho más dinámicas, acorde a las necesidades de las aplicaciones actuales.
- La calidad de servicio (QoS) en las redes tradicionales se maneja de manera manual, al ser introducidas las políticas por el administrador en cada dispositivo. Las SDN permiten tener calidad de servicio de una manera automática y sin necesidad de configuraciones separadas en cada dispositivo.
- Otro punto débil en las redes tradicionales es la configuración manual de la seguridad en cada dispositivo. Esto puede causar políticas inconsistentes

y huecos de seguridad que pueden ser explotados por atacantes. Las SDN ofrecen seguridad centralizada que previene la introducción de políticas inconsistentes y huecos de seguridad.

- Finalmente, las redes tradicionales no tienen independencia de vendedores y están sujetas a las actualizaciones que realizan los fabricantes. Las SDN permiten que el usuario directamente actualice características de la red e introduzca innovaciones muy fácilmente a través de la programación de reglas que definen patrones de flujo.

1.2 EL PROTOCOLO OPENFLOW [2], [9]

OpenFlow es el primer estándar que se ha desarrollado para una SDN y su objetivo es tener una interfaz entre los planos de control y datos dentro de una arquitectura SDN. Este protocolo permite la manipulación del plano de datos al especificar las primitivas básicas usadas por las aplicaciones corriendo en el servidor para programar dicho plano en los dispositivos de red.

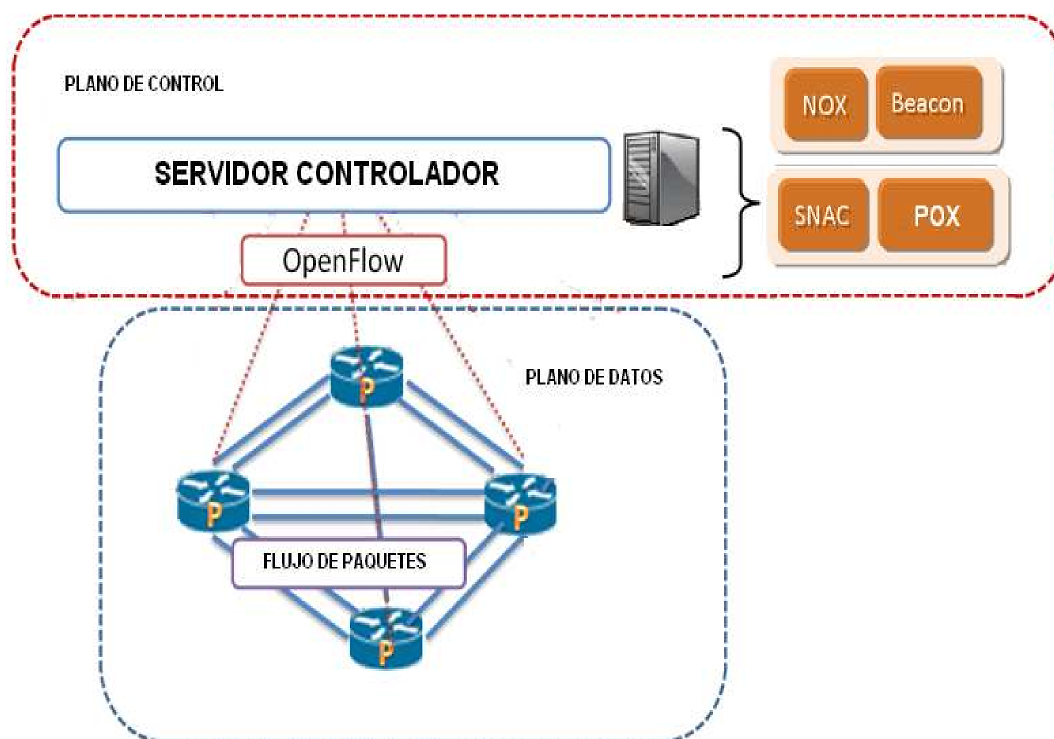


Figura 1.8 Componentes de una red SDN ejecutando OpenFlow [2]

Este protocolo se usa entre los dispositivos de red y el software ejecutándose en el controlador SDN y maneja el concepto de flujos, lo que permite identificar el tráfico que se maneja en la red y darle el tratamiento adecuado mediante la implementación de reglas predefinidas.

En la Figura 1.8 se presentan los elementos constitutivos de una red SDN ejecutando el protocolo OpenFlow. En ella se hallan componentes como el software controlador, del cual se habló en la sección 1.1 y otros componentes, como aplicaciones, herramientas de monitoreo y del hardware, que en este caso vendría a ser un conmutador OpenFlow, del cual se hablará en la siguiente sección.

En la Figura 1.8 se puede observar una jerarquía de componentes de una red SDN que utiliza el protocolo OpenFlow. En el primer nivel, se tiene el hardware de la red, que en este caso se compone de conmutadores usados como dispositivos de interconectividad o software que simula las labores de un conmutador en una SDN. En segundo lugar se encuentra el *slicing software*, es decir software que permite separar una red de forma lógica en varios *slices* o agrupaciones de menor complejidad y simplifica los aspectos de semántica entre las capas uno y tres del modelo ISO/OSI. Específicamente FlowVisor [11] es un software que actúa como *Proxy* cuando se tienen varios controladores OpenFlow. En tercer lugar se tiene el software controlador, del cual se habló anteriormente. En la cuarta jerarquía se tienen las aplicaciones que se ejecutan sobre OpenFlow y por último se tienen las herramientas para monitoreo y control de errores de la red, que son similares a las de una red normal.

En síntesis, el protocolo OpenFlow indica al tráfico como fluir; para lo que se toman en consideración varios parámetros como el patrón de uso de la red, las aplicaciones y los requerimientos específicos de cada una de ellas. Las reglas son definidas para cada flujo, con lo que se obtiene un control sumamente granular que permite a la red tener una gran capacidad de respuesta a los cambios que pueden producirse.

El protocolo OpenFlow puede ser fácilmente desplegado en una red existente ya sea física o virtual. El despliegue depende del soporte de OpenFlow por parte de los dispositivos de conectividad. Si los dispositivos permiten trabajar con OpenFlow, la implementación se realiza de manera sencilla actualizando el *firmware* de los dispositivos de la red.

1.2.1 CONMUTADOR OPENFLOW

Un conmutador OpenFlow es un dispositivo de conectividad que ha sido creado o modificado para poder conectarse con el servidor controlador con el fin de recibir las reglas que definirán patrones de flujo de tráfico mediante el protocolo OpenFlow.

La instalación de OpenFlow en estos dispositivos se ve facilitada por el hecho de que varios de estos dispositivos ya usan tablas de flujos, ya sea para implementar *firewalls*, QoS o para recolectar estadísticas de la red. A pesar de que esto depende en gran medida de los proveedores, se han encontrado funciones comunes que usan la mayoría de ellos.

En el conmutador, OpenFlow ofrece un protocolo para programar la tabla de flujos. Cada entrada de la tabla de flujos se asocia con una determinada acción, que puede ser:

1. Enviar el paquete por un determinado puerto para que sea enrutado a través de la red.
2. Encapsular y enviar al controlador paquetes de un determinado flujo de datos. Esta acción se usa comúnmente en el primer paquete de un nuevo flujo de datos, para determinar si el flujo se añade o no a la tabla de flujos.
3. Eliminar los paquetes de un determinado flujo de datos, principalmente por motivos de seguridad, para prevenir posibles ataques.

Un conmutador OpenFlow se compone de los siguientes elementos:

- La tabla de flujos, en la que se indica el comportamiento que debe seguir cada tipo de tráfico.
- Un canal seguro que permite conectar el dispositivo de conectividad al servidor controlador.
- El protocolo OpenFlow ejecutándose dentro del dispositivo.

Además, un conmutador OpenFlow puede ser clasificado en dos categorías de acuerdo a la implementación del protocolo, teniendo así:

- Conmutador OpenFlow dedicado: el protocolo ya viene instalado en el dispositivo y no se da soporte a las capas dos y tres del modelo tradicional de red. Este dispositivo no tiene inteligencia por sí mismo, ya que envía el tráfico como lo definió el controlador. El flujo es definido y solo es limitado por la capacidad de implementación particular de la tabla de flujo.
- Conmutador OpenFlow habilitado: las características OpenFlow no vienen incluidas pero son añadidas mediante actualizaciones del *firmware*. Se añade la tabla de flujos, el canal seguro y el protocolo OpenFlow para que el dispositivo pueda ser usado dentro de una red SDN.

Un flujo puede entenderse como una serie de paquetes con encapsulación TCP que van de una dirección MAC o IP a otra o como paquetes que atraviesan una determinada VLAN.

Una entrada de una tabla de flujos OpenFlow posee los siguientes campos:

- Un encabezado del paquete, que define o caracteriza el flujo.
- La acción, que identifica cuál debe ser el procesamiento del paquete.
- Información estadística, que permite dar seguimiento al número de paquetes y bytes de cada flujo e información de tiempo que facilita la eliminación de flujos inactivos.

Los conmutadores también pueden clasificarse en dos grupos dependiendo de las funcionalidades con las que cuentan:

- Conmutador tipo 0: soporta los formatos de encabezado que se muestran en la Figura 1.9 y las tres acciones básicas, enumeradas anteriormente, más una acción que permite enviar los paquetes de un flujo a través de la secuencia de procesamiento normal de un conmutador.

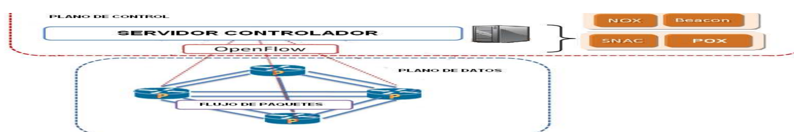


Figura 1.9: Encabezados soportados por un conmutador tipo 0 [2]

- Conmutador tipo 1: soporta funciones adicionales a las del tipo 0, como, por ejemplo, la de sobre escribir porciones de los encabezados de los paquetes y asignar prioridades a los paquetes. Un ejemplo es NAT, que cambia las direcciones IP del encabezado al realizar substituciones de direcciones públicas por privadas y viceversa.

1.2.2 MENSAJES OPENFLOW

OpenFlow cuenta con una gran cantidad de mensajes que le permiten transmitir acciones e información desde el servidor controlador hacia los dispositivos de red o de los dispositivos de red hacia el controlador. Los mensajes se dividen en: mensajes del controlador al conmutador, mensajes asincrónicos y mensajes simétricos [14].

1.2.2.1 Mensajes del controlador al conmutador

Estos mensajes se originan en el controlador y no necesariamente requieren de una respuesta por parte del conmutador. Los mensajes de este tipo son los siguientes [14]:

- *Features*: se envía cuando el controlador requiere obtener las características del conmutador. El conmutador le responde con sus características. Se usa normalmente en el establecimiento de una conexión OpenFlow.
- *Configuration*: mensajes de consulta de los parámetros de configuración del conmutador.
- *Modify-State*: se envían desde el controlador para la gestión de estados en el conmutador, como por ejemplo añadir o quitar flujos o cambiar el estado de un determinado puerto.
- *Read-State*: El controlador lo usa para reunir características del conmutador.
- *Packet-Out*: El controlador los usa para enviar paquetes por un determinado puerto del conmutador y para redireccionar paquetes *Packet-In*. En su interior este paquete contiene el paquete a ser redireccionado y las acciones que se aplicarán al mismo.
- *Barriel*: El controlador lo usa para asegurarse que las dependencias de un mensaje se han cumplido o para recibir notificaciones por operaciones finalizadas.

1.2.2.2 Mensajes asincrónicos

Estos mensajes pueden ser generados sin necesidad de que previamente el controlador haya generado un mensaje. Los conmutadores lo envían cuando un paquete ha llegado, ha ocurrido un cambio en su estado o ha ocurrido un error. Los paquetes de este tipo son los siguientes [14]:

- *Packet_In*: Paquete enviado desde el conmutador hacia el controlador cuando no tiene una entrada en su tabla de flujos que concuerde con el paquete entrante. El controlador procesa el paquete y responde con un mensaje *Packet-Out*.
- *Flow-Removed*: Se envía cuando el tiempo de inactividad de un flujo o el tiempo de vida del flujo vence (estos parámetros se fijarán en los

capítulos 2 y 3). Puede ser enviado tanto por el controlador como por el conmutador.

- *Port-Status*: Se envía del conmutador al controlador cuando la configuración de un puerto cambia.
- *Error:* El conmutador notifica al controlador si existen problemas con estos mensajes.

1.2.2.3 Mensajes simétricos

Estos mensajes se envían en cualquier dirección sin una solicitud previa. En esta clasificación se encuentran los siguientes mensajes [14]:

- *Hello:* Estos mensajes se intercambian al momento del establecimiento de la conexión entre los conmutadores y el controlador.
- *Echo:* Se usan para medir la latencia o el ancho de banda o para verificar que un dispositivo esté activo. Por cada petición que llegue al destino, se generará una respuesta que será enviada al origen. Puede ser generado por el conmutador o el controlador.
- *Experimenter:* Permite dar funcionalidades adicionales a un conmutador OpenFlow, en general este paquete ha sido planeado para futuras versiones de OpenFlow.

1.3 MININET [7] [8]

Mininet es una herramienta de software que permite crear SDN usando Linux. Esta herramienta permite crear, personalizar e interactuar con un prototipo de SDN. Mininet tiene como característica que las configuraciones realizadas en la simulación pueden moverse a las implementaciones físicas sin mayores cambios; además es una solución poco costosa al estar disponible de manera gratuita, soporta topologías personalizadas y tiene una API basada en Python incluida. OpenFlow ha sido escrito, en su mayoría, en lenguaje Python; sin embargo tiene partes y componentes que han sido escritos en C.

La virtualización que usa se basa en procesos y además usa espacios de nombres, que permiten manejar procesos individuales, con interfaces, tablas de ruteo y tablas ARP separadas.

Como desventaja, Mininet depende del *kernel* de Linux, por lo que no puede ser implementado sobre otro sistema operativo, aunque actualmente se pretende dar soporte al uso de esta plataforma en otros sistemas operativos.

Existen otras herramientas en el mercado similares a Mininet, como por ejemplo, OpenFlow VMS. Sin embargo, Mininet se destaca por ser la herramienta más rápida de todas y la que tiene una mayor escalabilidad, por ello se la ha elegido para la etapa de simulación del presente Proyecto de Titulación.

Además, Mininet brinda una gran facilidad en la instalación, ya que es posible descargar una máquina virtual con el software ya instalado y configurado. Otra forma alternativa de instalarlo es hacerlo directamente sobre un sistema Ubuntu.

En el siguiente capítulo se realizará la simulación de una SDN con la herramienta Mininet. En ella se aplicarán la mayoría de fundamentos teóricos expuestos en el presente capítulo.

REFERENCIAS:

[1] Opennetworking. /www.opennetworking.org/images/stories/downloads/white-papers/wp-sdn-newnorm.pdf (Consultado el 6 de Noviembre de 2012)

[2] OpenFlow. http://www.openflow.org (Consultado el 6 de Noviembre de 2012)

[3] NOX. http://www.noxrepo.org/nox (Consultado el 9 de Noviembre de 2012)

[4] Web Service. http://en.wikipedia.org/wiki/Web_service (Consultado el 15 de Noviembre de 2012)

[5] POX. http://www.noxrepo.org/pox (Consultado el 15 de Noviembre de 2012)

[6] OpenFlow Stanford. https://openflow.stanford.edu (Consultado el 14 de Noviembre de 2012)

[7] Mininet. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet (Consultado el 23 de Noviembre de 2012)

[8] DREIER, Gustavo, Tutorial Mininet, Documento Electrónico http://www.cedia.org.ec/images/stories/documentos_descarga/CursoOpenFlowDeRedClara/ 06a-mininet.pdf (Consultado el 3 de Diciembre de 2012)

[9] DREIER, Gustavo, Introducción SDN y OpenFlow, Documento Electrónico http://www.cedia.edu.ec/images/stories/documentos_descarga/CursoOpenFlowDeRedClara/ 01-introduccion%20sdn.pdf (Consultado el 3 de Diciembre de 2012)

[10] DREIER, Controladores para OpenFlow, Documento Electrónico http://www.cedia.org.ec/images/stories/documentos_descarga/CursoOpenFlowDeRedClara/ 05a-controladores.pdf (Consultado el 3 de Diciembre de 2012)

[11] FlowVisor. https://openflow.stanford.edu/display/DOCS/Flowvisor (Consultado el 8 de Diciembre de 2012)

[12] Floodlight. http://www.projectfloodlight.org (Consultado el 8 de Marzo de 2013)

[13] Documentación de Floodlight. http://docs.projectfloodlight.org/display/floodlightcontroller (Consultado el 8 de Marzo de 2013)

[14] Especificación OpenFlow. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf (Consultado el 8 de Marzo de 2013)

CAPÍTULO II

2. SIMULACIÓN DE LA RED

2.1 COMPONENTES USADOS EN LA SIMULACIÓN

2.1.1 COMPONENTES DE HARDWARE

En el capítulo anterior se realizó una exposición resumida del marco teórico referente a las redes definidas por software, con el fin de obtener los elementos necesarios para realizar la simulación de una SDN utilizando la herramienta Mininet. A continuación se detallarán las herramientas y configuraciones necesarias para llevar a cabo dicha simulación.

Para la simulación de una SDN con la herramienta de software Mininet, se utilizó una computadora portátil Dell Inspiron 5520, que cuenta con las siguientes características:

- Procesador Intel Core i5 2.5 GHz.
- Sistema Operativo Windows 7 Home Premium Service Pack 1 de 64 bits.
- Memoria RAM de 8 GB.

La página web de Mininet [11] no proporciona información acerca de las características mínimas de hardware que se requieren para el funcionamiento del software, pero como se indicó en el capítulo anterior la instalación de esta herramienta puede hacerse en una máquina física o descargando una máquina virtual ya configurada, siendo el único requisito para una instalación en una máquina física tener un sistema operativo Ubuntu 11.0 o superior.

En la simulación se usó la opción de la máquina virtual con Mininet instalado [1] [2] [3] sobre un sistema operativo Ubuntu 11. El motivo por el que se decidió usar la máquina virtual con Mininet ya instalado y configurado es principalmente por facilidad de uso. Específicamente, para la opción de instalar Mininet en una máquina física, se requieren varias dependencias; por ejemplo, el controlador NOX, Wireshark y todo el código perteneciente a los tutoriales de OpenFlow que ya vienen incluidos en el caso de la opción de la máquina virtual [1]. Sin embargo, si se decidiese por la opción de la instalación en una máquina física sin hacer uso de la máquina virtual antes mencionada por cuestiones de mejor rendimiento o porque ya se tiene el sistema operativo instalado, se puede emplear un par de comandos que instalarán todos los componentes necesarios:

```
$git clone git://github.com/mininet/mininet
$mininet/util/install.sh –a
```

O para tener una versión simplificada de Mininet, se usan los comandos:

```
$git clone git://github.com/mininet/mininet
$mininet/util/install.sh -fnv
```

2.1.2 COMPONENTES DE SOFTWARE

Para la simulación de una red SDN con Mininet, los elementos de software requeridos son los siguientes [2] [3]:

- Un administrador de máquinas virtuales.
- Un sistema de ventanas X, que se instala en la máquina física que contiene el administrador de máquinas virtuales.
- Un cliente de Secure Shell (SSH), que también se instala en este dispositivo.

La instalación y configuración de las herramientas de software pueden encontrarse en detalle en el Anexo A. En la siguiente sección se probará el

funcionamiento de Mininet al crear la topología utilizada en el desarrollo del presente Proyecto de Titulación.

2.2 TOPOLOGÍA USADA EN LA SIMULACIÓN

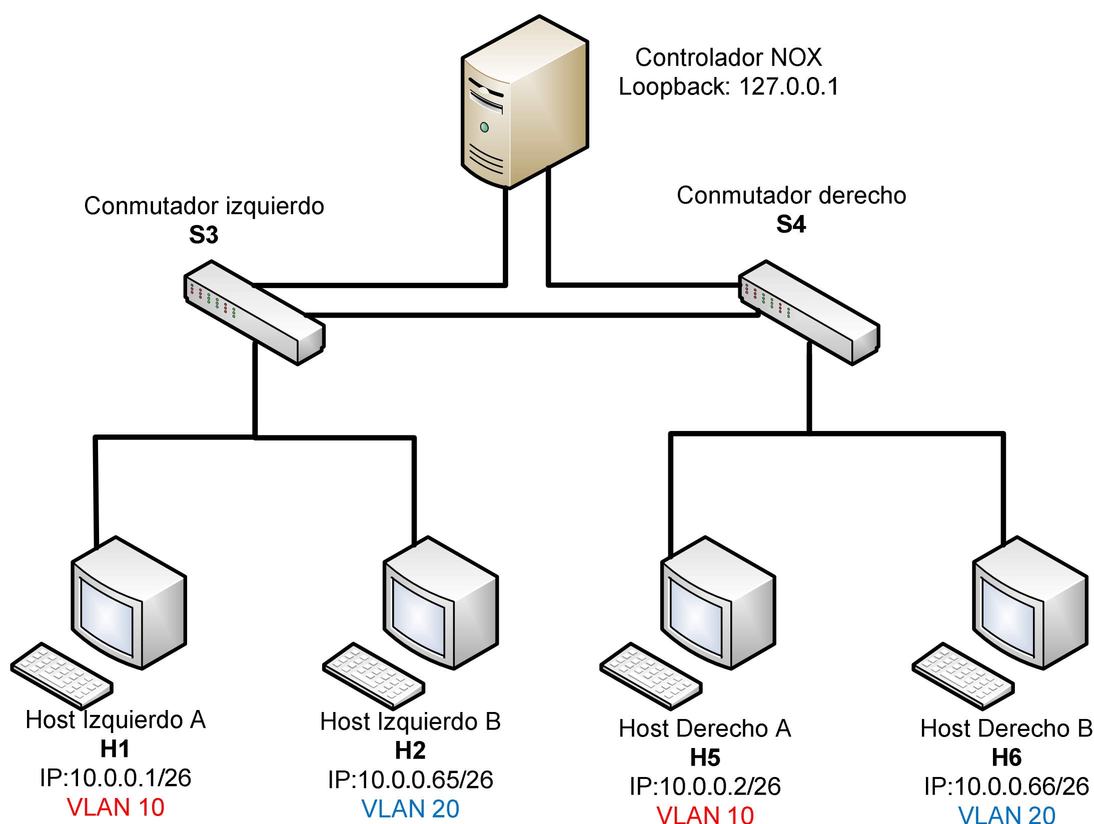


Figura 2.1 Topología usada para la simulación de la red

Para la topología de la SDN se tienen varias opciones: una topología en árbol (conmutadores conectados entre sí en varios niveles jerárquicos), lineal (conmutadores conectados entre sí en un nivel jerárquico) o estrella (conmutadores interconectados en forma de estrella). Para el presente proyecto se ha elegido una topología de dos conmutadores interconectados entre sí con dos hosts cada uno, como se indica en la Figura 2.1.

La selección de esta topología se da principalmente porque representa una red que podría ser usada en la práctica pero a una escala pequeña. En la mayoría de

redes se tiene una topología en árbol, similar a la topología que se usará pero con un mayor número de dispositivos y mayor complejidad. En la simulación e implementación se usarán solo conmutadores que manejen el protocolo OpenFlow, aunque pueden existir topologías con conmutadores que no lo manejen. Esto se debe a que el objetivo del presente Proyecto de Titulación es comprender el funcionamiento de OpenFlow y cómo poder usarlo; la integración con redes tradicionales está fuera del alcance fijado. La topología de la simulación es similar a la de la implementación del prototipo, por lo que se dará una clara muestra del funcionamiento de la red.

En el diagrama de topología de la Figura 2.1 se observan los siguientes dispositivos:

- Host izquierdo A (H1), perteneciente a la VLAN 10, conectado al conmutador S3.
- Host izquierdo B (H2), perteneciente a la VLAN 20, conectado al conmutador S3.
- Conmutador izquierdo (S3).
- Conmutador derecho (S4).
- Host derecho A (H5), perteneciente a la VLAN 10, conectado al conmutador S4.
- Host derecho B (H6), perteneciente a la VLAN 20, conectado al conmutador S4.

La conectividad entre hosts de diferentes VLAN no se implementará ya que se encuentra fuera del alcance del presente Proyecto de Titulación, aunque es posible implementarla a través de la definición de flujos de manera similar a la que se hará para la conmutación a nivel de capa 2 del modelo OSI. Sin embargo cada host podrá comunicarse con otro dentro de su misma VLAN. Con ello se podrán implementar y demostrar las tareas fundamentales que realiza un dispositivo capa dos del modelo ISO/OSI.

Los objetivos fundamentales de emplear esta topología son: a) Comprender cómo se realiza el intercambio de mensajes del protocolo OpenFlow y b) Entender el proceso de instalación de flujos en la tabla de flujos de cada dispositivo, que en este caso son los dos conmutadores habilitados para el uso de OpenFlow.

2.2.1 CREACIÓN DE TOPOLOGÍAS POR DEFECTO EN MININET

La primera forma de crear topologías de red con Mininet es hacer uso de las opciones disponibles por defecto [1] [2] [3]. La creación de estas topologías es sumamente fácil y rápida, pero limitada al momento de pretender hacer una topología personalizada. Las opciones de topologías incluidas en Mininet son:

- *Single* o única: Esta topología consiste de un único conmutador conectado a un número determinado de hosts. El comando ingresado en la máquina virtual de Mininet para la creación de esta topología se indica a continuación:

  ```
  $sudo mn –topo single,N
  ```

  En donde el parámetro N es el número de hosts que se desea tener conectados al conmutador.

  Una clara desventaja de esta topología es que el usuario se encuentra atado a usar un solo conmutador.
- *Linear* o lineal: Esta topología consta de un determinado número de conmutadores interconectados de forma lineal, como se indica en la Figura 2.2.

Figura 2.2 Topología lineal

Cada conmutador tiene un host conectado a él. Para crear esta topología se usa el siguiente comando:

```
$sudo mn –topo linear,N
```

En donde N es el número de conmutadores y hosts que se desean añadir a la topología.

A simple vista, se puede observar que esta topología permite un manejo más flexible en cuanto a la adición de conmutadores, pero tiene el gran limitante de que el número de conmutadores debe ser igual al número de hosts, ya que necesariamente cada host se conecta a un conmutador. Esto no permite crear topologías totalmente personalizadas.

- *Tree* o árbol: En este caso es posible crear una topología, como su nombre lo indica en forma de árbol. El comando para la creación de esta topología se muestra a continuación:

```
$sudo mn –topo tree,depth=N,fanout=M
```

Para esta topología se tienen dos variables: N que es el nivel de profundidad del árbol que se desea tener y M es un parámetro denominado *fanout* o esparcimiento. Así la topología se asemeja a un árbol, en donde N indica el número de niveles de conmutadores interconectados y M el número de hosts conectados a cada conmutador del último nivel.

Por ejemplo, si los parámetros M y N son iguales a dos, se tendrán dos niveles, el primero con un conmutador y el segundo con dos, y conectados a cada conmutador del nivel dos, se tendrán dos hosts. Esta distribución se puede observar de mejor manera en la Figura 2.3.

De las opciones de creación de topología presentadas anteriormente, ninguna es apropiada para crear la topología planteada para el desarrollo de la simulación. En la siguiente sección se presenta una manera para crear topologías totalmente personalizadas que se adaptará al escenario propuesto anteriormente.

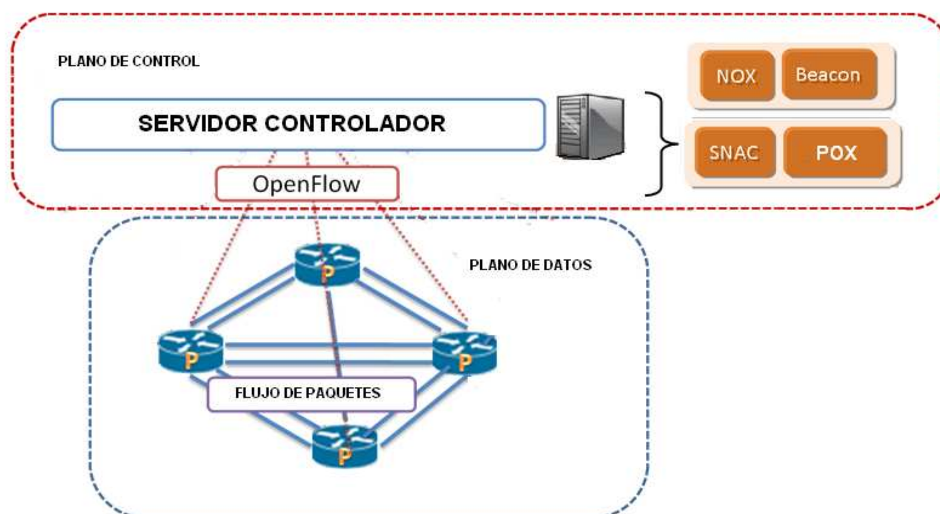


Figura 2.3 Topología árbol, con los parámetros M=2 y N=2

2.2.2 CREACIÓN DE TOPOLOGÍAS PERSONALIZADAS EN MININET

Si ninguna de las opciones presentadas anteriormente de creación de topología satisface los requerimientos, es necesario hacer un *script* básico en el lenguaje de programación Python para la creación de la topología requerida [1].

En la máquina virtual de Mininet se presentan varios ejemplos de topologías personalizadas que ya están creadas, y que pueden usarse como base para la creación de nuevas topologías. Estos *scripts* pueden encontrarse bajo el directorio ~/mininet/examples [3]. Sobre todo, las topologías que más ayudan para la personalización de una topología específica son emptynet.py y scratchnet.py[4].

Sin embargo, si se desea crear una topología de manera más rápida y sencilla, se puede modificar el archivo topo-2sw-2host.py que viene incluido en el directorio ~/mininet/custom [1]. Este archivo es fácil de editar y de comprender, por lo que en base a él se ha desarrollado la topología requerida.

[4] El *script* emptynet.py permite crear un objeto Mininet sin crear una topología para luego agregar nodos manualmente, mientras que el *script* scratchnet.py permite crear una red de un host y un controlador y agregar las rutas directamente.

En un principio este archivo consta de dos conmutadores interconectados entre sí; a cada conmutador se conecta un host. Como se puede deducir, la topología es prácticamente igual a la requerida, solo es necesario añadir dos hosts y conectarlos a cada conmutador.

En el Código 2.1 se presenta el *script* para la generación de la topología requerida en el presente Proyecto de Titulación, cuyo nombre de archivo es topologia.py:

```
"""Topología SDN Proyecto de Titulación

Implementación de un prototipo de una Red Definida por Software (SDN)
empleando una solución basada en hardware

Autor: Juan Carlos Chico

Descripción: Dos conmutadores conectados entre sí con dos hosts
conectados a cada conmutador. Cada conmutador se conecta con el
controlador SDN, del que recibe las reglas para crear patrones de flujos de
datos:

                  ←controlador→
  host --- conmutador --- conmutador --- host
  host ---l                              l--- host

La conexión entre los conmutadores y el controlador se realiza sin
necesidad de ser definida, porque el controlador se ejecuta en la misma
máquina (en este caso virtual) que se realiza la simulación.
"""


#Se importan las superclases Topo y Node a partir de las cuales se definirá
la topología y nodos a usarse en la simulación.
from mininet.topo import Topo, Node
```

Código 2.1 Topologia.py (continúa...)

```
#Definición de la clase MyTopo que hereda los atributos y métodos de la superclase Topo.
class MyTopo( Topo ):
    "Topología SDN Juan Carlos Chico"

#__init__ indica que este código se ejecutará luego de crear un objeto de la clase (instanciación).

#El parámetro self sirve para referirse al objeto actual y es necesario para poder acceder a atributos y métodos del objeto.

#El parámetro enable_all=True indica que se podrá habilitar los dispositivos llamando al método enable_all().
    def __init__( self, enable_all = True ):
        "creación de la topología"

# Se llama al método de inicialización (constructor de la clase base): se agregan los miembros por defecto a la clase. Miembros que ya se encuentran definidos y se usan durante la creación de la topología. Sin embargo no se los manipula directamente por lo que solo es necesario agregarlos.
        super( MyTopo, self ).__init__()

# Establecimiento de identificadores para conmutadores y hosts. Posteriormente, en el subcapítulo de pruebas, se usará el identificador de los conmutadores para enviar flujos al conmutador
        leftHosta = 1   #host izquierdo A conectado a conmutador izquierdo
        leftHostb = 2   #host izquierdo B conectado a conmutador izquierdo
        leftSwitch = 3    #conmutador izquierdo
```

Código 2.1 Topologia.py (continúa...)

```
        rightSwitch = 4  #conmutador derecho
        rightHosta = 5    #host derecho A conectado a conmutador derecho
        rightHostb = 6    #host derecho B conectado a conmutador derecho

# Se añaden los nodos, diferenciándolos entre conmutadores y hosts mediante el parámetro is_switch (True= Conmutador, False= Host)
        self.add_node( leftSwitch, Node( is_switch=True ) )
        self.add_node( rightSwitch, Node( is_switch=True ) )
        self.add_node( leftHosta, Node( is_switch=False ) )
        self.add_node( leftHostb, Node( is_switch=False ) )
        self.add_node( rightHosta, Node( is_switch=False ) )
        self.add_node( rightHostb, Node( is_switch=False ) )

# Agregando los enlaces entre hosts y conmutadores
        self.add_edge( leftHosta, leftSwitch )
        self.add_edge( leftHostb, leftSwitch )
        self.add_edge( leftSwitch, rightSwitch )
        self.add_edge( rightSwitch, rightHosta )
       self.add_edge( rightSwitch, rightHostb )

# Habilitando (encendiendo) a todos los dispositivos
        self.enable_all()

#Se crea el diccionario (hash) llamado topos, el cual dispone de un par llave(mytopo)/valor(lambda: MyTopo()) para generar la nueva topología, lamdba permite crear una función anónima, la cual usa la llave mytopo para llamar a la clase MyTopo().
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

Código 2.1 Topologia.py

La identificación de las interfaces (puertos físicos del conmutador) se la puede hacer fácilmente ya que en el *script* para la creación de la topología, los enlaces se agregan en orden ocupando una interfaz a la vez, a partir de la interfaz 1.

Para la ejecución del *script* en el que se crea la topología, se ingresa el siguiente comando, con la referencia **mytopo** que ejecuta la función anónima que llama a la clase **MyTopo():**

```
$sudo mn –custom ~/mininet/custom/topología.py –topo mytopo
```

Se debe verificar que no hayan errores de sintaxis; Mininet se ejecuta y se ingresa a la línea de comandos (CLI) propia de Mininet, tal como se muestra en la Figura 2.4.

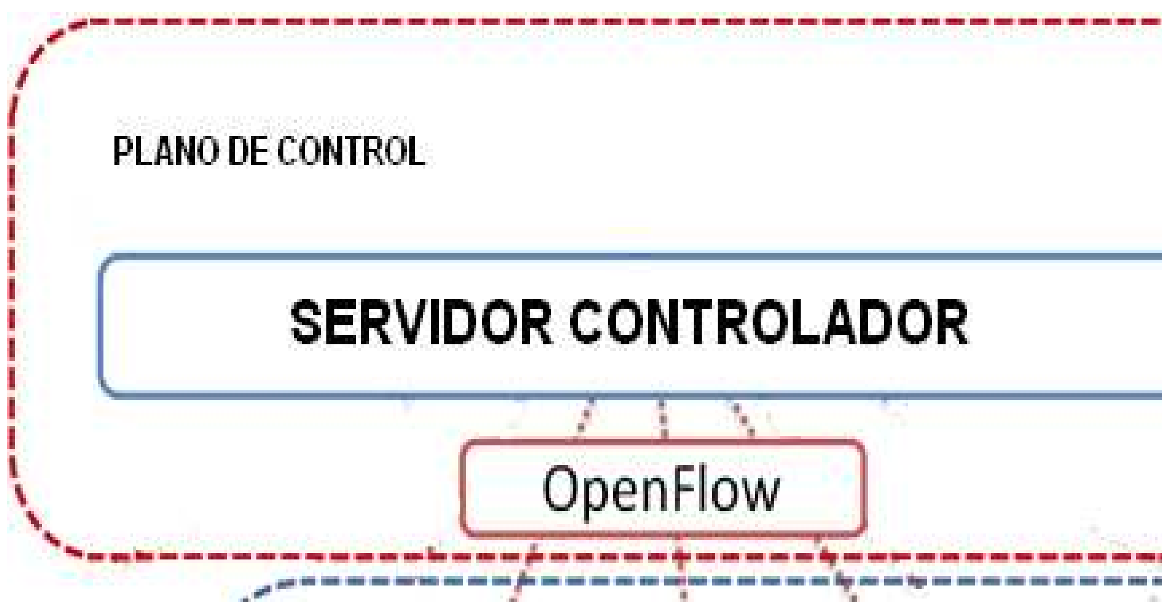


Figura 2.4 Creación de la topología en Mininet

2.2.3 CREACIÓN MEDIANTE MINIEDIT DE TOPOLOGÍAS PERSONALIZADAS

Miniedit [1] es un *script* escrito en Python que se encuentra incluido dentro de la máquina virtual de Mininet en el directorio ~/mininet/examples que permite la creación de topologías personalizadas a través de una interfaz gráfica.

Para su uso es necesario ejecutar el *script* miniedit.py, con la instrucción que se muestra a continuación:

```
$sudo python miniedit.py
```

Inmediatamente se abrirá la interfaz gráfica y se procede a agregar los nodos y enlaces de la topología, como se indica en la Figura 2.5:

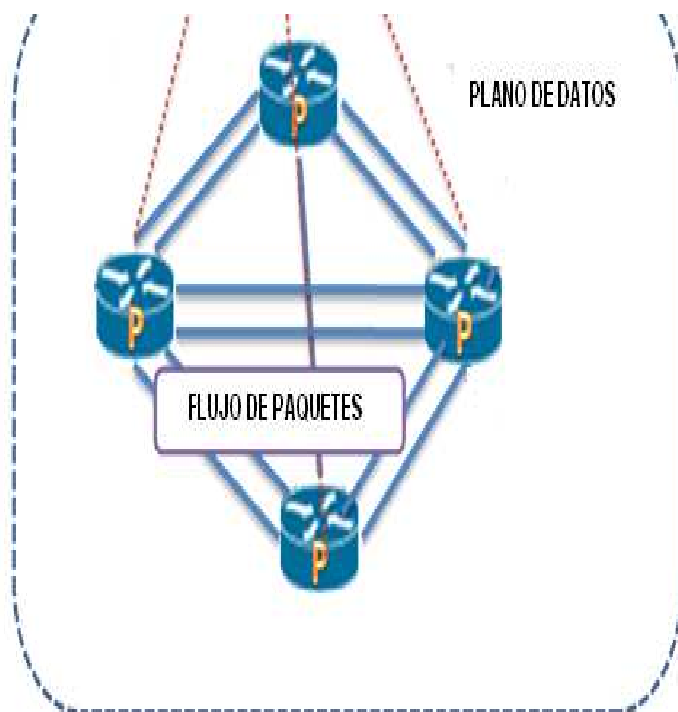


Figura 2.5 Interfaz gráfica de Miniedit

Se procede a ejecutar el *script* que generará la topología. Su uso es similar a la línea de comandos.

Sin embargo, de las pruebas realizadas con la herramienta Miniedit, se puede concluir que no es del todo estable, ya que luego de la primera ejecución y de agregar los dispositivos, el código del *script* se corrompe y la herramienta queda inutilizable para futuros usos, por lo que se la ha descartado para el desarrollo de la simulación. En su lugar se ha escogido la interfaz de línea de comandos para realizar todas las pruebas necesarias.

2.3 CONTROLADOR NOX DE MININET

La máquina virtual de Mininet incluye un controlador NOX [1] [8], por lo que no es necesario instalar un software controlador adicional. Esto ahorra tiempo y configuraciones, y permite probar la topología que se construyó anteriormente de manera inmediata.

Además, la máquina virtual incluye el código de componentes NOX de ejemplo que cumplen diferentes propósitos y funcionalidades. Estos códigos pueden ser encontrados bajo la carpeta ~/noxcore/src/nox [3], y son, por ejemplo, controladores que hacen actuar al dispositivo como concentrador, conmutador capa dos o tres del modelo OSI, dependiendo de las instrucciones que contengan y del tratamiento de los paquetes.

Para usar NOX, primero se debe comprobar que el software se ejecuta correctamente. Para ello, en el directorio ~/noxcore/build/src, se ingresa el comando que se muestra a continuación:

```
$./nox_core –v –i ptcp:
```

En donde **–i ptcp:** hace referencia a la interfaz por la cual escuchará el controlador, en este caso no se añade la dirección IP por ser la dirección *loopback*, el protocolo que se usará para la comunicación entre el controlador y los conmutadores es TCP (identificado por ptcp) y el parámetro **–v** indica que se requiere obtener información de depuración detallada cuando el controlador realice una acción determinada (información de *debug*). En la Figura 2.6 se muestra la ejecución de NOX.

Figura 2.6 Ejecución del controlador NOX

El controlador NOX en Mininet se ejecutará por defecto con el componente pyswitch que hará comportarse a los conmutadores como un dispositivo capa dos de una red tradicional. Si se desea ejecutar el controlador con componentes específicos, como el componente que se creará posteriormente en el subcapítulo de pruebas, es recomendable abrir dos terminales para ejecutar el controlador en la primera y ejecutar el *script* para la creación de la topología en la segunda. El intercambio de mensajes OpenFlow entre los conmutadores y el controlador se discutirá en el subcapítulo de pruebas.

2.4 COMANDO *DPCTL*

Como alternativa al controlador NOX, se puede usar el comando *dpctl* [9], que es una utilidad de la línea de comandos que permite enviar mensajes destinados a crear y manipular flujos OpenFlow desde cualquier host o conmutador que entienda el protocolo OpenFlow al conmutador al que se desea modificar los flujos sin necesidad de tener un controlador.

Los mensajes que serán usados son principalmente del tipo *add-flow*, que permite añadir un flujo especificado a la tabla de flujos del dispositivo, *mod-flow* que permite modificar el flujo agregado anteriormente y *del-flow* que permite eliminar el flujo.

La sintaxis del comando **dpctl add-flow** se detalla a continuación:

#**dpctl** **add-flow** [protocolo]:[ip_conmutador]:[puerto] nw_proto =[protocolo_filtrado],nw_src=[ip_origen],nw_dst=[ip_destino],dl_src=[mac_origen], dl_dst=[mac_destino],tp_src=[puerto_origena],tp_dst=[puerto_destino],in_port=[#puerto_entrada],idle_timeout=[tiempo_expiracion_inactividad],hard_timeout[tiempo_expiracion],dl_vlan=[VLAN],priority[prioridad],**actions**=[output:#puerto_salida]/[normal]/[flood]/[enqueue:puerto:id]/[all]/[controller]/[mod_vlan_id:id]

En donde:

- **[protocolo]:** es el protocolo mediante el cual se realizará la comunicación, que podrá ser SSL cuando se quiere una conexión segura, Unix cuando se trabaja en una máquina local o TCP cuando se requiere transmisión sin cifrado, siendo el protocolo más usado TCP y el que se usará en la comunicación por facilidad de comprensión [10].
- **[ip_conmutador]:** es la dirección IP asignada al conmutador, en este caso es la dirección de *loopback*.
- **[puerto]:** define el puerto en el que el controlador escucha los mensajes OpenFlow.
- **[protocolo_filtrado]:** se usa para filtrar los paquetes de un determinado protocolo, como por ejemplo IP, HTTP, etc.
- **[ip_origen]** e **[ip_destino]**: se usan para caracterizar al flujo, es decir, clasificarlo de acuerdo a la dirección IP que origina la transmisión y la dirección de destino. Se puede usar una dirección de red con máscara para especificar un rango de direcciones.
- **[mac_origen]** y **[mac_destino]**: son las direcciones físicas MAC (capa dos del modelo OSI) de los dispositivos ya sea de origen o de destino de la comunicación.
- **[puerto_origen]** y **[puerto_destino]**: son los identificadores de número de puerto TCP o UDP ya sea de origen o de destino de la comunicación.
- **[#puerto_entrada]:** es el puerto por el que ingresan los paquetes de un determinado flujo, y servirá para definir y caracterizar el flujo.
- **[tiempo_expiracion_inactividad]**: es el tiempo en el que un determinado flujo se mantiene en la tabla de flujos antes de ser eliminado por inactividad. Si no se envían paquetes de este flujo en este periodo, el flujo se elimina de la tabla. Se usa cero (0) para flujos permanentes.
- **[tiempo_expiracion]**: es el tiempo en el que un determinado flujo se mantiene en la tabla de flujos antes de ser eliminado ya sea que hayan llegado paquetes de ese flujo o no. Se usa cero (0) para flujos permanentes.

- **[VLAN]:** es la etiqueta de una determinada VLAN que el tráfico entrante podría tener
- **[prioridad]:** define la prioridad con la que se tratará el paquete.

Opciones de tratamiento del paquete (**actions**):

- **[output:#puerto_salida]:** es el puerto al cual se desea redirigir el tráfico. Por ejemplo el flujo llegó al puerto denominado eth0 y se desea reenviarlo a un equipo conectado al puerto eth2.
- **[normal]:** el conmutador le da un comportamiento normal de capa 2/3 al paquete.
- **[flood]:** Envía los paquetes a todas las interfaces del conmutador, excepto por la que el paquete entro y las que tienen la opción *flooding* deshabilitada.
- **[enqueue:puerto:id]:** especifica que se pondrá en cola al paquete. Como parámetros se define el puerto físico de salida y un identificador para monitoreo del paquete.
- **[all]:** envía el paquete a todas las interfaces (puertos), menos a la interfaz por la que el paquete entró.
- **[controller]:** envía el paquete al controlador, para que este decida qué hacer con él.
- **[mod_vlan_id:id]:** sirve para modificar el identificador de la VLAN cuando se redirige el paquete hacia la interfaz de salida.

En el conmutador, el principal comando a ser utilizado es *dump-flows*, que imprimirá en pantalla todos los elementos que se encuentran presentes en la tabla de flujos. Para ello primero se debe abrir un terminal para s3, con el comando:

```
>xterm s3
```

Ejecutado en la línea de comandos de Mininet; y, finalmente, se introduce el comando:

```
#dpctl dump-flows tcp:<ip controlador:puerto>
```

Desde el terminal para s3 recientemente abierta. Como no se ha agregado ningún flujo, el conmutador tendrá su tabla de flujos vacía, tal como se muestra en la Figura 2.7. Posteriormente se podrá observar la información de flujos cuando se los haya agregado.

Figura 2.7 Tabla de flujos vacía del conmutador s3

Para que el dispositivo OpenFlow se comporte como un conmutador capa dos del modelo ISO/OSI, es necesario definir flujos desde un host hacia otro. Para ello se deben agregar los siguientes flujos [2] [9] dirigidos al conmutador de la izquierda identificado como s3:

- ```
  dpctl add-flow tcp:127.0.0.1:6634 ip, nw_src=10.0.0.1, nw_dst=10.0.0.2,priority=10,idle_timeout=60,hard_timeout=0,actions= output:3
  ```

  Este comando permite que el tráfico que ingresa desde el host h1 (10.0.0.1) y que se dirige hacia el host h5 (10.0.0.2), sea direccionado al puerto físico número tres (identificado como eth3) que se conecta con el conmutador s4. En primer lugar, **add-flow** indica que se añadirá un flujo, posteriormente **tcp:127.0.0.1:6634** indica la dirección IP del conmutador y el puerto con el que *dpctl* creará la conexión TCP con el conmutador, en este caso como todo es local se usa la dirección de *loopback*, a continuación se ingresan las direcciones IP de origen y destino mediante los parámetros **nw_src** y **nw_dst** respectivamente. Opcionalmente se agrega la prioridad que se dará al paquete, en el caso de que se desee
  ```

implementar calidad de servicio. Finalmente se define la acción a realizar por el conmutador para este flujo, que en este caso será la redirección a un puerto de salida.

- ```
  dpctl add-flow tcp:127.0.0.1:6634 ip, nw_src=10.0.0.65,
  nw_dst=10.0.0.66,priority=10,idle_timeout=60,hard_timeout=0,actions=
  output:3
  ```

  Este comando permite que el tráfico que ingresa desde el host h2 (10.0.0.65) hacia el host h6 (10.0.0.66), sea direccionado al puerto físico número tres que se conecta con el conmutador derecho s4.

Lo mismo se realiza en el conmutador de la derecha, identificado como s4. El puerto en el que escucha normalmente el controlador las peticiones de los dispositivos es el 6633, pero se lo puede cambiar de acuerdo a las necesidades.

Por defecto en Mininet el controlador escucha peticiones en el puerto 6634, pero en una implementación real se puede usar cualquier puerto para la comunicación con los conmutadores, como se verificará en el siguiente capítulo.

- ```
  dpctl add-flow tcp:127.0.0.1:6634 ip,nw_src=10.0.0.2,
  nw_dst=10.0.0.1,priority=10 , idle_timeout=60,hard_timeout=0,
  actions=output:1
  ```
- ```
  dpctl add-flow tcp:127.0.0.1:6634 ip,nw_src=10.0.0.66,
  nw_dst=10.0.0.65,priority=10, idle_timeout=60,
  hard_timeout=0,actions=output:1
  ```

Finalmente, se verifica en cada conmutador que los flujos se hayan agregado, para ello se abre un terminal y se ingresa el siguiente comando:

```
>xterm s3 s4
```
```

Como resultado, se desplegarán dos interfaces de línea de comandos correspondientes a los conmutadores s3 y s4. En el terminal de s3 se verifica la tabla de flujos usando el comando que se indica a continuación:

```
#dpctl dump-flows tcp:127.0.0.1:6634
```

Aparecerá la información de los dos flujos ingresados anteriormente, como se muestra en la Figura 2.8:

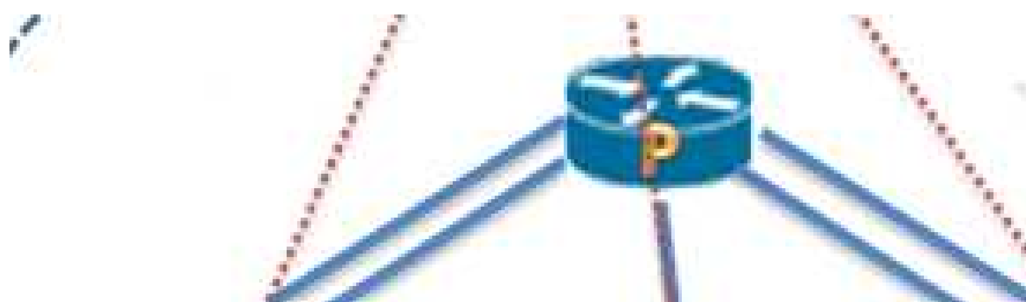


Figura 2.8 Flujos ingresados en la tabla de flujos del conmutador s3

Como resultado se muestra la información de los flujos. En primer lugar el identificador **cookie** que indica si la información del flujo se ha guardado anteriormente o no, **duration** especifica cuánto tiempo ha estado el flujo en la tabla del conmutador (duration_sec indica la parte entera de la duración en segundos y duraton_nsec indica la parte decimal en nano segundos. Así, el primer flujo ha estado en la tabla 46.619s), el identificador **table_id** se encarga de identificar la tabla de flujos, en caso de que existan varias, **priority** es la prioridad con la que se tratan los mensajes de cada flujo, **n_packets** y **n_bytes** indican la cantidad de paquetes o bytes que se han recibido por un determinado flujo, y el resto de información corresponde a los parámetros definidos en el flujo. Posteriormente, en el terminal de s4 se ingresa el siguiente comando:

```
#dpctl dump-flows tcp:127.0.0.1:6634
```

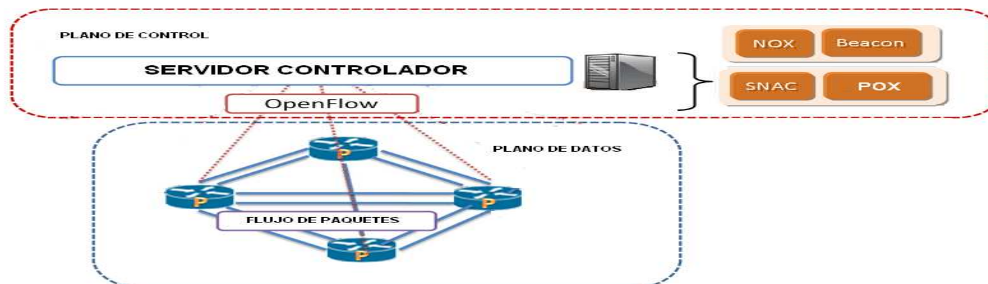


Figura 2.9 Flujos ingresados en la tabla de flujos del conmutador s4

Y se verifica la información de los flujos ingresados anteriormente, como se muestra en la Figura 2.9. El comando **dpctl dump-flows** es muy útil para obtener información de los flujos instalados, mediante las reglas definidas para cada flujo, de los contadores de tiempo que informan cuánto ha durado el flujo en la tabla de flujos y de las acciones a realizar con cada flujo.

Mediante el comando *dpctl* se pueden realizar muchas otras acciones para moldear el tráfico según las necesidades de los usuarios. Si se desea modificar el flujo ya instalado, se puede usar el comando *dpctl flow_mod*, que tiene la misma sintaxis del comando *add-flow*. Por ejemplo, cuando un host cambia de dirección IP o cuando cambia de puerto es necesario modificar el flujo para que exista comunicación.

Adicionalmente, al igual que un conmutador tradicional, es posible manipular los puertos físicos del conmutador habilitado para OpenFlow. Esto se realiza mediante el comando

```
#dpctl mod-port [protocolo]:[ip_controlador]:[puerto] [puerto_fisico] [estado]
```

En donde:

- **[puerto_fisico]:** es el identificador de puerto físico del conmutador.
- **[estado]:** es el comportamiento que tiene el puerto, y puede ser *down* cuando se desea apagar el puerto, *up* cuando se lo desea levantar, *flood*

cuando se desea que el puerto participe cuando se requiera una inundación de mensajes y *noflood* cuando no se desea que participe.

Para comprobar el funcionamiento de este comando, se ejecuta desde la terminal de Mininet, las instrucciones:

```
>s3 dpctl mod-port tcp:127.0.0.1:6634 2 down
>s3 dpctl mod-port tcp:127.0.0.1:6634 3 noflood
```

El resultado de la ejecución de estos comandos se puede observar en la Figura 2.10.

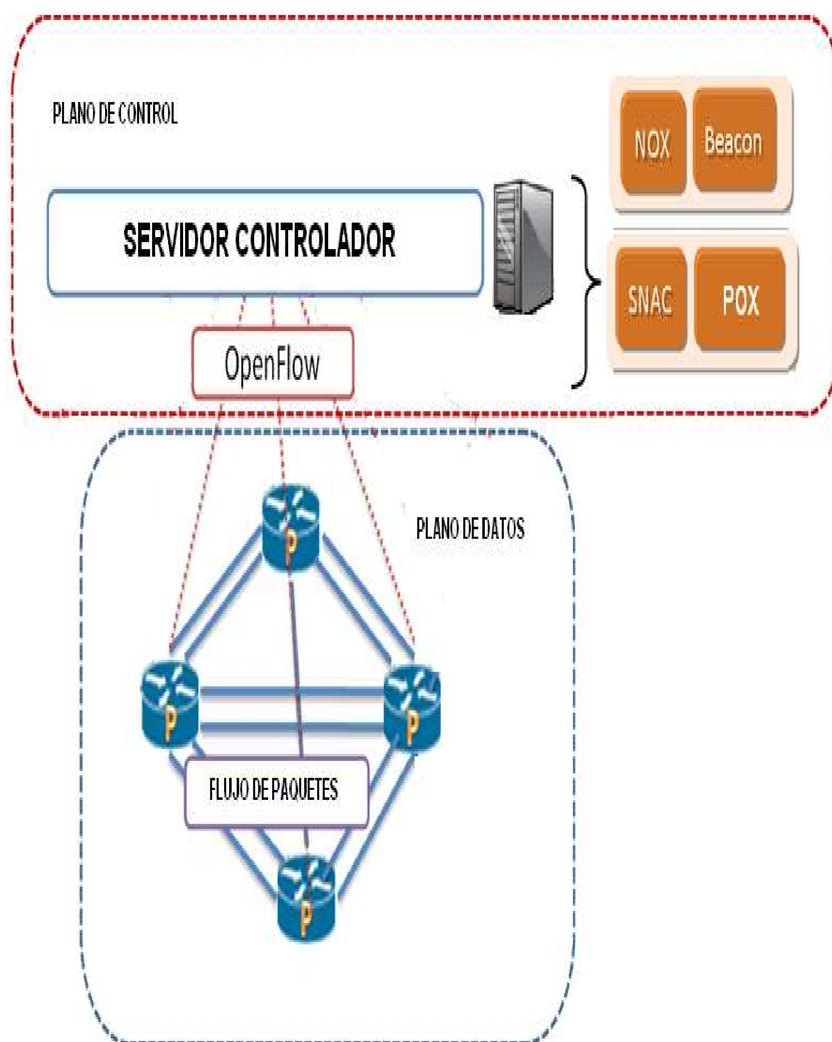


Figura 2.10 Ejecución del comando *mod-port* y su resultado en la interfaz

En la Figura 2.10 se muestra el resultado de la operación de bloqueo del puerto eth2 y paso al estado *noflood* del puerto eth3. Se puede observar que el puerto dos pasa a una configuración y estado de puerto apagado, representado por el identificador **0x1**, mientras que el puerto tres pasa a un estado *noflood* representado por **0x10** en la configuración.

Se puede verificar que el puerto eth2 está apagado ya que no incluye la palabra UP en la segunda línea que muestra el comando *ifconfig*. El estado *noflood* no permite enviar paquetes por una determinada interfaz cuando existe un proceso de inundación de paquetes.

Un ejemplo para comprobar el estado *noflood* de un puerto es temporalmente cambiar la dirección de h2 a 10.0.0.2, sin controlador y con la tabla de flujos vacía agregar los siguientes flujos en el conmutador s3:

```
$dpctl add-flow tcp:127.0.0.1:6634 nw_src=10.0.0.1,actions=flood
$ dpctl add-flow tcp:127.0.0.1:6634 nw_src=10.0.0.2,actions=flood
```

Y se realiza una prueba *ping* desde h1 a h2, la cual es exitosa, ya que con las reglas anteriores el dispositivo se comportará como repetidor. Ahora se ubica al puerto 2, que es el enlace con h2 en estado de *noflood* y las peticiones ICMP[5] fallarán de inmediato, debido a que antes de enviar peticiones ICMP se envían mensajes ARP[6], los cuales utilizan un proceso de inundación que se ha deshabilitado para este puerto. El resultado se muestra en la Figura 2.11.

Figura 2.11 Envío de paquetes ICMP fallido, puerto 2 de s3 en estado *noflood*

[5] *Internet Control Message Protocol*, es un protocolo de control del Protocolo de Internet IP.
[6] *Address Resolution Protocol*, protocolo que resuelve las direcciones de capa de red (IP) a direcciones de capa de enlace de datos (MAC).

Ahora, en vez de usar esta herramienta, se puede programar el controlador NOX para que envíe flujos definidos por el usuario de manera automática cada vez que un conmutador se conecte mediante la creación de un componente programado por el usuario. En la siguiente sección se indicará como se programó este controlador y se realizarán las pruebas de la red para verificar el intercambio de mensajes OpenFlow.

2.5 PRUEBAS Y RESULTADOS

Para la prueba de conectividad se empleará el comando *ping*. Se comprobará que es posible enviar paquetes ICMP entre los hosts de una determinada VLAN y que no será posible comunicarse con hosts de otras VLAN.

En primer lugar, es recomendable comenzar la captura de paquetes OpenFlow en segundo plano con el software Wireshark en la máquina virtual, para ello se ingresa el comando que se muestra a continuación en la línea de comandos:

```
$sudo wireshark &
```

Se abrirá la ventana del programa y, posteriormente, se iniciará una captura en la interfaz de *loopback,* usando como filtro la expresión "*of*", para paquetes OpenFlow.

En segundo lugar se ejecuta el *script* para la creación de la topología a usarse, con el comando:

```
$sudo mn –custom ~/mininet/custom/topología.py –topo mytopo
```

Desde la línea de comandos de Mininet se abren consolas para cada host y cada conmutador con el comando:

```
>xterm h1 h2 s3 s4 h5 h6
```

En cada nodo se configura la dirección IP correspondiente, con el comando:

```
#ifconfig <host-eth0> <dirección IP> netmask <máscara de subred>
```

En donde el parámetro host es el identificador del host, por ejemplo h1, h2, h5 o h6. Así, para el host h1 que tiene una tarjeta de red identificada como eth0, el parámetro a ingresar sería h1-eth0. La configuración de la dirección IP de cada host se muestra en la Figura 2.12.

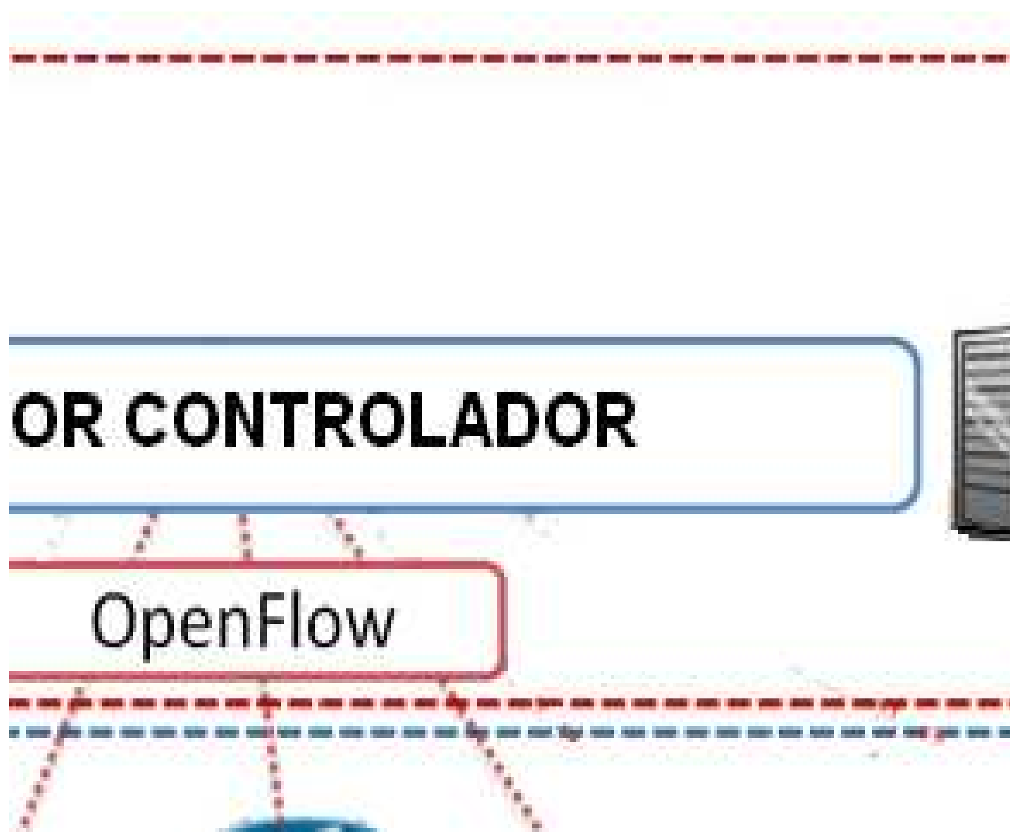


Figura 2.12 Configuración de los hosts

2.5.1 PRUEBAS CON EL CONTROLADOR NOX USANDO EL COMPONENTE PYSWITCH

Ahora con el controlador NOX ejecutándose con el componente de conmutador capa dos (pyswitch) que se inicia por defecto con Mininet se ejecuta el comando *ping* desde el terminal del host h1. Como se puede observar en la Figura 2.13, el comando *ping* hacia el otro host en la VLAN 10 es exitoso.

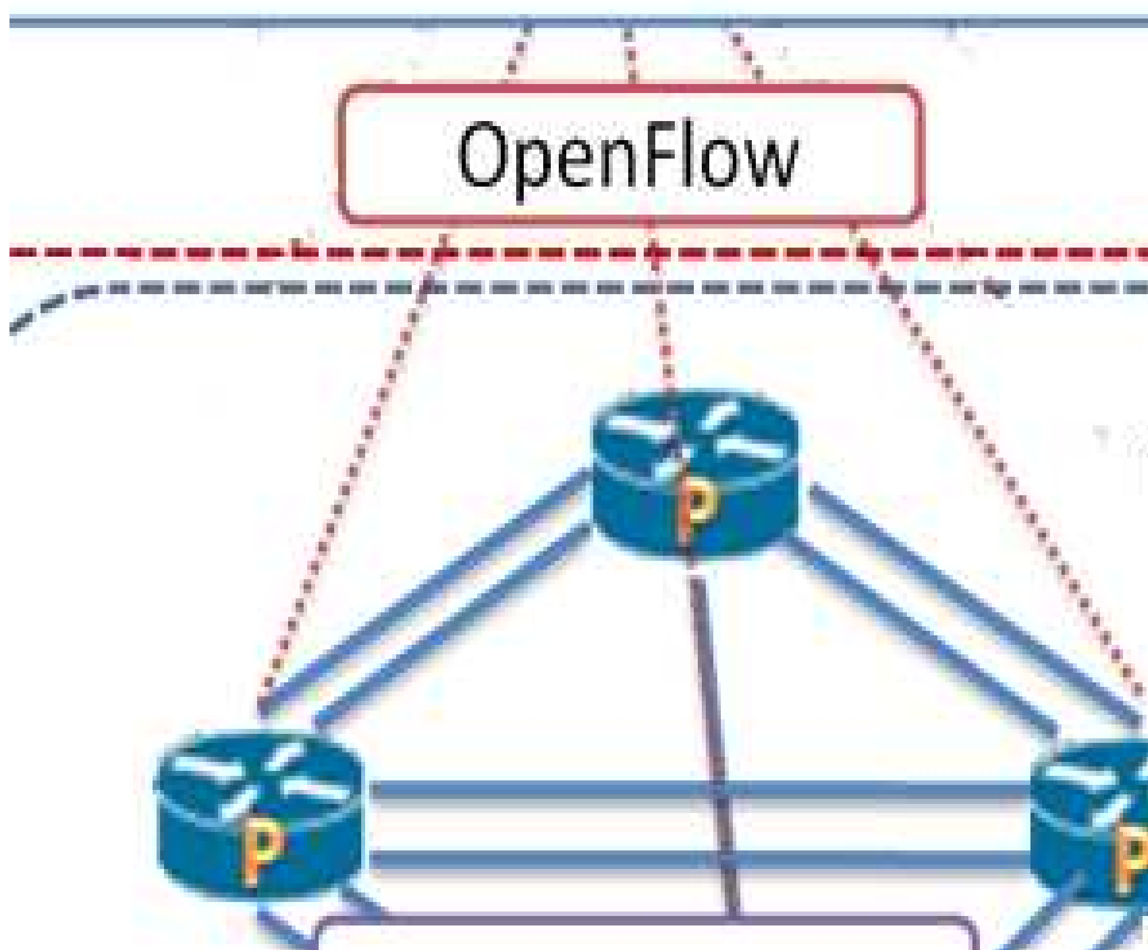
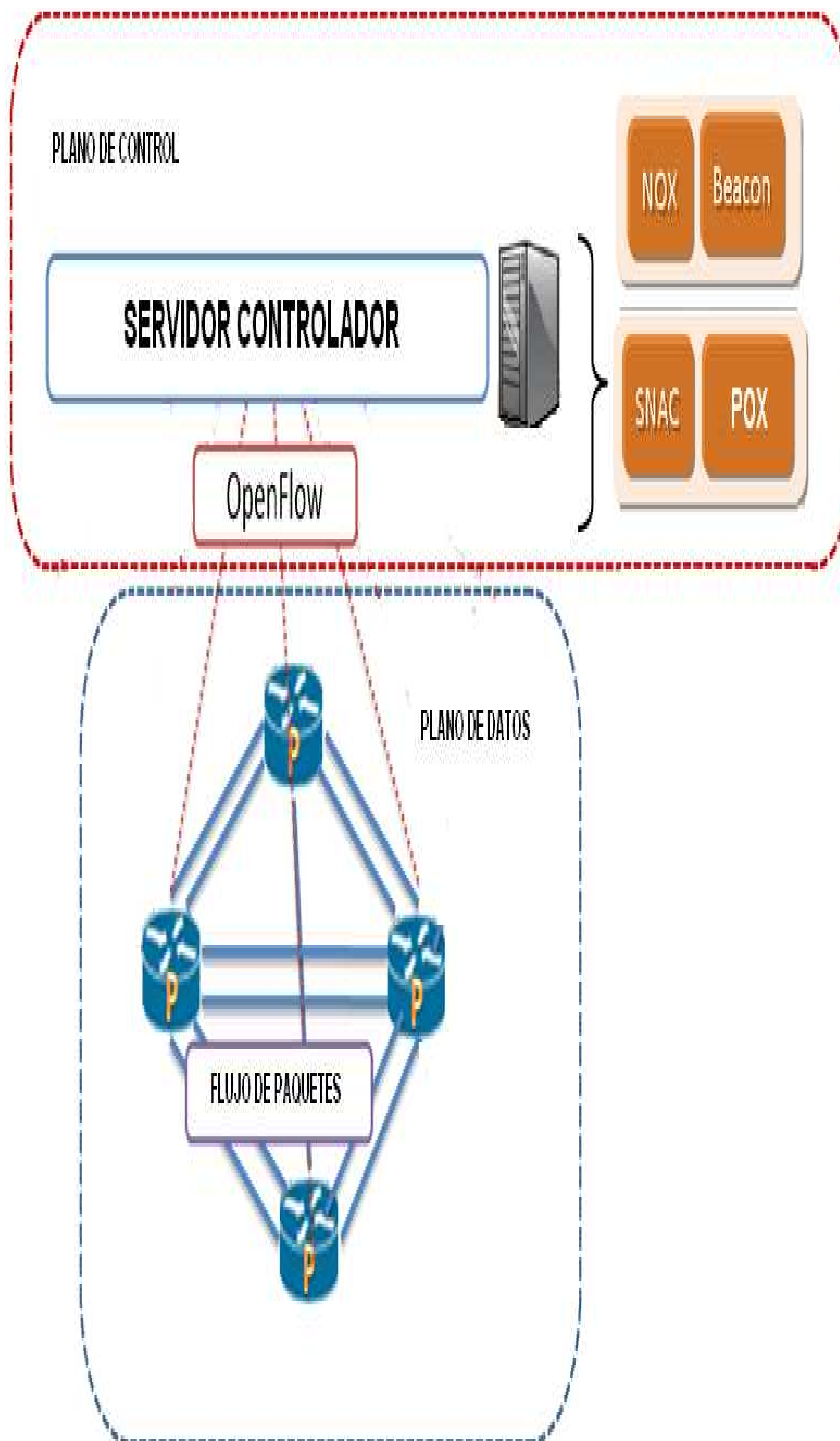


Figura 2.13 Envío de mensajes ICMP entre dos hosts de la VLAN 10

El primer paquete se transmite en un tiempo mayor debido a que antes de las peticiones ICMP se realiza una petición ARP, pero como los dos conmutadores no tienen flujos definidos, los conmutadores envían la petición al controlador, que inunda la red con paquetes ARP hasta que se encuentra una respuesta, con lo cual el controlador agrega los flujos respectivos de acuerdo a las direcciones de la petición y respuesta recibidos en ambos conmutadores.

Este proceso es similar al que realiza un conmutador capa dos tradicional, salvo que ya no existe la tabla de conmutación, pero en su lugar existen reglas enviadas por el controlador para manipular el tráfico ARP y permitir que las peticiones y respuestas lleguen a su destino.


PLANO DE CONTROL
NOX
Beacon
SERVIDOR CONTROLADOR
SNAC
POX
OpenFlow
PLANO DE DATOS
FLUJO DE PAQUETES

Una vez resuelta la petición ARP, se transmiten las peticiones ICMP, pero como no hay flujos para estas, los dos conmutadores envían la primera petición ICMP al controlador para que este les proporcione un flujo de manera similar a lo que se hizo con ARP. Una vez proporcionado el flujo se envían las peticiones al destino siguiendo las entradas especificadas anteriormente en la tabla de flujos sin pasar por el controlador.

En este proceso, la red SDN no es personalizable. Sin embargo, posteriormente se programará un componente que hará que la red sea totalmente personalizable; es decir que no actúe como un conmutador normal, sino que su comportamiento se base en las reglas especificadas en el componente por el administrador.

Desde el segundo paquete se nota una reducción considerable en el tiempo RTT[7] de los paquetes, debido a que los flujos ya se encuentran instalados y en uso en los conmutadores y los hosts ya cuentan con la información de las direcciones MAC en sus tablas ARP.

En la Figura 2.13 se pueden observar tiempos de 0 ms, debido a que el RTT es muy corto para entrar en la escala especificada, esto se debe a que todo se realiza de manera local y no se consideran los retardos por transmisión.

Consecutivamente se envía otro paquete ARP encapsulado en OpenFlow (número 1258) que corresponde al mismo proceso de *broadcast*, esta vez enviado desde el conmutador s4. Este paquete llega al controlador, el cual responderá con un mensaje *PacketOut* que también indica un proceso de inundación de la red.

Para cada mensaje ARP en los conmutadores s3 y s4, se envía un mensaje *PacketOut* (paquetes número 1256 y 1259), que indica que se está enviando el paquete de vuelta a los conmutadores respectivos con instrucciones específicas de reenvío (inundar la red), en este caso por el enlace entre los conmutadores s3 y s4 en primer lugar y luego por el enlace entre s4 y el host h2.

[7] *Round Time delay Time*, tiempo que tarda un paquete en ir del emisor al destinatario y regresar.

Una vez encontrado el host con dirección 10.0.0.2 (h5) se retorna el resultado de la petición hacia h1 pasando por los conmutadores s3 y s4 que nuevamente al no tener reglas para estos flujos, envían el mensaje al controlador en busca de acciones a realizar con el flujo, el cual responde con las acciones correspondientes (paquetes número 1261 1262 1263 y 1264). Esta vez se añaden los flujos mediante paquetes *Flow-Mod,* ya que se cuenta con la información de las direcciones lógicas y físicas de los hosts origen y destino y los puertos físicos por los cuales se logran comunicar.

Posteriormente se realiza el mismo proceso, pero esta vez con mensajes OpenFlow que encapsulan mensajes ICMP. Estos mensajes, al ser de diferente tipo que los ARP también deben ser enviados al controlador para que encuentre una ruta para ellos, pero como ya se conocen las direcciones y puertos físicos se añaden los flujos sin necesidad de un paquete *Packet-Out*.

Los mensajes *FlowMod* permiten al controlador añadir en la tabla de flujos de los conmutadores los flujos requeridos, es por ello que la principal característica de una SDN es tener una red altamente escalable y flexible, que se adapta rápidamente a las necesidades que surgen en un determinado instante de tiempo.

Si por un camino existe demasiado tráfico y los retardos aumentan, el controlador puede redirigir el tráfico a caminos alternativos y así prevenir pérdida de datos y tener menos retardos en la red para mantener la calidad de servicio de acuerdo al tipo de tráfico.

En la Figura 2.15 se muestra el detalle del primer paquete OpenFlow (numero 1265) que encapsula el mensaje ICMP. Este paquete se envía desde el conmutador s3 hacia el controlador, y solicita que se le proporcione una regla de flujo para las peticiones ICMP entre los hosts h1 y h5, lo que se evidencia claramente en el motivo del mensaje (*Reason sent: No matching flow* (0)). Posteriormente el controlador le proporcionará un camino mediante un mensaje *Flow Mod* (número 1266).

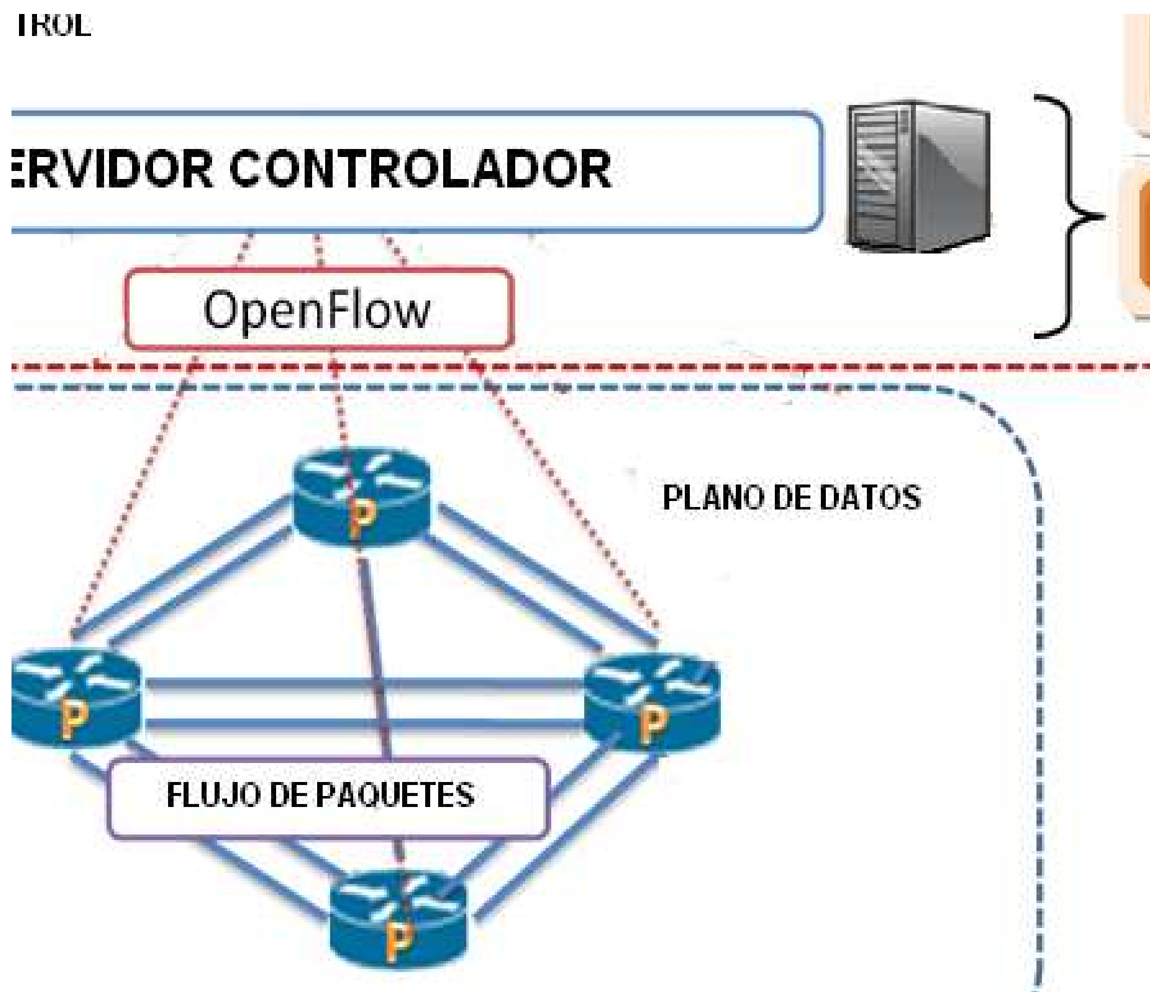


Figura 2.15 Detalle de un paquete OpenFlow

Como se puede observar en la Figura 2.16, luego de un cierto tiempo ya no se envían mensajes *Flow-Mod*, ya que el flujo es constante en el envío y recepción de paquetes. En esta captura se muestran los mensajes que se envían entre los conmutadores s3 y s4, por lo que se muestran como mensajes OpenFlow y no como tráfico ICMP, en realidad el tráfico ICMP es encapsulado sobre OpenFlow cuando este atraviesa la SDN, que en este caso se compone de dos conmutadores.

Posteriormente, la cabecera OpenFlow se elimina cuando el paquete ha traspasado la red y llega al destino como un paquete ICMP corriente. Por cada petición existe una respuesta que se encapsula de la misma manera.

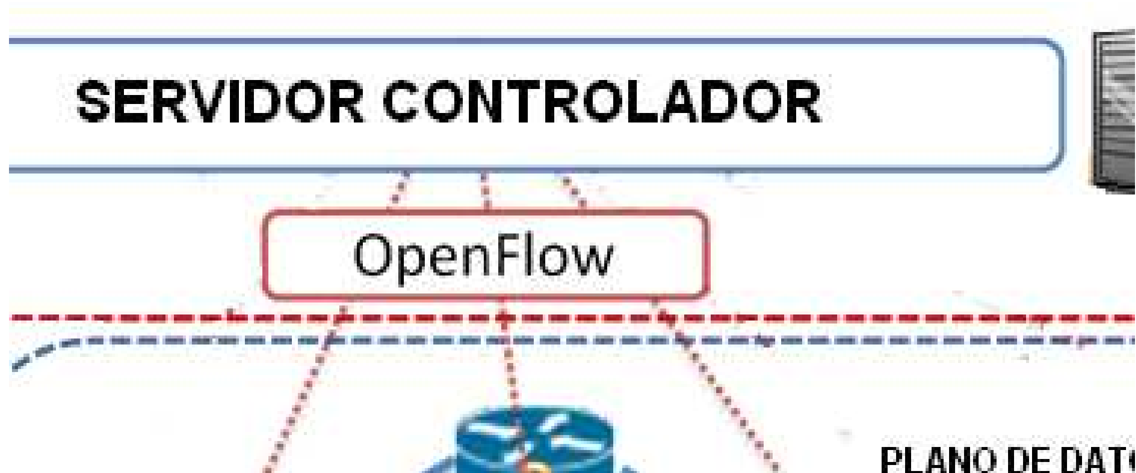


Figura 2.16 Captura de mensajes OpenFlow

Ahora se hará la misma prueba entre los hosts de la VLAN 20, como se muestra en la Figura 2.17.

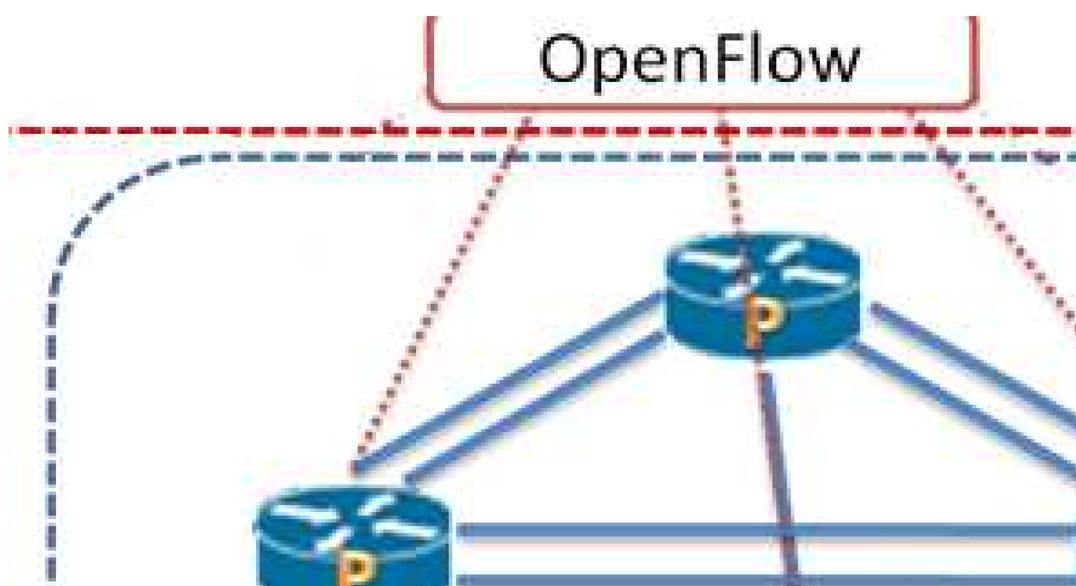


Figura 2.17 Envío de mensajes ICMP entre dos hosts de la VLAN 20

Una vez más se verifica que la primera petición se demora un mayor tiempo hasta modificar el flujo respectivo y realizar la petición ARP. Los siguientes paquetes se transmiten en un menor tiempo. Algunos, en este caso, también se encuentran fuera de la escala de medición de tiempo por lo que se reporta como 0ms.

El conmutador, en estos casos, se torna un dispositivo sin inteligencia, que solo conmuta tráfico que considera genérico, ya que no conoce qué tipo de tráfico es, a partir de las órdenes que el controlador le envía. En la siguiente sección se programará un componente que enviará solo los flujos especificados por el administrador al controlador. Las pruebas de captura con Wireshark del comando *dpctl* se harán en el capítulo tres, para tener una mejor idea de cómo se intercambian los mensajes, esta vez entre dispositivos físicos que tienen sus propias direcciones y no comparten la dirección de *loopback*.

2.5.2 PRUEBAS CON EL CONTROLADOR NOX USANDO UN COMPONENTE PERSONALIZADO

Ahora se realizarán las pruebas con un componente programado a la medida de las necesidades del usuario. Para ello en primer lugar se deben definir las tareas que el conmutador realizará. En este caso se permitirá el envío de mensajes ICMP entre los hosts h1 y h5 y se bloqueará otro tipo de tráfico que no sea ARP. Entonces las reglas a definir serían:

- Permitir los mensajes ARP entre los hosts h1 y h5.
- Permitir los mensajes ICMP entre los hosts h1 y h5, denegar cualquier otro tipo de tráfico entre los otros hosts.
- Los mensajes ARP deberían ser transparentes para el usuario, así que se agregarán por defecto, mientras que los mensajes ICMP podrán ser modificados de acuerdo a la conveniencia del usuario en un archivo externo al código del componente.

Entonces partiendo de estas premisas se diseñará el componente. El archivo pytutorial.py incluido bajo el directorio ~/noxcore/src/nox/ coreapps/examples/tutorial es un buen punto de partida para programar el componente. De este archivo solo se toman las estructuras básicas para tener una referencia de como programar el componente. El código del componente se muestra en Código 2.2.

```
#Componente NOX para el paso de rutas a los conmutadores


#Se importan las librerías necesarias para manejar mensajes OpenFlow, paquetes Ethernet y para leer constantes de un archivo externo

from nox.lib.core import *

import nox.lib.openflow as openflow

from nox.lib.packet.ethernet import ethernet

import ConfigParser


#Se define la clase pytutorial como componente de NOX

class pytutorial(Component):


#Se define el constructor de la clase y se lo inicializa, con los parámetros self (referencia al objeto actual) y ctxt (información de contexto)

    def __init__(self, ctxt):

        Component.__init__(self, ctxt)


#Ahora se define el método por el cual se enviarán los flujos, con parámetros de entrada self(referencia al objeto actual), dpid(id del conmutador), inport (puerto de entrada), packet (paquete recibido), buf_id (identificador del buffer para recibir o escribir información)

    def envio_flujo(self, dpid, inport, packet, buf, bufid):

        """Envio de flujos hacia los conmutadores."""


#se crea un objeto config para leer los parámetros de los flujos ubicados en el archivo example.cfg. Igualmente se definen dos contadores que serán usados para leer los parámetros escritos en este archivo.

          config = ConfigParser.RawConfigParser()

          config.read('example.cfg')
```

Código 2.2 Componente NOX personalizado (continúa...)

```
        contadora = 0

        contadorb = 0


#código para el conmutado s3 cuyo identificador (dpid) es 3, valor que se
otorgó en el script para la creación de la topología

        if (dpid == 3 ):

        #los flujos se definen como permanentes

                idle_timeout = openflow.OFP_FLOW_PERMANENT

                hard_timeout = openflow.OFP_FLOW_PERMANENT

        #se crea un diccionario (hash) con los atributos de los flujos

                attrs = {}

        #se crea la regla para el flujo ARP entre h1 y h5: tipo de paquete
        ARP, puerto de entrada 1, puerto de salida 3.

                attrs[core.DL_TYPE] = ethernet.ARP_TYPE

                attrs[core.IN_PORT] = 1

                actions = [[openflow.OFPAT_OUTPUT, [0, 3]]]

                #se instala la regla para el flujo ARP

                self.install_datapath_flow(dpid,        attrs,        idle_timeout,
                hard_timeout, actions)

        #y se crea el flujo para el camino de retorno.

                attrs[core.DL_TYPE] = ethernet.ARP_TYPE

                attrs[core.IN_PORT] = 3

                actions = [[openflow.OFPAT_OUTPUT, [0, 1]]]

                self.install_datapath_flow(dpid,        attrs,        idle_timeout,
hard_timeout, actions)


                #ahora se lee el número de flujos del archivo example.cfg

                numflujoa = config.getint('Section1', 'numeroflujos')
```

Código 2.2 Componente NOX personalizado (continúa...)

```
#se agregan tantos flujos como se indicó en el archivo de configuración
    while contadora < numflujoa:

    #se lee la interfaz de origen de cada flujo
        aux = 'interfazorigen'+ str(contadora)
        into = config.getint('Section1', aux)
    #se lee la interfaz destino de cada flujo
        aux1 = 'interfazdestino'+ str(contadora)
        intd = config.getint('Section1', aux1)
    #se leen las direcciones IP origen y destino
        aux2 = 'iporigen' + str(contadora)
        ipo = config.get('Section1', aux2)
        aux3 = 'ipdestino' + str(contadora)
        ipd = config.get('Section1', aux3)
    #y el protocolo especificado en el campo IP
        aux4 = 'ipprotocolo' + str(contadora)
        ippr = config.getint('Section1', aux4)

    #se especifica identificador del tipo de protocolo de capa tres que se maneja en la cabecera de la capa dos, en este caso IP.
        attrs[core.DL_TYPE] = ethernet.IP_TYPE
    #se escriben los valores leídos en el diccionario
        attrs[core.IN_PORT] = into
        attrs[core.DL_TYPE] = ethernet.IP_TYPE
        attrs[core.NW_SRC] = ipo
```

Código 2.2 Componente NOX personalizado (continúa...)

```
                        attrs[core.NW_DST] = ipd
                        attrs[core.NW_PROTO] = ippr
                #y se envía al flujo por la interfaz de salida especificada
                        actions = [[openflow.OFPAT_OUTPUT, [0, intd]]]
                        self.install_datapath_flow(dpid, attrs, idle_timeout, hard_timeout, actions)
                #se incrementa el contador para repetir el lazo while mientras existan flujos
                        contadora = contadora + 1
#ahora se realiza el mismo proceso, pero con el conmutador s4, como no hay otro conmutador no es necesario comparar su identificación
        else:
                idle_timeout = openflow.OFP_FLOW_PERMANENT
                hard_timeout = openflow.OFP_FLOW_PERMANENT
                attrs = {}
                attrs[core.DL_TYPE] = ethernet.ARP_TYPE
                attrs[core.IN_PORT] = 2
                actions = [[openflow.OFPAT_OUTPUT, [0, 1]]]
                self.install_datapath_flow(dpid, attrs, idle_timeout, hard_timeout, actions)

                attrs[core.DL_TYPE] = ethernet.ARP_TYPE
                attrs[core.IN_PORT] = 1
                actions = [[openflow.OFPAT_OUTPUT, [0, 2]]]
                self.install_datapath_flow(dpid, attrs, idle_timeout, hard_timeout, actions)
                numflujob = config.getint('Section2', 'numeroflujos')
                while contadorb < numflujob:
```

Código 2.2 Componente NOX personalizado (continúa...)

```
                    aux = 'interfazorigen'+ str(contadorb)
                    into = config.getint('Section2', aux)
                    aux1 = 'interfazdestino'+ str(contadorb)
                    intd = config.getint('Section2', aux1)
                    aux2 = 'iporigen' + str(contadorb)
                    ipo = config.get('Section2', aux2)
                    aux3 = 'ipdestino' + str(contadorb)
                    ipd = config.get('Section2', aux3)
                    aux4 = 'ipprotocolo' + str(contadorb)
                    ippr = config.getint('Section2', aux4)
                    attrs[core.IN_PORT] = into
                    attrs[core.DL_TYPE] = ethernet.IP_TYPE
                    attrs[core.NW_SRC] = ipo
                    attrs[core.NW_DST] = ipd
                    actions = [[openflow.OFPAT_OUTPUT, [0, intd]]]
                    self.install_datapath_flow(dpid, attrs, idle_timeout,
hard_timeout, actions)
                    contadorb = contadorb + 1


#cuando llega un mensaje de un paquete que no tiene una acción a realizar, se llama al método envio_flujo para que envíe los correspondientes flujos
  def packet_in_callback(self, dpid, inport, reason, len, bufid, packet):
    self.envio_flujo(dpid, inport, packet, packet.arr, bufid  )
    return CONTINUE
#se define un método para llamar al método register_for_packet_in, especificado en el controlador NOX
  def install(self):
```

Código 2.2 Componente NOX personalizado (continúa...)

```
        self.register_for_packet_in(self.packet_in_callback)


# Método para adquirir del controlador la interfaz en la cual éste escucha peticiones
    def getInterface(self):
        return str(pytutorial)


# Método para adquirir del controlador la casa fabricante, es necesario para cualquier componente NOX
def getFactory():
    class Factory:
        def instance(self, ctxt):
            return pytutorial(ctxt)
    return Factory()
```

Código 2.2 Componente NOX personalizado

Ahora se debe crear el archivo example.cfg bajo el directorio ~/noxcore/build/src, el cual contendrá las especificaciones de las reglas de flujo que el usuario desee agregar a cada uno de los conmutadores.

En el Archivo de Configuración 2.1 se presentan los parámetros usados en esta implementación.

```
[Section1]
#Total de flujos para s3, se debe especificar un número entero
numeroflujos = 2
#cada flujo tiene un identificador al final de cada parámetro, por ejemplo al final del primer flujo se ubica el identificador 0, en el segundo flujo 1 y así sucesivamente
#interfazorigen, interfazdestino e ipprotocolo deben ser enteros, iporigen e ipdestino deben ser direcciones IP en formato decimal
```

Archivo de Configuración 2.1 (continúa...)

```
#flujos para los mensajes ICMP
interfazorigen0 = 1
interfazdestino0 = 3
iporigen0 = 10.0.0.1
ipdestino0 = 10.0.0.2
#campo protocolo de la cabecera IP, en este caso 1=ICMP
ipprotocolo0 = 1

interfazorigen1 = 3
interfazdestino1 = 1
iporigen1 = 10.0.0.2
ipdestino1 = 10.0.0.1
ipprotocolo1 = 1

[Section2]
#flujos para el conmutador s4
numeroflujos =2
interfazorigen0 = 2
interfazdestino0 = 1
iporigen0 = 10.0.0.2
ipdestino0 = 10.0.0.1
ipprotocolo0 = 1

interfazorigen1 = 1
interfazdestino1 = 2
iporigen1 = 10.0.0.1
ipdestino1 = 10.0.0.2
ipprotocolo1 = 1
```

Archivo de Configuración 2.1

Ahora se procede en primer lugar a detener la ejecución del controlador NOX que se ejecuta por defecto en Mininet con el comando

```
$sudo killall controller
```

Posteriormente se ejecuta el controlador NOX con el componente programado desde un terminal, mediante el siguiente comando desde el directorio ~/noxcore/build/src

```
$./nox_core –v –I ptcp: pytutorial
```

Y desde otro terminal se ejecuta el *script* para la creación de la topología ahora con el parámetro **controller remote.**

```
$sudo mn --custom ~/mininet/custom/topologia.py --topo mytopo --controller remote
```

El controlador inmediatamente enviará los flujos necesarios a ambos conmutadores. Ahora se realizará una prueba de *ping* entre todos los hosts para verificar que solo hay conectividad entre h1 y h5, cuyo resultado se muestra en la Figura 2.18. En este caso se usa el comando *pingall* ejecutado desde la línea de comandos de Mininet.

Figura 2.18 Prueba *pingall*

Pingall es un comando que permite verificar la conectividad entre todos los hosts que se agregaron a la simulación. En la Figura 2.18 se puede observar que el host denominado h1 intenta enviar mensajes ICMP hacia los otros tres hosts. Las X indican que los mensajes han sido fallidos, mientras que h5 indica que los mensajes hacia este host fueron exitosos. Lo mismo se realiza con los otros tres hosts. En resumen los mensajes exitosos son transmitidos de h1 a h5 y de h5 a h1, es decir que hay dos mensajes exitosos y el resto, en este caso 10 mensajes, son fallidos.

Mediante la programación de este controlador ha quedado en evidencia la gran facilidad que ofrecen las SDN para que el administrador las adapte a sus necesidades. Como ejemplos de aplicación se podría permitir el tráfico de determinadas aplicaciones entre hosts mientras se bloquea otro tipo de tráfico no deseado. Haciendo una analogía con una red tradicional, se podría hablar que un conmutador OpenFlow puede realizar las funciones de un conmutador capa dos,

conmutador capa tres, enrutador y *firewall* en un solo dispositivo. En el siguiente capítulo se programarán tres controladores más.

Lamentablemente en la simulación solo se pudo usar tráfico ICMP; ya que, a pesar de que Mininet puede manejar otro tipo de tráfico, este requiere de programación adicional fuera del alcance del presente Proyecto de Titulación. Sin embargo, este tráfico es suficiente para llegar a comprender cómo actúan las reglas enviadas al controlador y para verificar el funcionamiento de las mismas. En el siguiente capítulo se agregarán reglas de flujos más exactas tomando en cuenta los números de puerto correspondientes a la capa 4 del modelo OSI.

REFERENCIAS:

[1] DREIER, Gustavo, Tutorial Mininet, Documento Electrónico http://www.cedia.org.ec/images/stories/documentos_descarga/CursoOpenFlowDeRedClara/ 06a-mininet.pdf (Consultado el 12 de Marzo de 2013)

[2] ANÓNIMO, Tutorial, Documento Electrónico http://www.cs.princeton.edu/courses/archive/fall10/cos561/assignments/ps2-tut.pdf (Consultado el 12 de Marzo de 2013)

[3] VirtualBox. https://www.virtualbox.org/ (Consultado el 15 de Marzo de 2013)

[4] Xming. http://www.straightrunning.com/XmingNotes/ (Consultado el 15 de Marzo de 2013)

[5] ANÓNIMO, Manual de referencia Red Hat Linux 4, Capítulo 20: Protocolo SSH, Documento Electrónico http://www.gb.nrao.edu/pubcomputing/redhatELWS4/RH-DOCS/rhel-rg-es-4/ch-ssh.html (Consultado el 15 de Marzo de 2013)

[6] PuTTY. http://www.putty.org/ (Consultado el 15 de Marzo de 2013)

[7] OpenFlow. http://www.openflow.org/wp/openflow-components/ (Consultado el 25 de Marzo de 2013)

[8] ANÓNIMO, HP Networking OpenFlow Workshop, Documento Electrónico http://www.cedia.org.ec/images/stories/documentos_descarga/CursoOpenFlowDeRedClara/05e-dpctl.pdf (Consultado el 2 de Abril de 2013)

[9] Tipo de topología. http://www.gobiernodecanarias.org/educacion/Conocernos_mejor/paginas/topologia.html (Consultado el 10 de Abril de 2013)

[10] Manual del comando dpctl en Linux.

[11] Mininet. http://mininet.org (Consultado el 2 de Abril de 2013)

CAPÍTULO III

3. IMPLEMENTACIÓN DEL PROTOTIPO

En el capítulo anterior se realizó la simulación en Mininet de la SDN propuesta para el presente Proyecto de Titulación. En este capítulo se implementará una SDN similar pero a nivel de prototipo, con dispositivos reales, usando cuatro controladores diferentes: NOX, POX, Beacon y Floodlight. En primer lugar se usará un componente incluido en el controlador para brindar funcionalidades de conmutador capa dos del modelo OSI a los conmutadores OpenFlow con los que se va a trabajar. Posteriormente se programarán componentes personalizados para enviar flujos determinados a cada conmutador y, finalmente, se realizarán pruebas con diferentes tipos de tráfico. Adicionalmente, en este capítulo se incluirá un presupuesto referencial del prototipo y en el subcapítulo de pruebas se presentarán las capturas de Wireshark con el comando *dpctl* que quedaron pendientes en el capítulo anterior.

En primer lugar se presentará la topología usada para el proyecto. Esta SDN es similar a la misma usada en la simulación realizada en el capítulo anterior, es decir dos conmutadores interconectados entre sí, pero a diferencia de la simulación solo se tendrá un host conectado a cada conmutador, pero con tres máquinas virtuales instaladas en cada uno.

3.1 TOPOLOGÍA

En la Figura 3.1 se observa la topología que se utilizará para la implementación de la red SDN del presente proyecto.

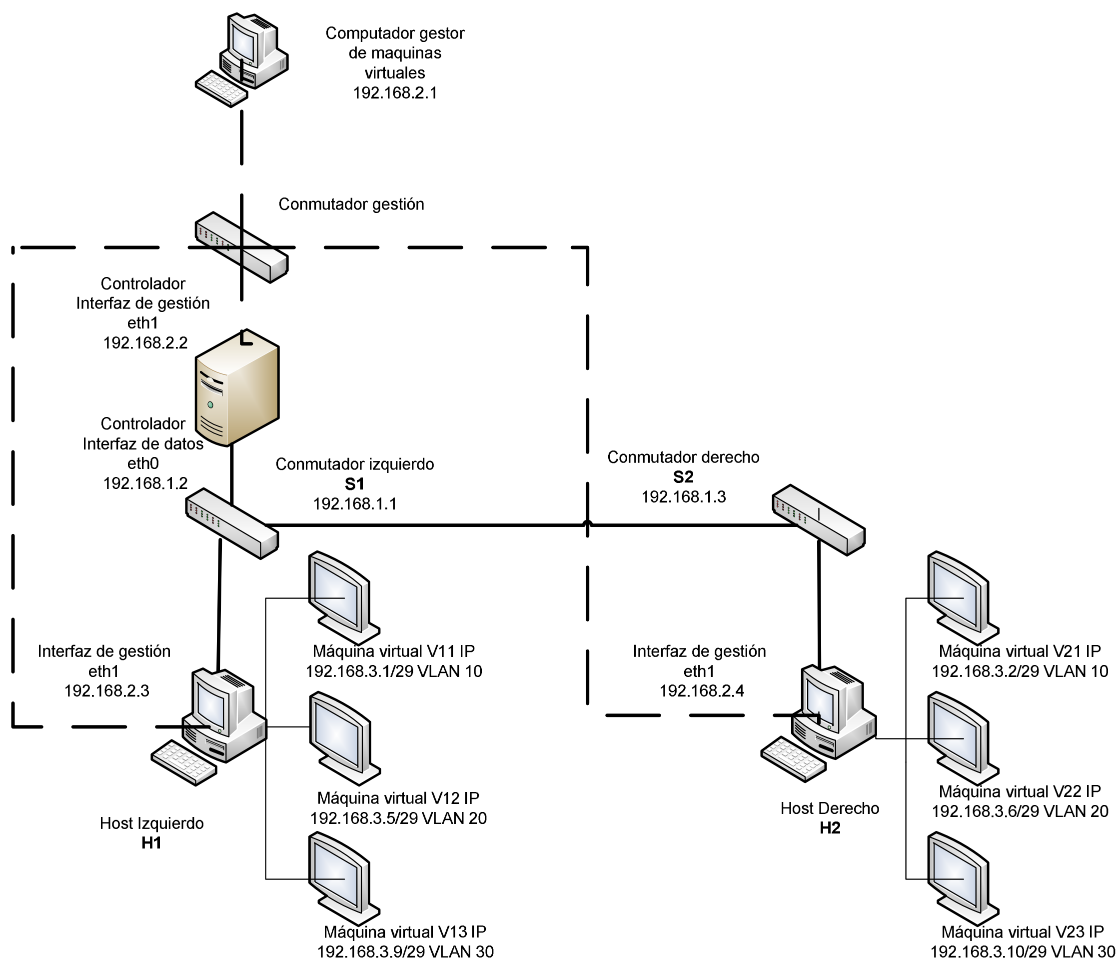


Figura 3.1 Topología usada para la implementación de la SDN

Esta topología se ha seleccionado por los mismos motivos y para cumplir los mismos objetivos descritos en el capítulo dos, que pueden resumirse en buscar una similitud con una red real.

En el diagrama de topología de la Figura 3.1 se observan los siguientes dispositivos:

- Host H1, que posee tres máquinas virtuales V11, V12 y V13, cada una ubicada en la VLAN 10, 20 y 30 respectivamente, todas ellas conectadas al conmutador S1.
- Conmutador izquierdo S1 habilitado para el manejo de OpenFlow.

- Conmutador derecho S2 habilitado para el manejo de OpenFlow.
- Host H2, que posee tres máquinas virtuales V21, V22 y V23, cada una ubicada en la VLAN 10, 20 y 30 respectivamente, todas ellas conectadas al conmutador S2.

En la Tabla 3.1 se puede encontrar la asignación de direcciones entre las VLAN 10, 20 y 30.

ASIGNACIÓN DE DIRECCIONES DE RED

VLAN	Número de Hosts	Dirección Red	Máscara
10	2	192.168.3.0	255.255.255.252
20	2	192.168.3.4	255.255.255.252
30	2	192.168.3.8	255.255.255.252

Tabla 3.1 Asignación de direcciones de red

Al igual que en la simulación del capítulo dos, se implementará sólo la conectividad entre hosts de una misma subred. Los hosts de diferentes subredes no podrán comunicarse, debido a que como ya se mencionó, no se han definido rutas qué correspondan a un proceso de enrutamiento capa tres del modelo OSI.

3.1.1 PLATAFORMA DE HARDWARE

3.1.1.1 Servidor Controlador

El servidor controlador se implementa en un computador Ultratech que cuenta con las siguientes características:

- Procesador Intel Core i7-3770 CPU @3.4GHz
- Memoria RAM de 8GB.
- Disco duro de 500 GB.

- Dos adaptadores de red Realtek PCIe GBE Family Controller, uno para administración y otro para el envío de datos.
- Sistema Operativo VMWare ESXi 5.1.0.
- Tecnología Intel Virtualization Technology (virtualización x86), que permite que la máquina física actúe como si se tuviese varios computadores independientes, para ejecutar varios sistemas operativos al mismo tiempo en la misma máquina física, compartiendo recursos como el procesador o la memoria de la máquina física.

El servidor controlador tiene el sistema operativo VMware vSphere Hypervisor (ESXi) 5.1 que permite tener varias máquinas virtuales en una máquina física compartiendo recursos tales como memoria y procesamiento entre ellas [1].

El servidor controlador con el sistema operativo ESXi permite configuraciones básicas como gestión de direcciones IP, pero para la gestión de las máquinas virtuales se necesita otro computador, conectado a la misma red de gestión del servidor controlador y que tenga instalado el software vSphere client. Este programa será el que se conecte con el servidor controlador, para la gestión de las máquinas virtuales que se encuentran ubicadas en él [1].

Cabe recalcar que este ordenador solo se utiliza para la gestión de las máquinas virtuales mediante una interfaz visual. En él no se encuentra instalada ninguna máquina virtual. Las máquinas virtuales se ejecutan independientemente en el servidor controlador. Sobre este computador se crearán cuatro máquinas virtuales para el manejo de cuatro servidores controladores diferentes que se ejecutarán de manera independiente, los cuales se indican en las siguientes secciones.

3.1.1.2 Conmutadores

Para la presente implementación se ha decidido usar el Access Point Linksys WRT54GL, del fabricante Cisco, como conmutador OpenFlow. El equipo posee las siguientes características [2]:

- Un puerto Internet RJ-45 10/100 Mbps.
- Cuatro puertos Ethernet RJ-45 10/100 Mbps.
- Todos los puertos soportan intercambio de pines por software MDI/MDI-X.
- CPU Broadcom BCM5352 @200MHz.
- Memoria RAM de 16 MB.
- Memoria Flash de 4MB.
- Soporta actualización de *firmware* OpenWRT.
- WLAN: soporta los protocolos IEEE 802.11b y g.

Para que el dispositivo soporte el protocolo OpenFlow, es necesario realizar una actualización de su *firmware*. El *firmware* a ser instalado es una versión de OpenWRT capaz de manejar el protocolo OpenFlow.

OpenWRT es una distribución de Linux para dispositivos embebidos [3]. El código fuente puede modificarse para personalizar el dispositivo y así adaptarse a las necesidades del usuario, que en este caso es la capacidad de interpretar el protocolo OpenFlow.

Se necesita una distribución de OpenWRT que pueda interpretar OpenFlow [4], la misma que puede ser descargada de la página web http://www.openflow.org/wk/index.php/OpenFlow_1.0_for_OpenWRT. Es necesaria una imagen con extensión .trx que luego se carga desde la interfaz de administración web que viene incluida en el equipo Cisco.

Una vez que se ha actualizado el *firmware*, se reinicia el equipo. Esta versión de OpenWRT no incluye una interfaz de administración web, por lo que es necesario acceder al dispositivo mediante telnet. El conmutador tiene como dirección por defecto 192.168.1.1 y se accede a él a través del servidor controlador que tiene una dirección IP 192.168.1.2 y se conecta directamente al puerto 1 del conmutador. En la Figura 3.2 se muestra el acceso mediante telnet al dispositivo.

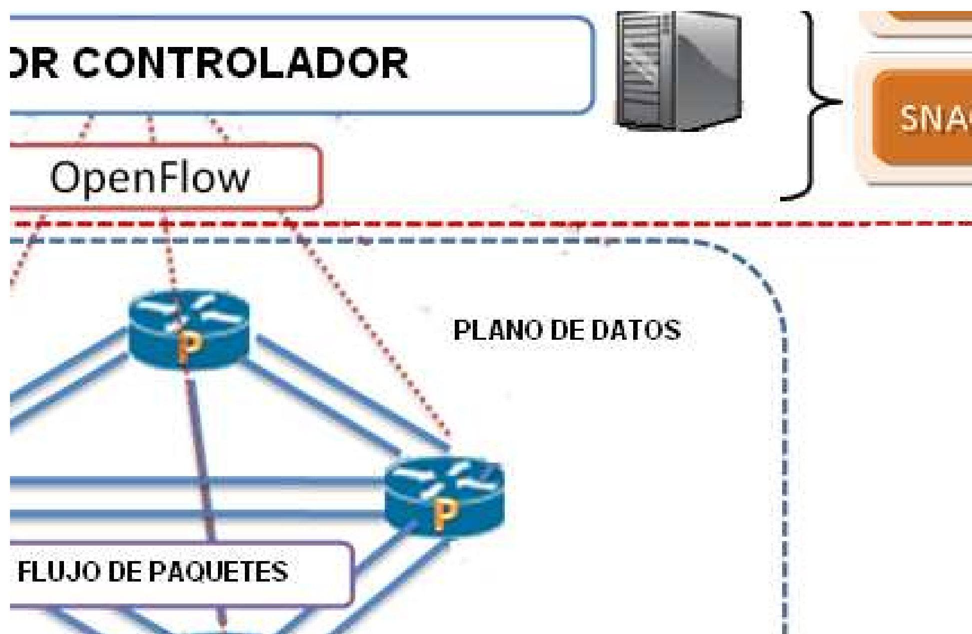


Figura 3.2 Acceso mediante telnet al conmutador

Lo primero que se hace es configurar el dispositivo. Para ello son fundamentales dos archivos de configuración ubicados en el directorio /etc/config. El primer archivo es "*network*", en donde se administrará la dirección IP del dispositivo, las VLAN asociadas a cada puerto y otras configuraciones de red. El siguiente archivo que se debe configurar es el denominado "*openflow*", en donde se introduce la dirección IP y el número de puerto del controlador al que el conmutador enviará peticiones. Para mayor información acerca de la configuración recomendada, referirse al Anexo B, en donde se presenta un ejemplo de los dos archivos de configuración.

Cabe recalcar que éste es un equipo habilitado para el manejo de OpenFlow, no es concebido en un inicio para que soporte OpenFlow por lo que se encontrarán algunas diferencias con los equipos diseñados para este protocolo. Una de ellas, y la más importante, es que los equipos Cisco que se emplean en el proyecto no necesitan una VLAN definida para conectarse con los controladores, a diferencia de, por ejemplo, conmutadores HP diseñados para soportar OpenFlow. Por ello solo es necesario especificar la dirección y puerto del controlador y conectarse con él en el puerto físico identificado con el número uno.

3.1.1.3 Hosts

En cuanto a los hosts, se ha decidido utilizar dos computadores con las mismas características que el controlador para poder utilizar máquinas virtuales en cada uno de ellos para tener así tres hosts virtuales en cada host físico.

Cada una de estas máquinas físicas contiene tres máquinas virtuales, como se indica en la topología de la Figura 3.1. Las máquinas virtuales tienen los siguientes sistemas operativos:

- V11 y V21 tienen instalado el sistema operativo Ubuntu 12.04 LTS[8]. V11 tiene instalado Smokeping[9] y un servidor web.
- V12 tiene instalado el sistema operativo Ubuntu 12.04 LTS. V21 tiene instalado el sistema operativo CentOS 6. V21 contiene un servidor telnet, mientras que V12 contiene un cliente telnet.
- V13 y V23 tienen el sistema operativo Windows 7. Ambos hosts tienen instalado el software VLC Media Player. V23 hará las veces de servidor de *streaming* de video, mientras que V13 hará las veces de cliente.

3.2 SOFTWARE DEL SERVIDOR CONTROLADOR

Para la implementación del presente Proyecto de Titulación, se ha decidido usar cuatro controladores diferentes: NOX, POX, Beacon y FloodLight.

En primer lugar se detallará la implementación de cada controlador usando el componente para convertir los dispositivos en conmutadores capa dos que se incluye en el controlador y, posteriormente; se indicará el código del componente personalizado creado para transmitir a los conmutadores flujos específicos.

3.2.1 NOX

[8] LTS: *Long Term Support,* versiones del sistema operativo que tienen un soporte de cinco años.
[9] Smokeping: software para medir retardos en la red.

El primer controlador que se implementará es NOX. Para ello como requisito previo se debe tener un sistema operativo Ubuntu. Para esta implementación en particular se ha usado una máquina virtual Ubuntu 12.04 LTS que posee los siguientes recursos:

- Disco duro virtual de 60 GB.
- Memoria de 3.8 GB.
- Adaptador de red VMNetwork.

En el Anexo C se muestran los pasos detallados que se requieren para la instalación de un servidor controlador NOX. Una vez instalado NOX, se procede a la ejecución del mismo. Para ello el software incluye una ayuda para entender la sintaxis con la cual puede ser ejecutado un comando. Para desplegar la ayuda se ejecuta el siguiente comando [5], [6], [7]:

```
$sudo ./nox_core –h
```

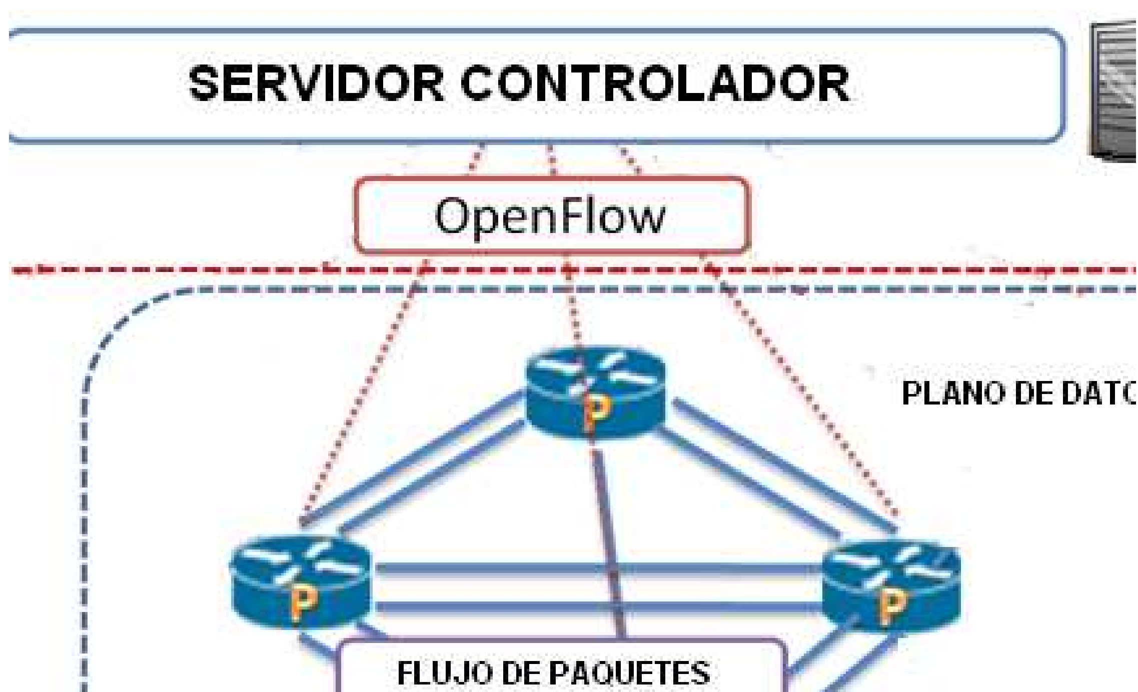


Figura 3.3 Ayuda de NOX

En la Figura 3.3 se muestra la ayuda proporcionada para la ejecución del *script* ./nox_core -h. En ella se puede observar la sintaxis básica con la cual se da inicio

a la ejecución del controlador. Una vez que este se ejecuta, el controlador se queda en un estado de espera hasta recibir mensajes de un dispositivo de red. La sintaxis básica que se usa para iniciar el controlador se detalla a continuación [5], [6], [7]:

```
$sudo ./nox_core –i <protocolo>:<dirección ip>:<puerto> <componentes adicionales>
```

En donde:

- **<protocolo>:** indica con qué protocolo se realizará la comunicación, y puede ser ptcp para el protocolo TCP o si se desea mayor seguridad se puede usar pssl para el protocolo SSL. El protocolo TCP se ejecuta en la capa transporte y el protocolo SSL se ejecuta en la capa presentación, usando TCP para su transporte.
- **<dirección ip>**:**<puerto>:** es la dirección IP y el puerto en la que el controlador escucha las peticiones de los dispositivos de red.
- **<componentes adicionales>:** Se los usa para dar ciertas funcionalidades a los dispositivos. Por ejemplo, existe el componente *routing* que permite agregar reglas de flujos de manera dinámica para dar las funcionalidades de un enrutador al dispositivo o *switch*, que en este caso será empleado, para que el dispositivo se comporte como un conmutador normal.

Concretamente, el comando que se usará, teniendo en cuenta que la dirección IP del controlador es 192.168.1.2, que se usará una comunicación mediante TCP y con un número de puerto 6633; y, que se usará el componente *switch*, es el siguiente:

```
$sudo ./nox_core –i ptcp:192.168.1.2:6633 switch
```

Este comando se ejecutará, en este caso, desde el directorio /home/openflow/nox/build/src, que es el directorio en donde se escogió realizar la

instalación de NOX, o se podría agregar el *path* al comando y *ejecutar* el *script* desde cualquier lugar.

Adicionalmente se puede agregar la opción de tener información acerca de la ejecución más detallada adicionando la opción –v al ejecutar el *script* anterior. Esto tiene utilidad en caso de que existiera algún problema de comunicación entre el controlador y los dispositivos de comunicación o para verificar el tráfico que existe entre ambos. En la Figura 3.4 se muestra la ejecución del controlador NOX.

Figura 3.4 Ejecución de NOX

Sin embargo NOX presenta un gran problema: el *firmware* de los conmutadores que se ha escogido para la implementación del prototipo no es compatible con el software del controlador. Los síntomas de este problema son: en primer lugar la conexión entre un conmutador y el controlador es posible solo en instantes de tiempo, la mayoría del tiempo la conexión no es posible. Posteriormente, se agregó un segundo conmutador y la conexión en este caso es prácticamente imposible, ya que si con un conmutador la conexión era inestable, mucho peor con dos conmutadores.

Posteriormente, se verificó si existía un problema similar en Internet [14] [15] y se encontraron varias páginas que indicaban que la desconexión podría ser causada por un *dpid* mal configurado o quizá por un archivo de configuración erróneo, que podía ser la configuración del servidor DHCP o de OpenFlow. Se siguieron todas las indicaciones para la solución de este problema, pero sin embargo al final el problema de desconexión persistía, por lo que se concluyó que la conexión entre los conmutadores y el controlador NOX, con el *firmware* usado actualmente en el equipo Linksys WRT54GL, no es posible. Esto no es limitante, ya que el mismo conmutador con un *firmware* mejorado, no disponible al momento de la

implementación, podría solucionar este problema o también se podrían usar otros dispositivos en una implementación real.

Sin embargo, esto no es restricción para el uso del controlador NOX, ya que en la etapa de simulación se usó la herramienta Mininet con el controlador NOX, para definir reglas de flujos personalizables, lo que es, en esencia, lo mismo que se debería realizar con un controlador físico. Entonces, las pruebas del capítulo anterior se pueden comparar a las pruebas realizadas con los siguientes controladores para encontrar similitudes y diferencias ya que vienen a ser prácticamente lo mismo, lo único que cambia es el ambiente lógico o físico del controlador.

3.2.2 POX

El segundo controlador que se implementará es POX. Para ello cómo requisito previo se debe tener un sistema operativo Linux, Windows o MacOS; en este caso, se eligió Ubuntu por la facilidad de instalación y por ser un sistema operativo de libre distribución y de código abierto. Para esta implementación en particular, se ha usado una máquina virtual Ubuntu 12.04 LTS que posee los siguientes recursos:

- Disco duro virtual de 60 GB.
- Memoria de 3.8 GB.
- Adaptador de red VMNetwork.

En el Anexo D se incluye una guía detallada de como instalar el software del servidor controlador POX. Una vez instalado POX, se ejecuta el *script* que dará inicio a la ejecución de este controlador. La sintaxis básica que se usa se detalla a continuación [8]:

```
$sudo ./pox.py --<opciones> <versión OpenFlow> --<dirección> --<puerto> --<componentes adicionales>
```

En donde:

- **<opciones>:** es un campo que permite indicar al controlador que se desea realizar acciones adicionales; como por ejemplo, se incluiría "verbose" en este campo para tener información adicional de la ejecución del controlador. Otra opción es incluir "no-openflow" en este campo para no iniciar el módulo OpenFlow automáticamente y por tanto, no escuchar conexiones OpenFlow.
- **<versión OpenFlow>:** es la versión de OpenFlow con la que se realizará la comunicación, normalmente la versión es la número uno identificada en este campo como openflow.of_01.
- **<dirección> <puerto>:** es la dirección IP y el puerto en el que el controlador escucha las peticiones de los dispositivos de red.
- **<componentes adicionales>:** Son usados para dar ciertas funcionalidades a los dispositivos. Por ejemplo existe el componente "forwarding.l2_learning" para tener las funcionalidades de un conmutador capa dos, que aprende direcciones MAC; el componente "forwarding.hub", que hace al dispositivo de red comportarse como un concentrador; el componente "forwarding.l3_learning", que permite implementar las funcionalidades de conmutador capa tres en los dispositivos de red o componentes adicionales que pueden ser programados por los administradores para adaptarse a sus necesidades.

Concretamente, el comando que se usará, teniendo en cuenta que la dirección IP del controlador es 192.168.1.2, el número de puerto es 6633 y que se usará el componente "forwarding.l2_learning", es el siguiente [8]:

```
$sudo ./pox.py openflow.of_01 --addres:192.168.1.2 --port:6633 forwarding.l2_learning
```

Una vez ejecutado el *script*, el servidor comienza a escuchar las peticiones de los clientes. En la Figura 3.5 se observa la ejecución de POX.

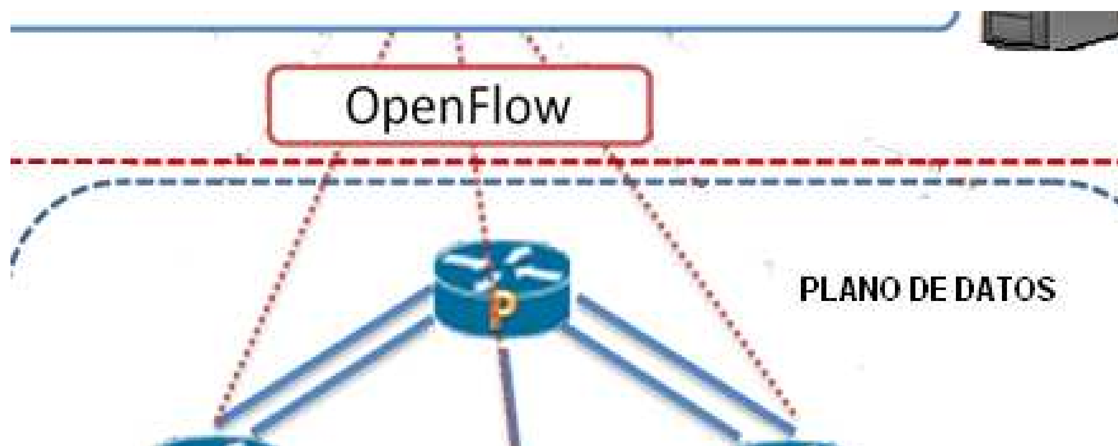


Figura 3.5 Ejecución de POX

En el ejemplo anterior se puede incluir la opción --verbose para tener información de ejecución adicional y así saber con certeza cuándo se establece la conexión.

Ahora se presentará el código de un componente que se ha programado para enviar ciertas reglas de flujos a los conmutadores mediante mensajes de modificación de flujos. En el código se enviarán directamente reglas de flujos para manejar el tráfico ARP entre todos los dispositivos y se leerá un archivo de configuración externo al código para instalar las reglas para otro tipo de flujos, como por ejemplo tráfico HTTP o Telnet, que el administrador considere necesarias. Este archivo de configuración se presentará en la sección de pruebas conjuntamente con las pruebas realizadas sobre las reglas de flujos definidas en este archivo.

Para la programación de este módulo se tomó en cuenta el archivo of_tutorial.py ubicado en el directorio pox/misc [12]. De este archivo se tomaron como referencia las clases básicas a partir de las cuales se definieron nuevas funcionalidades y acciones. En el Código 3.1 se presenta el código empleado para este componente.

```
#Se importan las librerías necesarias para el manejo de OpenFlow, para
leer archivos de configuración y obtener parámetros de configuración y
para el manejo de paquetes, como por ejemplo ARP
from pox.core import core
import pox.openflow.libopenflow_01 as of
import ConfigParser
import pox.lib.packet as pkt

#Se crea una instancia del método core.getLogger(), que permite registrar
información de log o bitácora
log = core.getLogger()

#Se define la clase Tutorial
class Tutorial (object):

#se pasa un objeto connection para cada conmutador al constructor de la
clase. Self indica que el constructor se refiere al objeto en uso actualmente
  def __init__ (self, connection):

# Se mantiene la conexión con el conmutador para poder enviarle
mensajes
    self.connection = connection
# Se añaden los conmutadores respectivos cuando se crea la conexión
    connection.addListeners(self)
#se define el  método para el envío de reglas de flujos
  def act_like_switch (self, packet, packet_in):

#se crean las reglas de flujos ARP entre todos los hosts
    msg = of.ofp_flow_mod()
#se añade la prioridad por defecto, que en este caso es 42
    msg.priority = 42
#dl_type indica qué clase de mensaje Ethernet (Ethertype) se está filtrando,
por ejemplo 0x0806 filtra mensajes ARP o 0x0800 filtra mensajes IPv4
    msg.match.dl_type = 0x0806
#si entra un mensaje ARP por el Puerto 2, se lo reenvía al Puerto 3
    msg.match.in_port = 2
    msg.idle_timeout = 60
    msg.actions.append(of.ofp_action_output(port =3))
    self.connection.send(msg)

#el mismo mensaje ARP pero en dirección contraria
    msg1 = of.ofp_flow_mod()
    msg1.priority = 42
    msg1.idle_timeout = 60
    msg1.match.dl_type = 0x0806
```

Código 3.1 Componente POX personalizado (continúa...)

```
    msg1.match.in_port = 3
    msg1.actions.append(of.ofp_action_output(port =2))
    self.connection.send(msg1)

#se crea una variable contador para leer los flujos del archivo de
configuración
    contador = 0

#se lee el archivo de configuración
    config = ConfigParser.RawConfigParser()
    config.read('example.cfg')

#se lee el número de reglas de flujos del archivo de configuración
    numflujo = config.getint('Section1', 'numeroflujos')

#mientras haya flujos en el archivo de configuración, añadir los flujos. Los
parámetros como interfaz de orígen (into), interfaz de destino (intd),
dirección IP de origen y de destino (ipo e ipd) se pasan a variables y luego
se cargan a la estructura de comparación de la regla para el flujo.
    while contador < numflujo:
           aux = 'interfazorigen' + str(contador)
           into = config.getint('Section1', aux)
            aux1 = 'interfazdestino' + str(contador)
           intd = config.getint('Section1', aux1)
            aux2 = 'iporigen' + str(contador)
           ipo = config.get('Section1', aux2)
            aux3 = 'ipdestino' + str(contador)
           ipd = config.get('Section1', aux3)
           msg2 = of.ofp_flow_mod()
#Se indica una prioridad por defecto, 0x800 indica que solo se tratarán
mensajes del tipo IP
           msg2.priority = 42
           msg2.match.dl_type = 0x800
          msg2.match.nw_src = ipo
          msg2.match.nw_dst = ipd
         msg2.idle_timeout = 60
            aux4 = 'ipprotocolo' + str(contador)
            ippr = config.get('Section1', aux4)
            msg2.match.nw_proto = int(ippr)
            aux5 = 'puertoorigen' + str(contador)
            puertoo = config.get('Section1', aux5)

#ahora se leerán los puertos capa 4, salvo que se especifique que no se
tomaran en cuenta mediante la palabra None.
```

Código 3.1 Componente POX personalizado (continúa...)

```
          if str(puertoo) != 'None':
                   msg2.match.tp_src = int(puertoo)
          aux6 = 'puertodestino' + str(contador)
          puertod = config.get('Section1', aux6)
          if str(puertod) != 'None':
                   msg2.match.tp_dst = int(puertod)
#del archivo de configuración se obtiene la interfaz de salida para esta
regla y posteriormente se envía la regla
          msg2.actions.append(of.ofp_action_output(port =intd))
          self.connection.send(msg2)
          contador = contador + 1
#Fin de while y act_like_swich

#Maneja mensajes PacketIn, llamando al método act_like_switch
definido arriba, con las varibles packet u packet_in como parámetros
de entrada
  def _handle_PacketIn (self, event):
          self.act_like_switch(packet, packet_in)

#Inicia el componente, definiendo un lanzador con funciones internas.
El parámetro event es una variable global que se usa dentro de la
función start_switch
def launch ():
  def start_switch (event):
    log.debug("Controlando el switch %s" % (event.connection))
    Tutorial(event.connection)
  core.openflow.addListenerByName("ConnectionUp", start_switch)
```

Código 3.1 Componente POX personalizado

3.2.3 BEACON

El tercer controlador a ser implementado es Beacon, cuya instalación se explica detenidamente en el Anexo E. Para esta implementación en particular se ha usado una máquina virtual con un sistema operativo Ubuntu 12.04 LTS que posee los siguientes recursos, idénticos a los usados en los controladores previos:

- Disco duro virtual de 60 GB.
- Memoria de 3.8 GB.
- Adaptador de red VMNetwork.

Una vez completada la instalación de Beacon, se procede a su ejecución, para ello se debe ejecutar el *script* beacon, como se indica a continuación [9]:

```
$sudo ./beacon
```

El controlador entonces escuchará por las peticiones que le envíen los dispositivos de red. En la Figura 3.6 se puede observar la ejecución del controlador.

Figura 3.6 Ejecución del controlador Beacon

Beacon cuenta además con una interfaz web, que se accede mediante el navegador en la página localhost:8080. En la Figura 3.7 se muestra la interfaz web, con un conmutador ya reconocido.

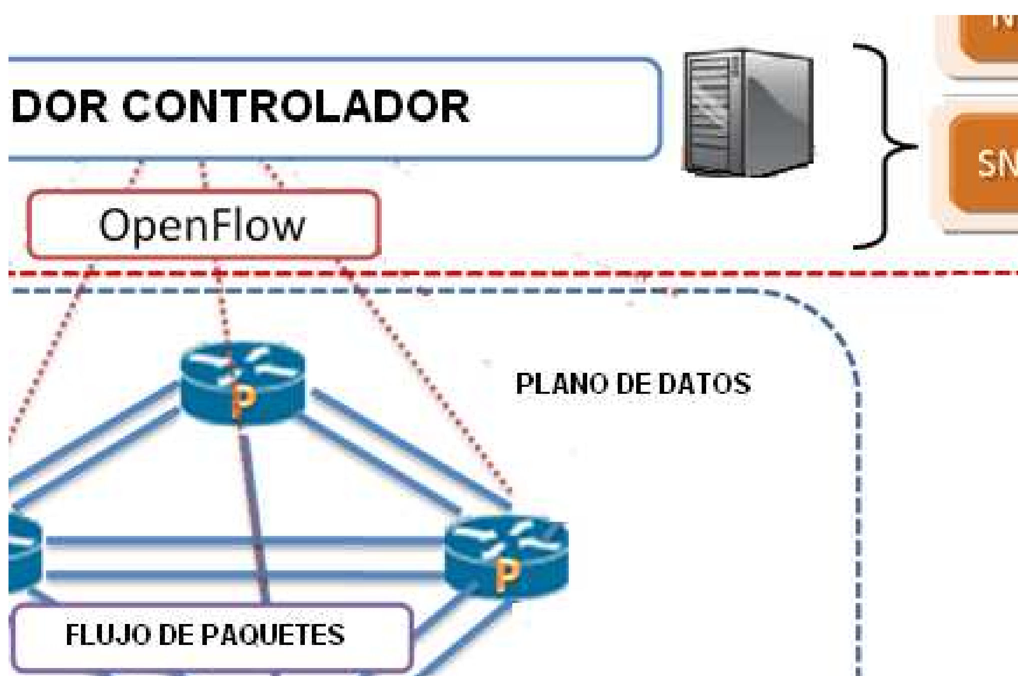


Figura 3.7 Interfaz web de Beacon

Para ejecutar Beacon, con el fin de que transmita a los conmutadores OpenFlow instrucciones de conmutación a nivel de capa dos, se agrega la opción de configuración “configurationSwitch” en el comando de ejecución como se indica a continuación:

```
$sudo ./beacon --configuration configurationSwitch
```

Inmediatamente, Beacon se ejecutará y en su interfaz web se podrá obtener información de los conmutadores que se han conectado con el controlador, como se observa en la Figura 3.8.

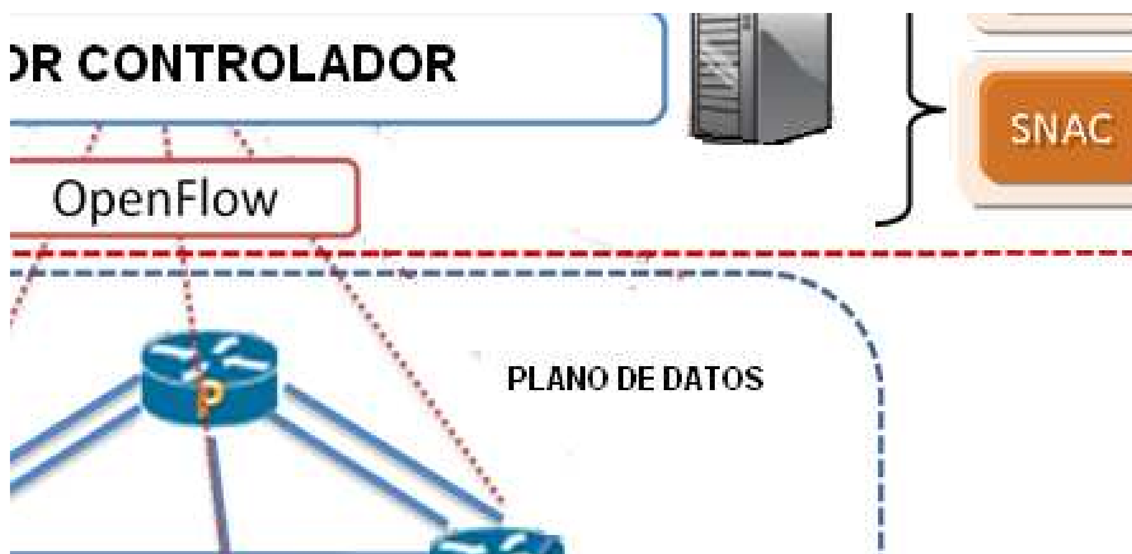


Figura 3.8 Adición de conmutadores en Beacon

Además se pueden observar los componentes que tiene el controlador, entre los cuales se puede diferenciar los componentes activos y no activos y se puede iniciar o detener la ejecución de cada uno de ellos; esto se indica en la Figura 3.9.

Finalmente, en la Figura 3.10 se puede observar que la interfaz web proporciona información acerca de los flujos que se han agregado de manera dinámica al controlador.

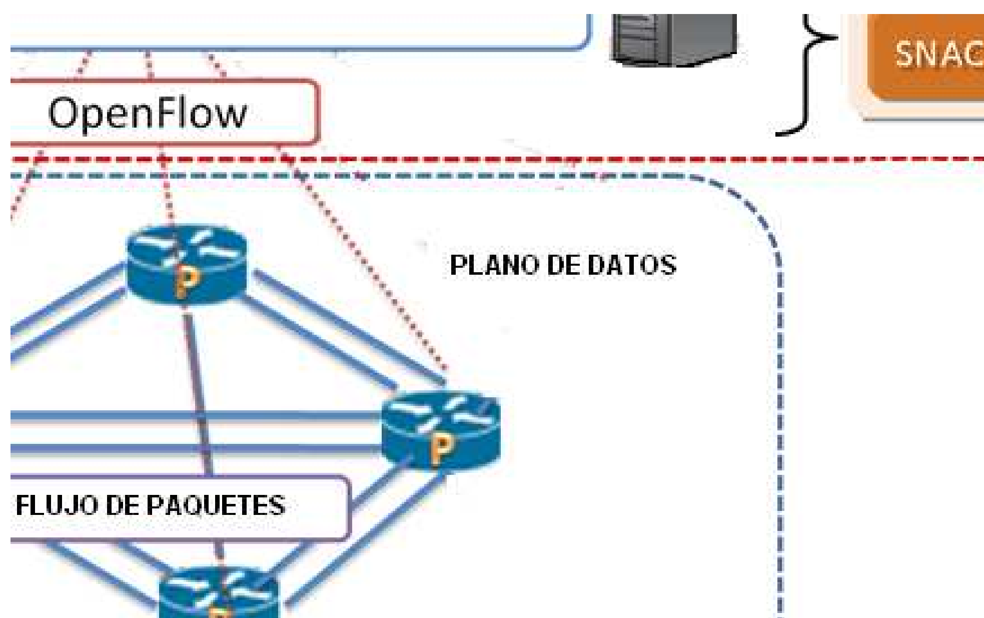


Figura 3.9 Componentes de Beacon

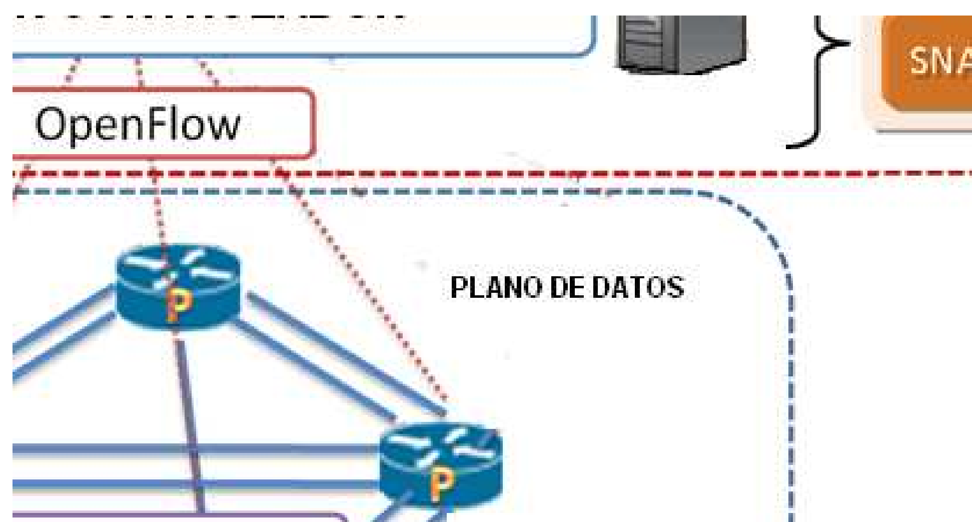


Figura 3.10 Detalle de los flujos en Beacon

Ahora, al igual que en el caso del controlador POX, se presentará el código de un componente que envíe reglas para ciertos flujos a los conmutadores. Este código realiza lo mismo que el que se presentó para POX, salvo que este se programó en Java y no en Python.

Al igual que el código anterior, en este caso se lee un archivo de configuración y se instalan los flujos contenidos en su interior. Cabe recalcar que se usó el archivo LearningSwitchTutorial.java, que se encuentra en el directorio

net.beaconcontroller.tutorial/src/main/java/net/ beaconcontroller/tutorial, como base para el desarrollo de este componente [12].

A partir de este archivo se modificaron las funcionalidades para adaptarlas a las necesidades de creación de flujos. El código usado se puede ver en el Código 3.2.

```
//se importan las librerías y paquetes necesarios
package net.beaconcontroller.tutorial;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;
import net.beaconcontroller.core.IBeaconProvider;
import net.beaconcontroller.core.IOFMessageListener;
import net.beaconcontroller.core.IOFSwitch;
import net.beaconcontroller.core.IOFSwitchListener;
import net.beaconcontroller.packet.Ethernet;
import net.beaconcontroller.packet.IPv4;
import org.openflow.protocol.OFFlowMod;
import org.openflow.protocol.OFMatch;
import org.openflow.protocol.OFMessage;
import org.openflow.protocol.OFPacketIn;
import org.openflow.protocol.OFPacketOut;
import org.openflow.protocol.OFPort;
import org.openflow.protocol.OFType;
import org.openflow.protocol.action.OFAction;
import org.openflow.protocol.action.OFActionOutput;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

//se crea la clase para el envío de reglas de flujo
public class LearningSwitchTutorial implements IOFMessageListener,
IOFSwitchListener {
//se agrega una instancia de información de logging
    protected static Logger log =
LoggerFactory.getLogger(LearningSwitchTutorial.class);
//se crean las variables para su uso con Beacon Provider
```

Código 3.2 Componente Beacon personalizado (continúa...)

```
        protected IBeaconProvider beaconProvider;
 //Cuando se recibe un paquete se extrae el mensaje del paquete y se
llama al método para la instalación de reglas
    public Command receive(IOFSwitch sw, OFMessage msg) throws
IOException {
           OFPacketIn pi = (OFPacketIn) msg; }

         forwardAsLearningSwitch(sw, pi);
       return Command.CONTINUE;
    }
//método para la instalación de reglas en las tablas de flujos
    public void forwardAsLearningSwitch(IOFSwitch sw, OFPacketIn pi)
throws IOException {
//Se crean las reglas para los flujos ARP
       OFFlowMod fm = new OFFlowMod();
       fm.setCommand(OFFlowMod.OFPFC_ADD);
 // Se define el tiempo de expiración
       fm.setIdleTimeout((short) 60);
      fm.setPriority((short) 42); //42 es la prioridad por defecto
       OFMatch match = OFMatch.load(pi.getPacketData(), pi.getInPort());
       match.setInputPort((short) 2);
//Al igual que en POX 0x8006 representa ARP
       match.setDataLayerType((short) 0x0806);
//en Beacon se deben usar Wildcards para elegir que coincidencias debe
tener el paquete con la regla para considerarlo parte del flujo
       match.setWildcards(1581102);
//se considera solo interfaz origen y destino
       short outPort = 3;
       fm.setMatch(match);
 // Se definen que acciones hacer con el flujo
       OFAction action = new OFActionOutput(outPort);
       fm.setActions(Collections.singletonList((OFAction)action));
//Se envía la regla al conmutador
       sw.getOutputStream().write(fm);
//lo mismo se realiza con el flujo ARP de regreso
       OFFlowMod fm1 = new OFFlowMod();
       fm1.setCommand(OFFlowMod.OFPFC_ADD);
       fm1.setIdleTimeout((short) 60);
       fm1.setPriority((short) 42);
       OFMatch match1 = OFMatch.load(pi.getPacketData(), pi.getInPort());
       match1.setInputPort((short) 3);
       match1.setDataLayerType((short) 0x0806);
       match1.setWildcards(1581102);
       short outPort1 = 2;
       fm1.setMatch(match1);
```

Código 3.2 Componente Beacon personalizado (continúa...)

```
        OFAction action1 = new OFActionOutput(outPort1);
        fm1.setActions(Collections.singletonList((OFAction)action1));
        sw.getOutputStream().write(fm1);

//Se leen las características de las reglas del archivo openflow.properties
        Properties props = new Properties();
        FileInputStream fis = new FileInputStream
("/home/openflow/Escritorio/openflow.properties");
       props.load(fis);
//se lee el número de flujos
        int numeroFlujos=Integer.parseInt(props.getProperty("openflow.
numeroFlujos"));
//al igual que POX se define un contador
        int contador=0;
//mientras haya reglas en el archivo de configuración se añaden las mismas
        while ( contador != numeroFlujos )
        {
            String aux="openflow.interfazEntrada"+ Integer.toString(contador);
            short intEntrada=Short.parseShort(props.getProperty(aux));
            String aux1="openflow.interfazSalida"+ Integer.toString(contador);
            short intSalida=Short.parseShort(props.getProperty(aux1));
            String aux2="openflow.ipOrigen"+ Integer.toString(contador);
            String ipOrigen=props.getProperty(aux2);
            String aux3="openflow.ipDestino"+ Integer.toString(contador);
            String ipDestino=props.getProperty(aux3);
            String aux4="openflow.tipoPaquete"+ Integer.toString(contador);
            byte tipoPaquete=Byte.parseByte(props.getProperty(aux4));
            String aux5="openflow.puertoOrigen"+ Integer.toString(contador);
            String pOrigen=props.getProperty(aux5);
            String aux6="openflow.puertoDestino"+ Integer.toString(contador);
            String pDestino=props.getProperty(aux6);
            OFFlowMod fm2 = new OFFlowMod();
            fm2.setCommand(OFFlowMod.OFPFC_ADD);
            fm2.setIdleTimeout((short) 60);
           fm2.setPriority((short) 42);
            OFMatch match2 = OFMatch.load(pi.getPacketData(),
pi.getInPort());
            match2.setInputPort(intEntrada);


//Todas las reglas de flujos serán para paquetes tipo IP (0x800)
            match2.setDataLayerType((short) 0x0800);
            match2.setNetworkSource(IPv4.toIPv4Address(ipOrigen));
           match2.setNetworkDestination(IPv4.toIPv4Address(ipDestino));
```

Código 3.2 Componente Beacon personalizado (continúa...)

```
          match2.setNetworkProtocol(tipoPaquete);

//Ahora se agregan los puertos capa 4, salvo que se especifique no
agregarlos en la regla, con lo cual se modifica el identificador
Wildcards para ignorar los puertos origen o destino, como se indicará luego
del código
          String a = "None";
          if ( pOrigen.equals(a)
{ //no se toma en cuenta el Puerto origen pero si el destino
             match2.setWildcards(3145806);
          }
          else
          {
             match2.setTransportSource(Short.parseShort(pOrigen));
          }
          if ( pDestino.equals(a) )
          { //no se toma en cuenta el puerto destino pero si el origen
             match2.setWildcards(3145870);
          }
          else
          {
            match2.setTransportDestination(Short.parseShort(pDestino))
          }
          fm2.setMatch(match2);
          OFAction action2 = new OFActionOutput(intSalida);
          fm2.setActions(Collections.singletonList((OFAction)action2));
          sw.getOutputStream().write(fm2);
          contador=contador+1;
       }

    }

   //Método existente para añadir un conmutador
    @Override
    public void addedSwitch(IOFSwitch sw) {
    }

 //Y para eliminar la conexión con un conmutador
    @Override
    public void removedSwitch(IOFSwitch sw) {
        macTables.remove(sw);
    }

    //Parámetros beaconProvider para que se fijen en  beaconProvider, que
es parte del controlador
```

Código 3.2 Componente Beacon personalizado (continúa...)

```
    public void setBeaconProvider(IBeaconProvider beaconProvider) {
        this.beaconProvider = beaconProvider;
    }

//Iniciando la ejecución
    public void startUp() {

  log.trace(“Ïniciando");
        beaconProvider.addOFMessageListener(OFType.PACKET_IN, this);
        beaconProvider.addOFSwitchListener(this);
    }

//Método para detener el componente y el controlador
    public void shutDown() {
        log.trace("Deteniendo el controlador");
        beaconProvider.removeOFMessageListener(OFType.PACKET_IN,
this);
        beaconProvider.removeOFSwitchListener(this);
    }

//Método para obtener el nombre del conmutador
    public String getName() {
        return "tutorial";
    }
}
```

Código 3.2 Componente Beacon personalizado

Ahora se explicará el concepto de *wildcards* que se usó en el código del componente. El parámetro *wildcards* permite definir qué parámetros se toman en cuenta cuando se realiza una comparación entre un paquete entrante y las reglas para un determinado flujo; por ejemplo, es posible ignorar la dirección de capa tres origen o el puerto de capa cuatro destino, entre otros. La estructura de este parámetro se observa en la Figura 3.11.

Como se puede observar este campo está representado de manera binaria. Lo que se debe hacer es ubicar un uno para ignorar el campo o un cero para tener en cuenta el campo. En el código este número binario fue transformado a decimal para obtener los números que se observan en él. Esto no se realizó en POX porque este controlador ubica este campo de manera automática infiriéndolo de

las reglas añadidas, mientras que Beacon toma en cuenta todos los campos del parámetro *wildcards*, por lo que no se puede realizar un filtrado correcto si Beacon incluye números de puerto o direcciones de manera automática sin que el administrador lo haya especificado; por ejemplo, añade el puerto origen 234 porque un mensaje le llegó con ese puerto, cosa que el administrador no ha definido, creando una regla poco precisa.

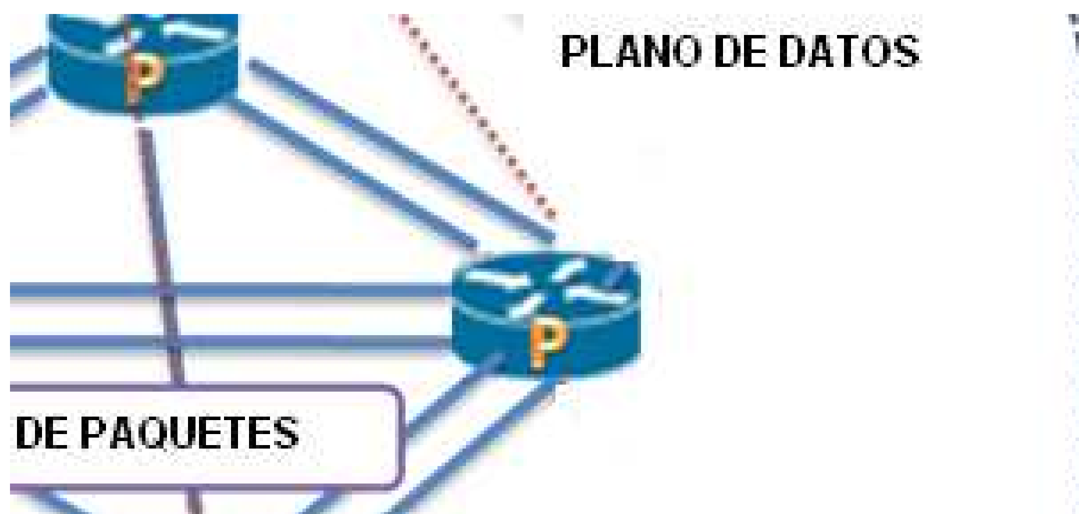


Figura 3.11 Campo Wildcards

3.2.4 FLOODLIGHT

El último controlador que se empleará es Floodlight. La instalación del controlador Floodlight se puede hacer en Linux o en MacOS. Para Linux se necesita un sistema operativo Ubuntu 10.04 o superior, en este caso, el sistema operativo es Ubuntu 12.04LTS que posee los mismos recursos que los anteriores controladores.

La instalación y configuración de Floodlight se encuentra disponible en el Anexo F. Una vez instalado el controlador, se procede a ejecutarlo. Para ello se cambia al directorio de Floodlight y se ejecuta el controlador, como se indica a continuación [10]:

```
$cd floodlight/target
$java –jar floodlight.jar
```

En la Figura 3.12 se muestra la ejecución del servidor controlador Floodlight, que inmediatamente entra en un estado de escucha de peticiones.

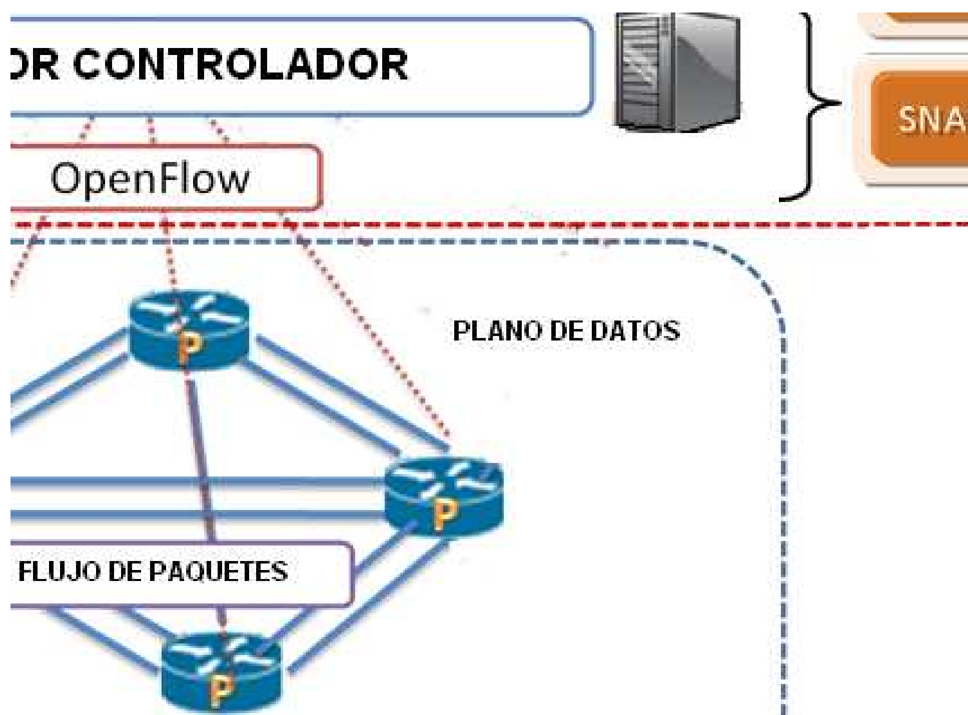


Figura 3.12 Ejecución de Floodlight

En Floodlight, para dar a los dispositivos de conectividad las funcionalidades de conmutadores capa dos, se deben especificar los componentes con los cuales el controlador trabajará para responder a esta necesidad [11]. Por defecto, Floodlight se inicia con la mayoría de componentes que brindan funcionalidades de conmutador capa dos: enrutamiento, descubrimiento de topologías, entre otras. Sin embargo, algunos módulos no son necesarios para brindar las funcionalidades de un conmutador, por lo que se debe especificar cuáles se van a utilizar. Estas especificaciones se encuentran en el archivo learningswitch.properties bajo el directorio /floodlight/src/main/resources y también las especificaciones por defecto en el archivo floodlightdefault.properties, que son las que se toman en cuenta al momento que se ejecuta Floodlight.

Entonces, con esta información, se procede a copiar el contenido del archivo learningswitch.properties al archivo floodlightdefault.properties para que al momento de la ejecución solo se tomen en cuenta los módulos estrictamente necesarios para dar una funcionalidad de conmutador capa dos al dispositivo.

Una vez hecho este cambio, se ejecuta el controlador de la misma manera que se lo hizo anteriormente, salvo que esta vez se cargarán menos módulos.

Ahora, al igual que con los controladores NOX y Beacon, se presenta el código del controlador. Floodlight tiene un componente que permite agregar reglas de flujo estáticas a los conmutadores de manera manual, denominado *Static Flow Pusher*. Este componente es accesible mediante una API REST[10] [11], así que es posible definir una regla de flujo mediante el comando *curl*, que sirve para transferir archivos con sintaxis URL [13].

Por lo mencionado, se ha programado un *script* en Linux para realizar el envío de reglas de flujos a los conmutadores de manera transparente para el administrador, el cual se encargará de tomar las reglas definidas en un archivo de configuración adicional. Antes de programar, es necesario observar la identificación de los conmutadores para diferenciarlos cuando se envíen los comandos *curl.* Para el envío se accede a la interfaz web de Floodlight en la página localhost/ui/index.html, sección *Switch*, en donde se puede observar el identificador de cada conmutador; esto se puede apreciar en la Figura 3.13.

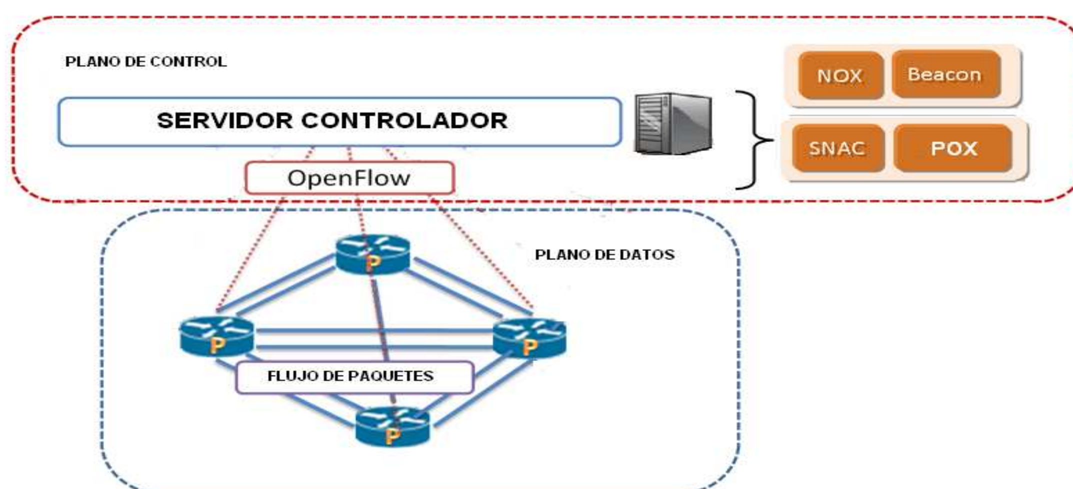


Figura 3.13 Identificación de los conmutadores en Floodlight

Una vez verificado el identificador de cada conmutador, se da paso a la programación del componente, que se muestra en el Código 3.3.

[10] REST: *Representational State Transfer*, describe una interfaz web simple que usa XML y HTTP.

```
#se indica que /bin/bash es el programa para ejecutar el fichero
#!/bin/bash

#se importan las variables definidas en el archivo parámetros
. parametros

#se imprime en pantalla la información del número de flujos
echo "El número de flujos es $numeroFlujos"

#se define una variable denominada CONTADOR y se la inicializa en cero
CONTADOR=0

#Se agregan la reglas para el flujo ARP (identificado por ether-type:
0x0806) entre los puertos físicos 2 y 3 y viceversa para cada conmutador,
usando el identificador adquirido previamente.
curl -d '{"switch": "00:00:00:23:20:b2:8c:28", "name":"flow-mod-arp1",
"ingress-port":"2", "ether-type":"0x0806", "actions":"output=3"}'
http://192.168.1.2:8080/wm/staticflowentrypusher/json

#y el flujo ARP entre los puertos 3 y 2
curl -d '{"switch": "00:00:00:23:20:b2:8c:28", "name":"flow-mod-arp2",
"ingress-port":"3", "ether-type":"0x0806", "actions":"output=2"}'
http://192.168.1.2:8080/wm/staticflowentrypusher/json

curl -d '{"switch": "00:00:00:23:20:6c:4f:eb", "name":"flow-mod-arp3",
"ingress-port":"2", "ether-type":"0x0806", "actions":"output=3"}'
http://192.168.1.2:8080/wm/staticflowentrypusher/json

curl -d '{"switch": "00:00:00:23:20:6c:4f:eb ", "name":"flow-mod-arp4",
"ingress-port":"3", "ether-type":"0x0806", "actions":"output=2"}'
http://192.168.1.2:8080/wm/staticflowentrypusher/json

#Mientras existan reglas, se agregan a cada conmutador
while [[ "$CONTADOR" -lt "$numeroFlujos" ]]; do
#se leen las características de las reglas de flujo del archivo parametros

iOrigen=${interfazOrigen[$CONTADOR]}
iDestino=${interfazDestino[$CONTADOR]}
ipOrigen=${ipOrigen[$CONTADOR]}
ipDestino=${ipDestino[$CONTADOR]}
```

Código 3.3 *Script* para el envío de reglas de flujo en Floodlight (continúa...)

```
tipoPaquete=${tipoPaquete[$CONTADOR]}
pOrigen=${puertoOrigen[$CONTADOR]}
pDestino=${puertoDestino[$CONTADOR]}

#ahora se leen los puertos origen y destino capa 4 del modelo OSI, si
existen se agregan las reglas con la información de los puertos, o si no se
agregan sin la información de puertos.

#Código para flujos con puerto orígen definido
        if [[ "$pOrigen" != "None" ]]; then
        {
        curl -d '{"switch": "00:00:00:23:20:6c:4f:eb ", "name":"flow-mod-
icmp'$CONTADOR'", "ingress-port":"'$iOrigen'", "ether-type":"0x0800",
"protocol": "'$tipoPaquete'", "src-ip": "'$ipOrigen'", "src-dst": "'$ipDestino'",
"src-port": "'$pOrigen'", "actions":"output='$iDestino'"}'
http://192.168.1.2:8080/wm/staticflowentrypusher/json

        curl -d '{"switch": "00:00:00:23:20:b2:8c:28", "name":"flow-mod-
icmp1'$CONTADOR'", "ingress-port":"'$iOrigen'", "ether-type":"0x0800",
"protocol": "'$tipoPaquete'", "src-ip": "'$ipOrigen'", "src-dst": "'$ipDestino'",
"src-port": "'$pOrigen'", "actions":"output='$iDestino'"}'
http://192.168.1.2:8080/wm/staticflowentrypusher/json
        }

#Código para flujos con puerto destino definido
        elif [[ "$pDestino" != "None" ]]; then
        {
        curl -d '{"switch": "00:00:00:23:20:6c:4f:eb "name":"flow-mod-
icmp'$CONTADOR'", "ingress-port":"'$iOrigen'", "ether-type":"0x0800",
"protocol": "'$tipoPaquete'", "src-ip": "'$ipOrigen'", "src-dst": "'$ipDestino'",
"dst-port": "'$pDestino'", "actions":"output='$iDestino'"}'
http://192.168.1.2:8080/wm/staticflowentrypusher/json

        curl -d '{"switch": "00:00:00:23:20:b2:8c:28", "name":"flow-mod-
icmp1'$CONTADOR'", "ingress-port":"'$iOrigen'", "ether-type":"0x0800",
"protocol": "'$tipoPaquete'", "src-ip": "'$ipOrigen'", "src-dst": "'$ipDestino'",
"dst-port": "'$pDestino'", "actions":"output='$iDestino'"}'
http://192.168.1.2:8080/wm/staticflowentrypusher/json
        }

#Código para flujos sin información de puerto origen o destino

        Else
{
```

Código 3.3 *Script* para el envío de reglas de flujo en Floodlight (continúa...)

```
        curl -d '{"switch": "00:00:00:23:20:6c:4f:eb "name":"flow-mod-
icmp'$CONTADOR'", "ingress-port":"'$iOrigen'", "ether-type":"0x0800",
"protocol": "'$tipoPaquete'", "src-ip": "'$ipOrigen'", "src-dst": "'$ipDestino'",
"actions":"output='$iDestino'"}'
http://192.168.1.2:8080/wm/staticflowentrypusher/json

        curl -d '{"switch": "00:00:00:23:20:b2:8c:28", "name":"flow-mod-
icmp1'$CONTADOR'", "ingress-port":"'$iOrigen'", "ether-type":"0x0800",
"protocol": "'$tipoPaquete'", "src-ip": "'$ipOrigen'", "src-dst": "'$ipDestino'",
"actions":"output='$iDestino'"}'
http://192.168.1.2:8080/wm/staticflowentrypusher/json

        }
        fi

#se incrementa la variable contador y se ejecuta el lazo while hasta que ya
no se cumpla la condición
        let CONTADOR=$CONTADOR+1
done
```

Código 3.3 *Script* para el envío de reglas de flujo en Floodlight

En el Anexo F se amplía la sintaxis del comando *curl* empleado en el Código 3.2. En el siguiente subcapítulo se realizará tanto la configuración de flujos así como las pruebas de cada controlador, teniendo como ambiente de pruebas la topología indicada anteriormente.

3.3 PRUEBAS Y RESULTADOS

En esta sección se realizarán las siguientes pruebas: en primer lugar se presentarán las capturas obtenidas con Wireshark de los paquetes enviados a través de los comandos *dpctl* y en segundo lugar se presentarán las pruebas con los componentes programados anteriormente usando reglas de flujo específicas.

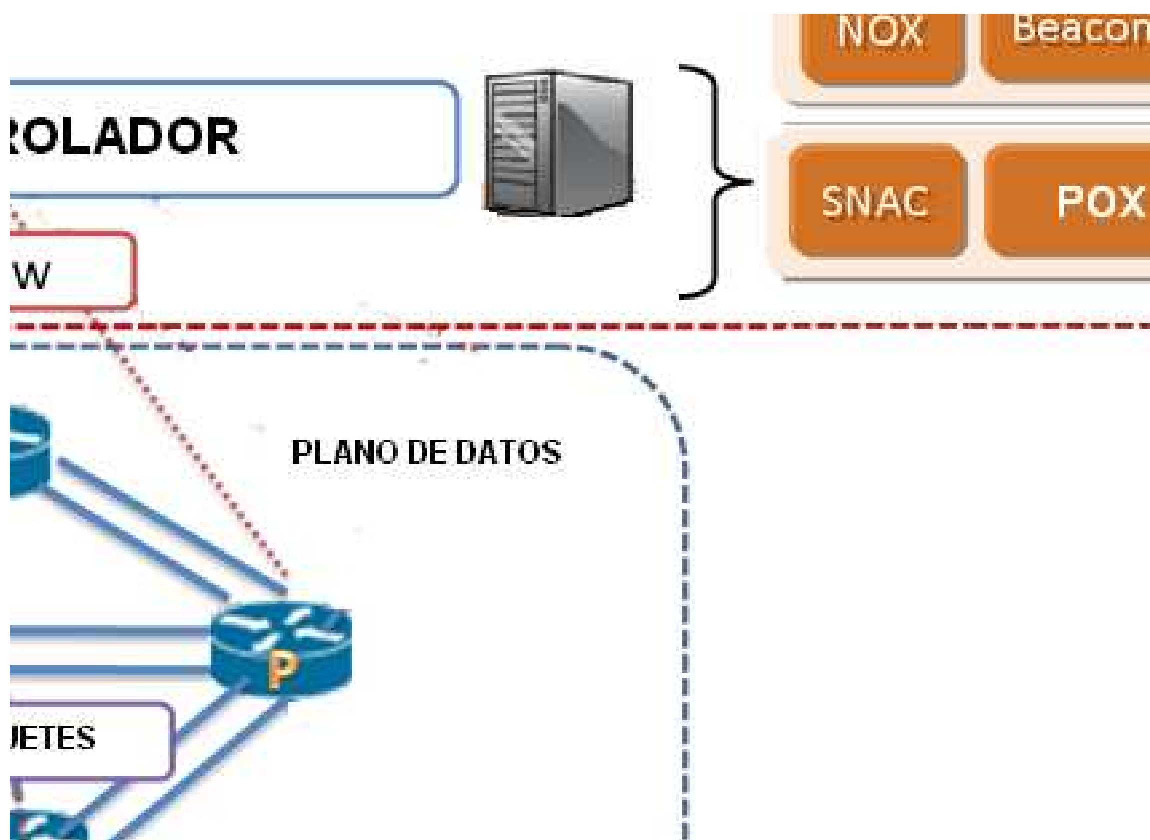
3.3.1 PRUEBAS DE MENSAJES OPENFLOW

En primer lugar se realizarán varias pruebas para comprobar de que manera se realiza el intercambio de mensajes OpenFlow. Se utilizará la herramienta *dpctl*

desde un host adicional ya que en una máquina virtual se ha instalado esta herramienta y el software Wireshark en conjunto. Para más información acerca del detalle de esta instalación referirse al Anexo G.

En primer lugar se verifican las características del equipo OpenWRT S1 con el comando:

```
$dpctl show tcp:192.168.1.1:6633
```



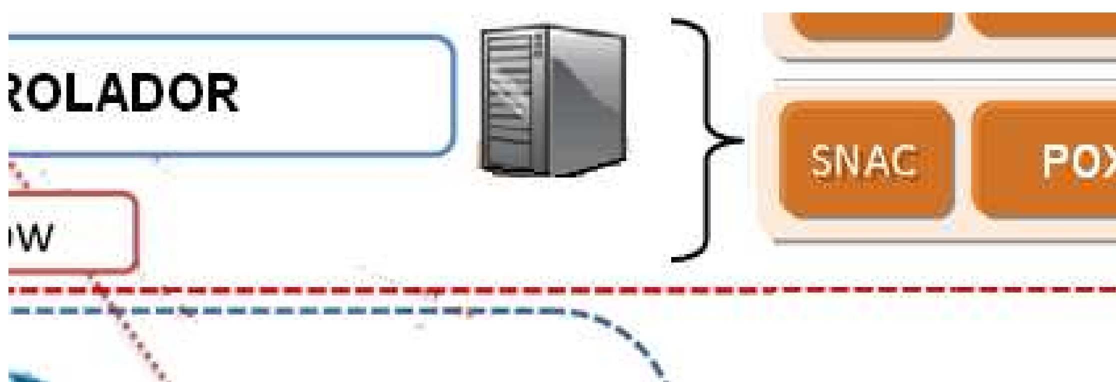
Fligura 3.14 Comando *dpctl show tcp*

En la Figura 3.14 se pueden observar datos importantes, como en el campo **ver** (versión del protocolo OpenFlow), que en este caso corresponde a la versión 1.0, el campo **dpid** muestra el identificador único asignado por el conmutador para esta instancia OpenFlow, **n_tables** que indica el número de reglas de flujo presentes en la tabla de flujos y **n_buffers** que indica el tamaño del buffer del dispositivo y, finalmente, la información de cada puerto con su respectiva

dirección MAC y las configuraciones de las capacidades de canal actuales y máximas de los puertos físicos.

En segundo lugar, se usa el comando *dump-ports* para verificar la información específica de los puertos físicos del dispositivo, el mismo que se detalla a continuación:

```
$dpctl dump-ports tcp:192.168.1.1:6633
```



Fligura 3.15 Comando *dpctl dump-ports*

En la Figura 3.15 se puede observar el resultado de la ejecución del comando. En el detalle se observa que el conmutador tiene cuatro puertos OpenFlow habilitados, cada uno de los cuales tiene estadísticas de transmisión y recepción de paquetes, de los paquetes eliminados y de los paquetes con errores. En este caso no se han detectado errores de transmisión, por lo que en la información se los identifica con un signo de interrogación (?).

Paralelamente a la ejecución del comando, se inició una captura con Wireshark.
En la Figura 3.16 se puede observar que, primer lugar se establece la conexión TCP (paquetes con número 1, 2 y 3) y el primer mensaje OpenFlow que se envía corresponde a un mensaje Hello, generado en el host con la herramienta *dpctl* instalada (192.168.1.2) que de ahora en adelante se denominará “host *dpctl*” hacia el conmutador S1 (192.168.1.1) (paquete 4) y la respuesta llega desde el host *dpctl* hacia el controlador (paquete 6). Estos paquetes se envían siempre que se establece una conexión OpenFlow.

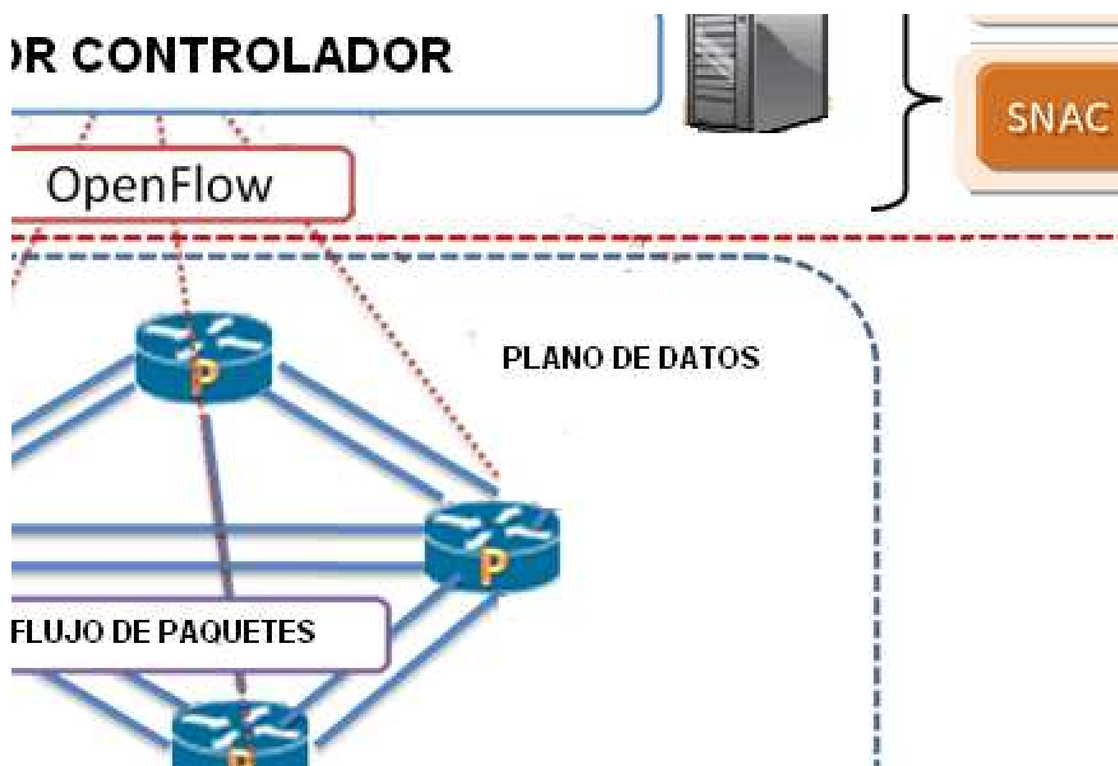


Figura 3.16 Captura de mensajes *Hello*

Posteriormente, se envía la solicitud correspondiente al comando *show* mediante un paquete OpenFlow *Features Request* (paquete número 8) desde el host *dpctl* hacia el conmutador S1, como se indica en la Figura 3.17.

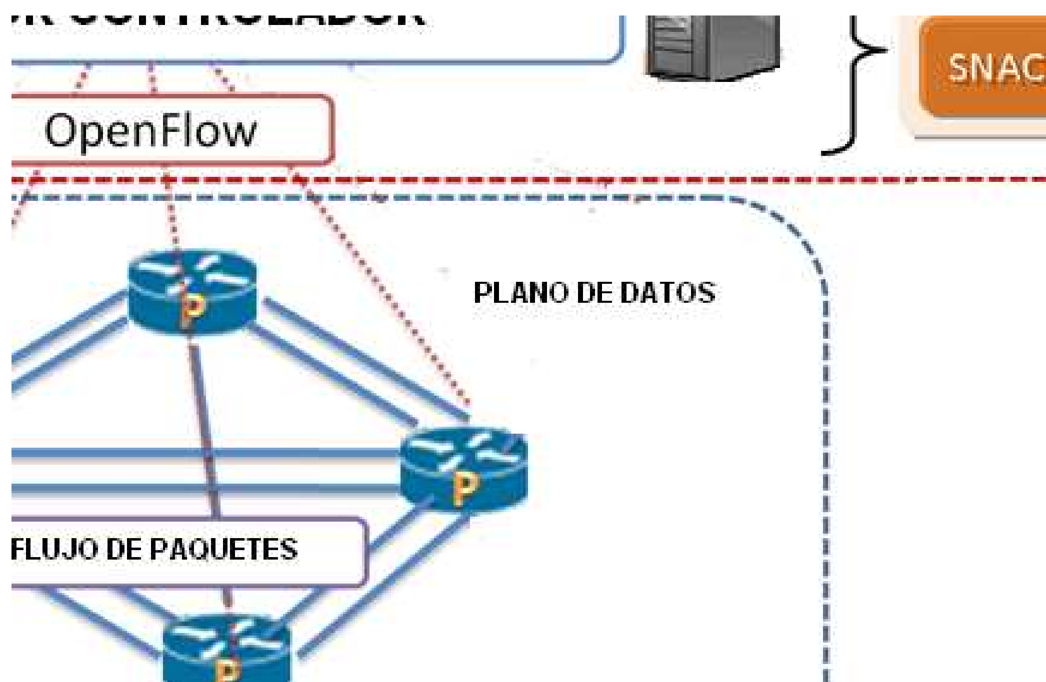


Figura 3.17. Captura del mensaje *Features Request*

Este mensaje viene acompañado de un mensaje *Packet-In* (paquete número 9), que corresponde a un mensaje de control de la conexión TCP. Posteriormente se envía la respuesta con un mensaje *Features Reply* (número 10) desde el host *dpctl* hacia el conmutador, con todos los ítems que conforman la configuración del dispositivo, como se indica en la Figura 3.18.

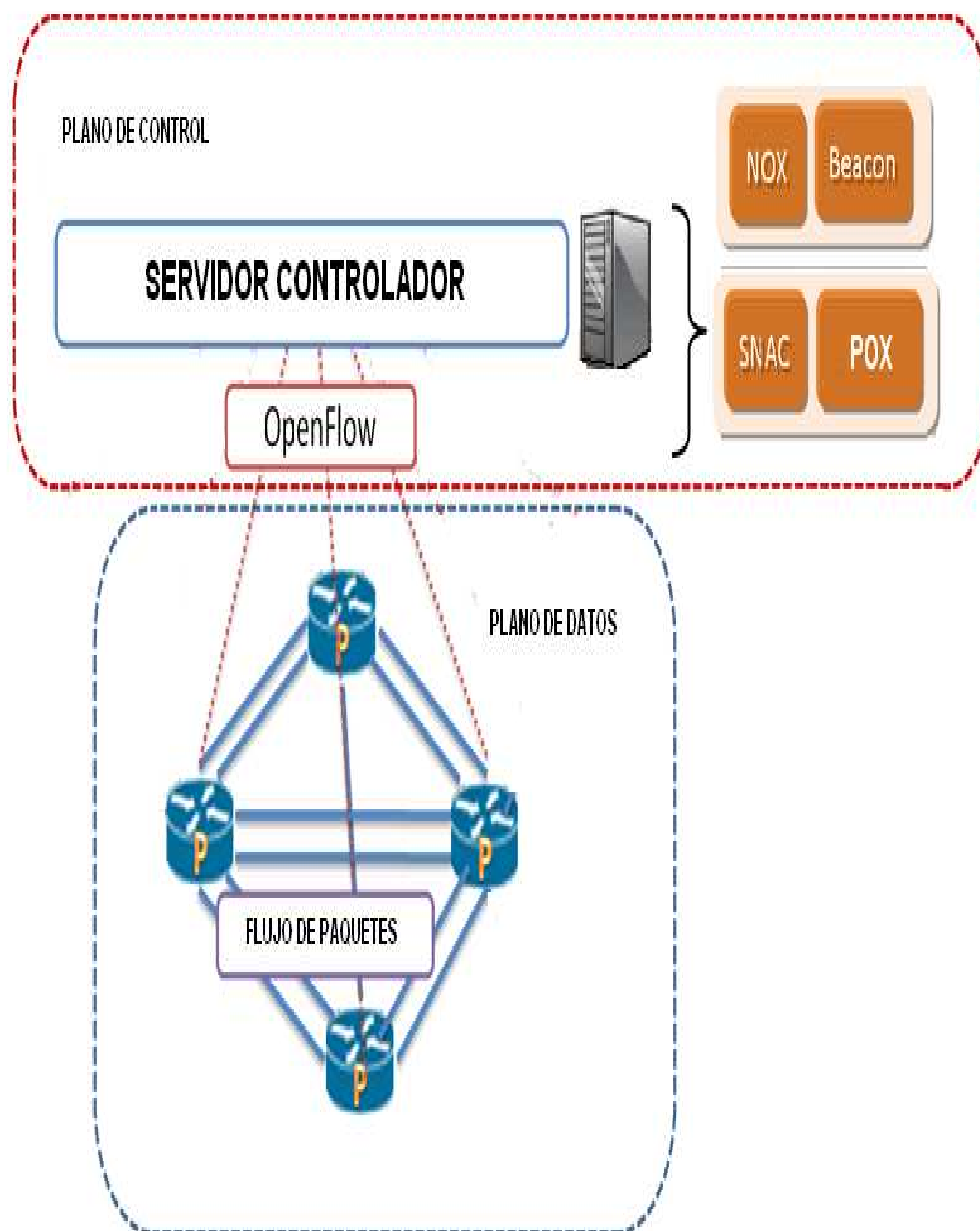


Figura 3.18 Detalle de un paquete *Features Reply*

Y así finaliza el intercambio de paquetes para el comando *dpctl show*. La siguiente prueba es la modificación de puertos. En primer lugar se apagará el puerto tres del conmutador S1 y, posteriormente, se enviará el comando show para verificar el estado del puerto. El comando usado para apagar el puerto se indica a continuación:

```
$dpctl mod-port tcp:192.168.1.1:6633 3 down
```

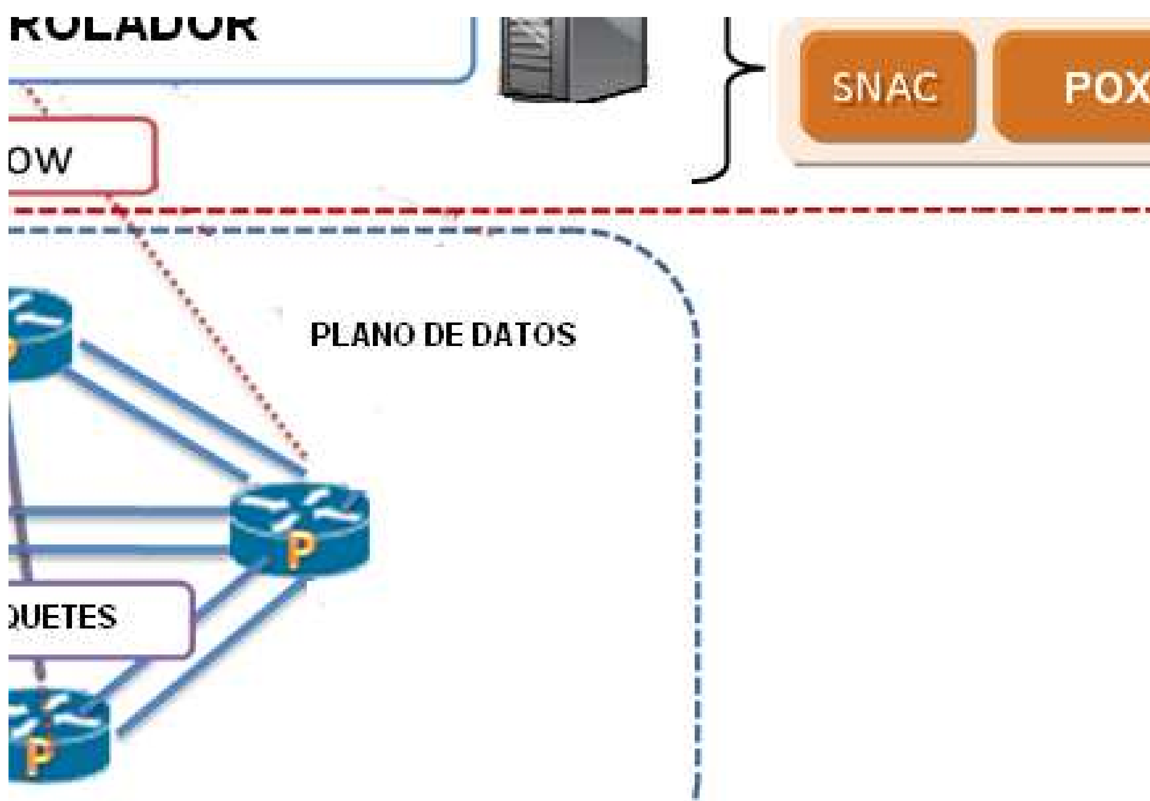


Figura 3.19 Resultado del comando *mod-port* (apagar puerto)

En la Figura 3.19 se ha remarcado la interfaz que se ha deshabilitado. Es necesario tomar en consideración que el campo **config** del puerto 3 (eth0.2) ha cambiado al valor hexadecimal de 0x1 para denotar que el puerto está apagado administrativamente. En la captura de Wireshark de la Figura 3.20 se muestra el mensaje de modificación de puerto (*Port Mod*) (número 15) con la indicación resaltada que el puerto ha sido apagado administrativamente. Los otros parámetros son los mismos que se detallaron en la explicación de la Figura 3.14.

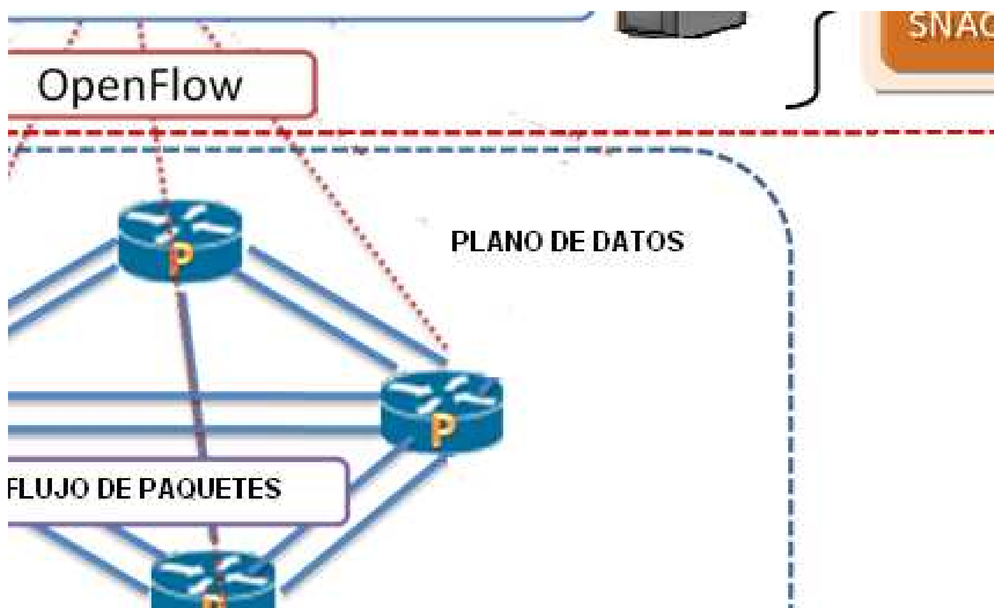


Figura 3.20 Detalle de un mensaje *Port Mod*

Posteriormente se realiza otra modificación de puerto, esta vez para no permitir la inundación o *flooding* en este puerto, es decir; que si una regla indica que se envíe un paquete por todos los puertos, el dispositivo no lo reenviará por este puerto. Esta acción se realiza mediante el comando que se indica a continuación:

```
$dpctl mod-port tcp:192.168.1.1:6633 3 noflood
```

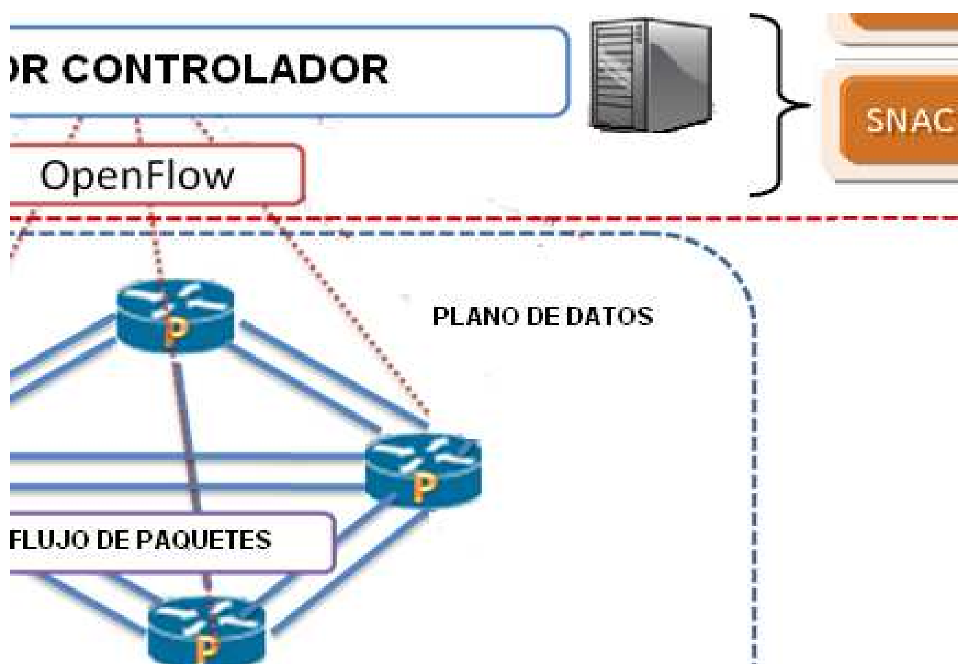


Figura 3.21 Resultado del comando *mod-port* (*noflood*)

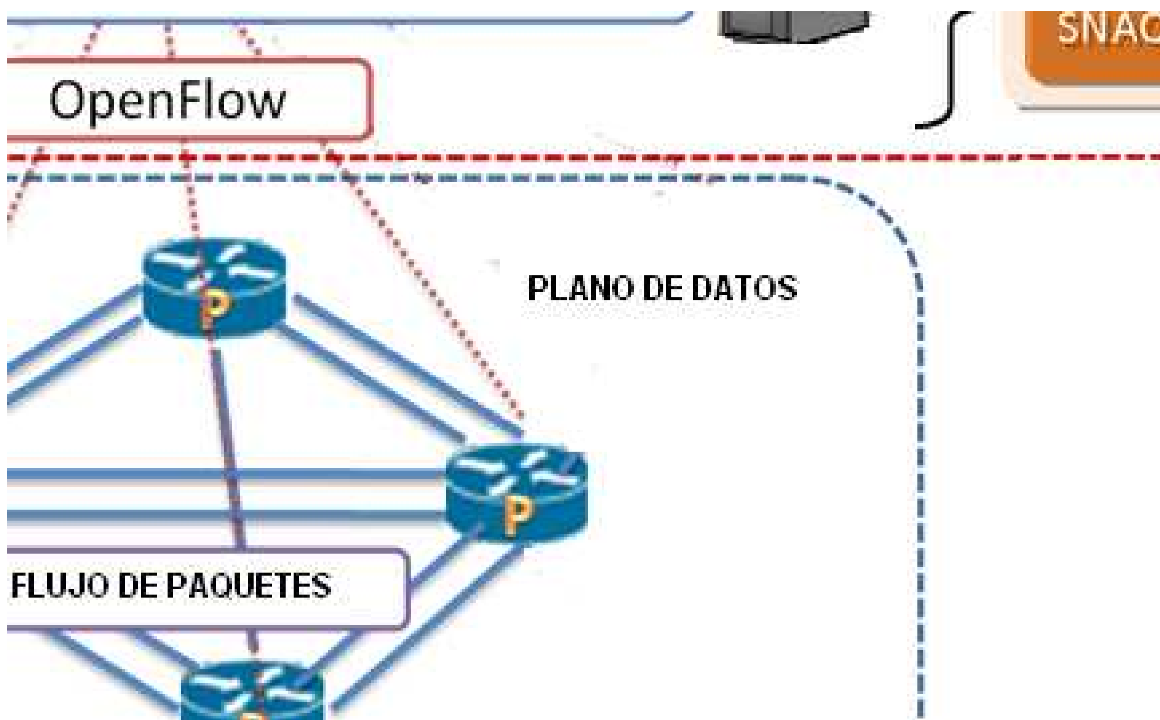


Figura 3.22 Detalle de un mensaje *Port Mod*

En la Figura 3.21 se puede observar que, nuevamente, el estado ha cambiado y esto se corrobora con la captura de Wireshark que se muestra en la Figura 3.22, en la cual el paquete *Port Mod* (número 16) indica que este puerto no se incluye cuando se necesita realizar una inundación.

Ahora se capturará el mensaje más importante con el que cuenta OpenFlow, que sirve para modificar flujos. Para ello, en este ejemplo, se eliminarán todos los flujos definidos en el conmutador S1 mediante el comando:

```
$dpctl del-flows tcp:192.168.1.1:6633
```

En la Figura 3.23 se puede observar que se envía un mensaje *Flow Mod* para la modificación de flujos, el cual indica en el campo comando que se eliminen todos los flujos existentes.

Estos son los mensajes básicos con los que cuenta OpenFlow para llevar a cabo determinadas tareas, los mismos que han sido generados con la herramienta

dpctl; estos mensajes también son enviados por todos los controladores, pero de manera automática cuando se ejecutan.

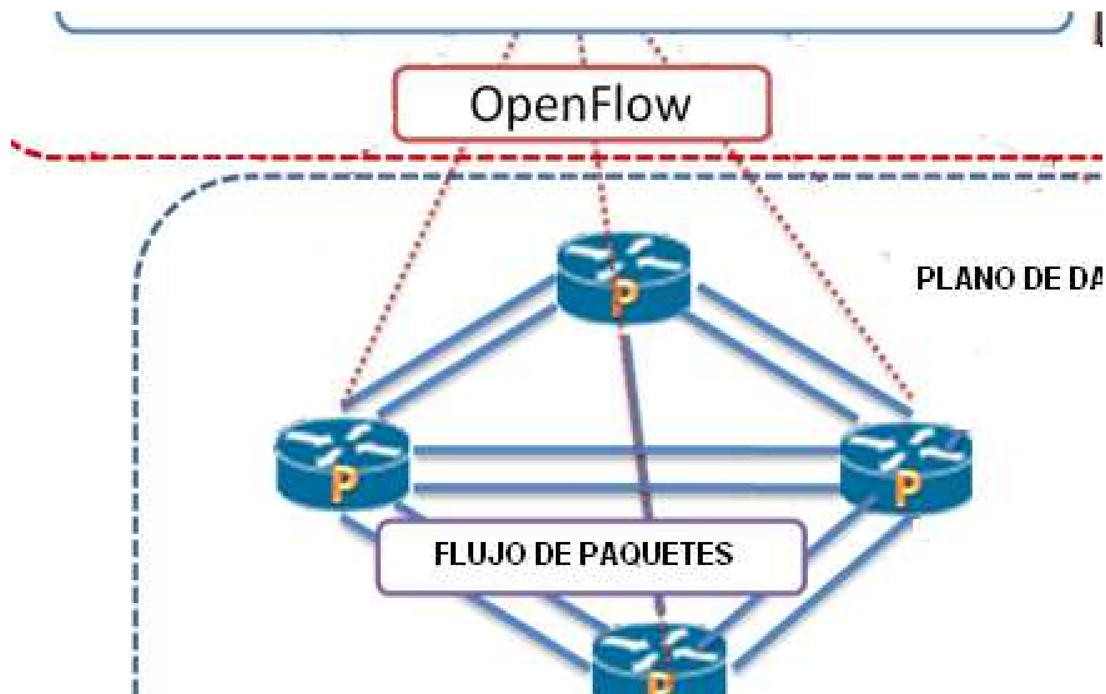


Figura 3.23 Detalle del comando *Flow Mod*

3.3.2 PRUEBAS CON LOS COMPONENTES PERSONALIZADOS

En esta sección se comprobará el funcionamiento de los componentes personalizados indicados en el subcapítulo anterior. En primer lugar, se presentará el archivo de configuración con las reglas de flujo definidas, para, posteriormente, indicar el funcionamiento del componente y, finalmente, verificar que las reglas de flujos se han agregado correctamente mediante la herramienta *dpctl*.

Para las pruebas, se definirán los siguientes flujos en todos los conmutadores, para tener un punto de comparación entre ellos y así buscar similitudes y diferencias en su programación y configuración:

- En primer lugar, el protocolo básico para que los hosts puedan comunicarse es ARP. Sin este protocolo no se tendrá comunicación entre

ningún host, por lo que las reglas correspondientes al intercambio de mensajes de este protocolo se definieron de manera directa en el código del controlador, como se lo pudo constatar en la sección anterior en el código personalizado de los tres componentes.

- En segundo lugar, se definirán reglas para la comunicación mediante mensajes ICMP entre los hosts V11 y V21. Estas reglas se agregan para demostrar, con una prueba de conectividad básica, que las reglas de flujo son tomadas en cuenta por los conmutadores. Por ello, los mensajes ICMP enviados entre estos hosts deben ser exitosos, mientras que los mensajes ICMP entre otros hosts deben ser fallidos.
- En tercer lugar, se usará un protocolo muy manejado en la actualidad, y que es, el que seguramente generará la mayor cantidad de reglas en una implementación a nivel de empresa. Es por ello que se han agregado reglas para la comunicación mediante el protocolo HTTP entre los hosts V11 y V21. Los otros hosts no se podrán comunicar a través de este protocolo.
- En cuarto lugar, se ha decidido usar el protocolo Telnet para comprobar el uso de puertos capa cuatro entre los hosts V12 y V22.
- Finalmente, se ha determinado realizar una prueba con *streaming* de video; para ello se han establecido rutas que permitan el envío de paquetes entre los hosts V13 y V23.

Con esta información se presentan los archivos de configuración de cada controlador.

3.3.2.1 POX

Para el componente POX, programado anteriormente, se presenta el archivo de configuración creado para definir las características de las reglas de flujo. Este archivo se ubica bajo el directorio /pox y su contenido se puede observar en el Archivo de Configuración 3.1.

```
[Section1]
#se define el número total de reglas de flujos
numeroflujos = 8

#Cada parámetro debe tener su correspondiente identificador al final, por
ejemplo la primera regla se identifica con el número cero, la segunda regla
con el número uno y así sucesivamente.
#interfazorigen e interfazdestino deben ser enteros, iporigen e ipdestino
direcciones IP, ipprotocolo debe ser entero y puertoorigen y puertodestino
pueden ser o enteros o la palabra None cuando no se desea especificar
esta información.
#Se especifica “None” cuando no se desea agregar información de los
puertos origen o destino capa 4 del modelo OSI.
#La primera y segunda regla permiten el intercambio de mensajes ICMP
entre los hosts V11 y V21

interfazorigen0 = 2
interfazdestino0 = 3
iporigen0 = 192.168.3.1
ipdestino0 = 192.168.3.2
ipprotocolo0 = 1
puertoorigen0 = None
puertodestino0 = None

interfazorigen1 = 3
interfazdestino1 = 2
iporigen1 = 192.168.3.2
ipdestino1 = 192.168.3.1
ipprotocolo1 = 1
puertoorigen1 = None
puertodestino1 = None

#Se agregan las reglas para la comunicación mediante el protocolo HTTP
entre el host V11 (Servidor) y el host V21 (Cliente)

interfazorigen2 = 3
interfazdestino2 = 2
iporigen2 = 192.168.3.2
ipdestino2 = 192.168.3.1
ipprotocolo2 = 6
puertoorigen2 = None
puertodestino2 = 80

interfazorigen3 = 2
interfazdestino3 = 3
```

Archivo de Configuración 3.1 example.cfg (continúa...)

```
iporigen3 = 192.168.3.1
ipdestino3 = 192.168.3.2
ipprotocolo3 = 6
puertoorigen3 = 80
puertodestino3 = None

#Se agregan las reglas para la comunicación mediante el protocolo Telnet
entre el host V12 (Cliente) y el host V22 (Servidor)

interfazorigen4 = 3
interfazdestino4 = 2
iporigen4 = 192.168.3.6
ipdestino4 = 192.168.3.5
ipprotocolo4 = 6
puertoorigen4 = 23
puertodestino4 = None

interfazorigen5 = 2
interfazdestino5 = 3
iporigen5 = 192.168.3.5
ipdestino5 = 192.168.3.6
ipprotocolo5 = 6
puertoorigen5 = None
puertodestino5 = 23

#Se agregan las reglas para la transmisión de video streaming entre el host
V13 (Cliente) y el host V23 (Servidor) en el puerto 8081

interfazorigen6 = 2
interfazdestino6 = 3
iporigen6 = 192.168.3.9
ipdestino6 = 192.168.3.10
ipprotocolo6 = 6
puertoorigen6 = None
puertodestino6 = 8081

interfazorigen7 = 3
interfazdestino7 = 2
iporigen7 = 192.168.3.10
ipdestino7 = 192.168.3.9
ipprotocolo7 = 6
puertoorigen7 = 8081
puertodestino7 = None
```

Archivo de Configuración 3.1 example.cfg

Ahora se ejecuta el controlador accediendo al directorio /pox, mediante el comando:

```
./pox.py --verbose openflow.of_01 --address=192.168.1.2  --port=6633 of_tutorial
```

En donde **pox.py** corresponde al *script* con el cual el controlador iniciará su ejecución, **--verbose** indica que se ejecutará el controlador con la opción de incluir información adicional de ejecución, **openflow.of_01** indica que se utilizará la versión número uno del protocolo OpenFlow, **--address=192.168.1.2** es la dirección IP del controlador POX, **--port=6633** es el puerto en el que el controlador escuchará las peticiones de los conmutadores y **of_tutorial** es el módulo que se programó anteriormente con el cual se ejecutará el controlador. La ejecución se muestra en la Figura 3.24.

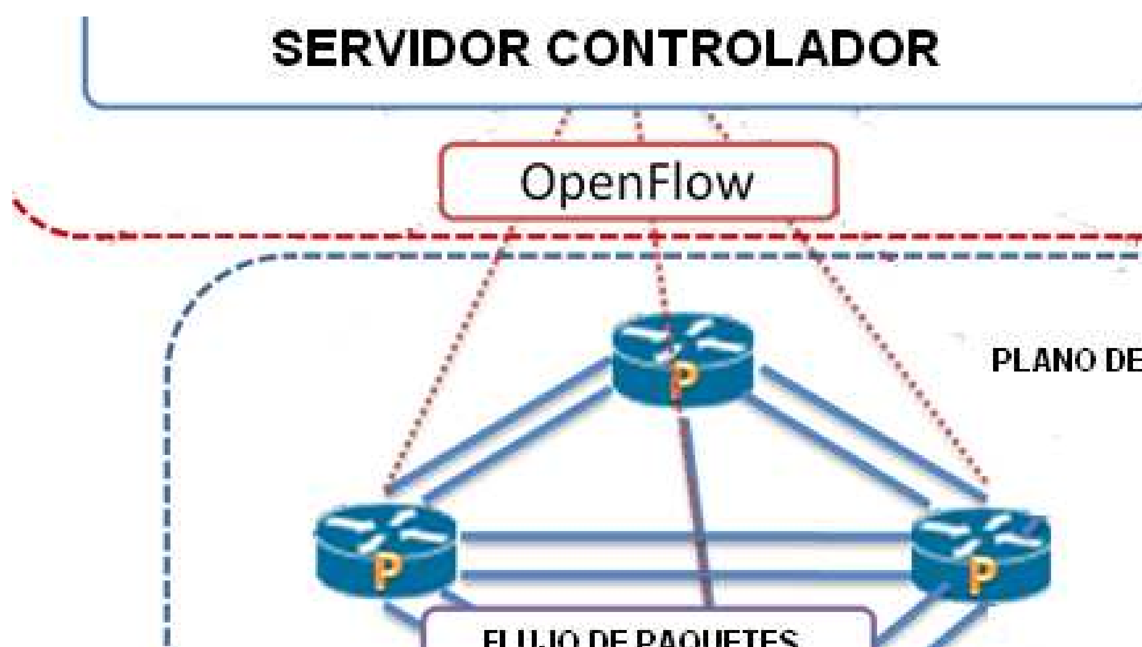


Figura 3.24 Ejecución de POX con el componente of_tutorial

Posteriormente, desde la máquina virtual en la que se instaló la herramienta *dpctl* se verifica que los flujos hayan sido agregados, mediante el comando *dump-flows* para el conmutador S1, no se lo hace para el conmutador S2 porque los flujos son exactamente los mismos para este caso. El resultado de este comando se muestra en la Figura 3.25.

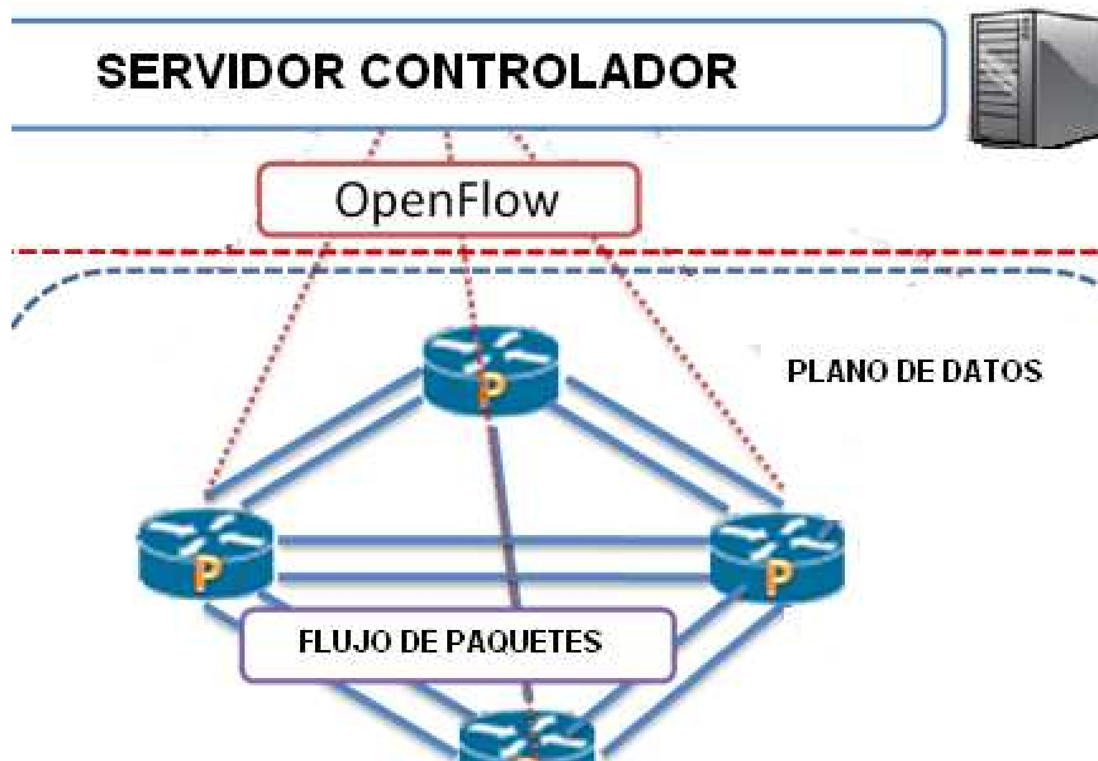


Figura 3.25 Reglas de flujos en el conmutador S1

Y ahora se verifica que las reglas de flujos ingresadas permitan tener comunicación entre los diferentes protocolos. En primer lugar se ejecuta el comando *ping* entre V11 (192.168.3.1) y V21 (192.168.3.2), las peticiones son exitosas. Cualquier otra petición es fallida, En la Figura 3.26 se observa el resultado de esta operación:

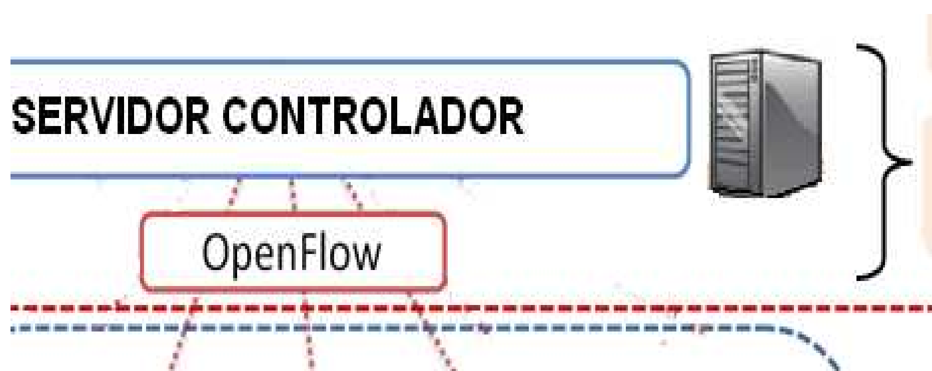


Figura 3.26 Envío de mensajes ICMP entre V11 y V21

Además se puede observar que los tiempos RTT son bastante bajos, debido a que los flujos ya se encuentran instalados en los conmutadores, por lo que los paquetes no deben ser enviados al controlador la primera vez. Ahora se realiza la siguiente prueba, que consiste en una comunicación cliente servidor mediante el protocolo HTTP entre V11 (Cliente) y V21 (Servidor). En la Figura 3.27 se observa que la comunicación es exitosa al abrir un explorador web en V21 y acceder a una página web almacenada en V11.

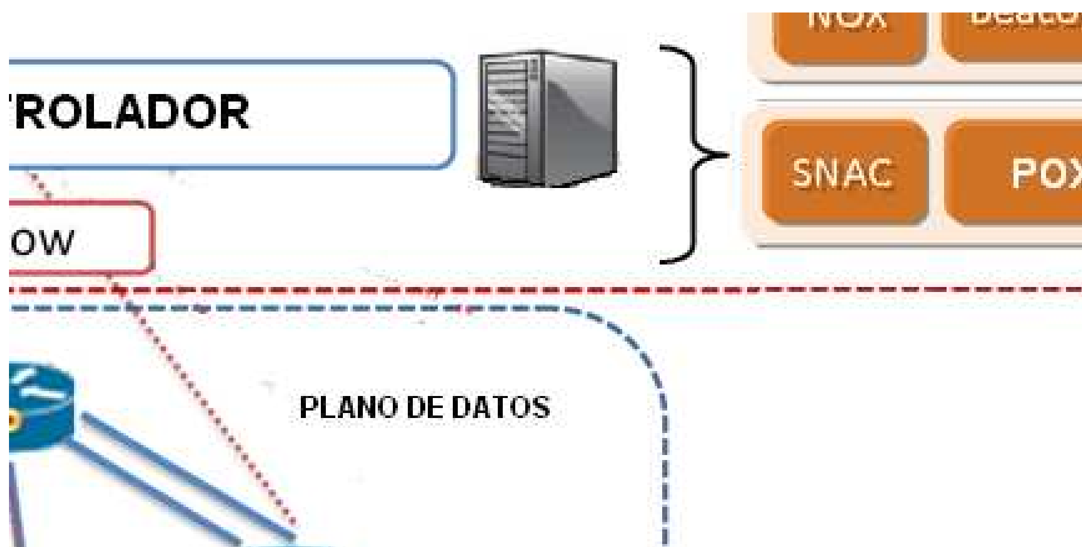


Figura 3.27 Comunicación con HTTP entre V11 y V21

Ahora se prueba la conexión mediante el protocolo telnet, al acceder remotamente a V22 (192.168.3.6) desde V12 (192.168.3.5), cuyo resultado se muestra en la Figura 3.28.

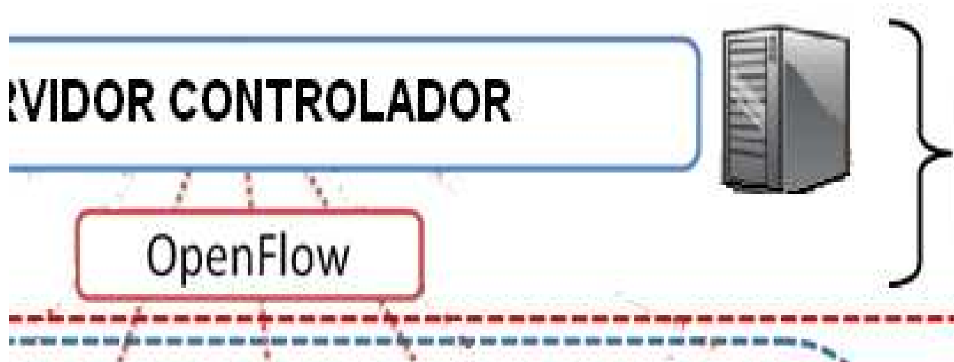


Figura 3.28 Comunicación telnet entre V12 y V22

Finalmente se prueba la última regla de tráfico, que consiste en el acceso a un servidor de video *streaming* ubicado en V23 (192.168.3.10), cuyo servicio escucha en el puerto número 8081, desde V13 (192.168.3.9). Una captura de la transmisión de video se muestra en la Figura 3.29.

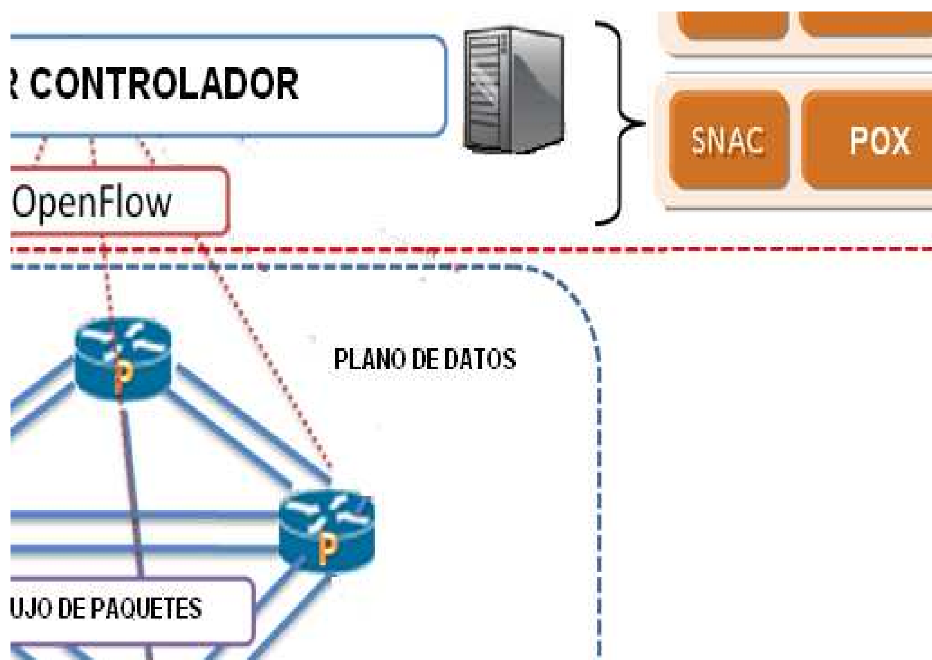


Figura 3.29 Captura de un *streaming* de video originado en V23

Con esta última prueba se verifica que los flujos que se han ingresado cumplen con el propósito para el cual fueron planeados. En los siguientes controladores se definirán exactamente los mismos flujos, para tener un punto de comparación entre la programación de cada uno de ellos.

3.3.2.2 Beacon

Para el componente Beacon, programado anteriormente, se presenta el archivo de configuración creado para definir las características de las reglas de flujo. Este archivo se ubica en el escritorio, se denomina openflow.properties y su contenido se puede observar en el Archivo de Configuración 3.2.

```
#Se define el número de reglas de flujos a agregarse
openflow.numeroFlujos=8

#Cada atributo de cada flujo tiene su identificador, así el primer flujo se
denota con un cero en la parte final, el segundo con un uno y así
sucesivamente
#interfazEntrada e interfazSalida deben ser enteros, ipOrigen e ipDestino
direcciones IP en formato decimal, tipoPaquete debe ser entero y
puertoOrigen y puertoDestino pueden ser o enteros o la palabra None.
#Se especifica "None" cuando no se desea agregar información de los
puertos origen o destino capa 4 del modelo OSI.

#La primera y segunda regla permiten el intercambio de mensajes ICMP
entre los hosts V11 y V21

openflow.interfazEntrada0=2
openflow.interfazSalida0=3
openflow.ipOrigen0=192.168.3.1
openflow.ipDestino0=192.168.3.2
openflow.tipoPaquete0=1
openflow.puertoOrigen0=None
openflow.puertoDestino0=None

openflow.interfazEntrada1=3
openflow.interfazSalida1=2
openflow.ipOrigen1=192.168.3.2
openflow.ipDestino1=192.168.3.1
openflow.tipoPaquete1=1
openflow.puertoOrigen1=None
openflow.puertoDestino1=None

#Se agregan las reglas para la comunicación mediante el protocolo HTTP
entre el host V11 (Servidor) y el host V21 (Cliente)

openflow.interfazEntrada2=2
openflow.interfazSalida2=3
openflow.ipOrigen2=192.168.3.1
openflow.ipDestino2=192.168.3.2
openflow.tipoPaquete2=6
openflow.puertoOrigen2=80
openflow.puertoDestino2=None

openflow.interfazEntrada3=3
openflow.interfazSalida3=2
openflow.ipOrigen3=192.168.3.2
```

Archivo de Configuración 3.2 openflow.properties (continúa...)

```
openflow.ipDestino3=192.168.3.1
openflow.tipoPaquete3=6
openflow.puertoOrigen3=None
openflow.puertoDestino3=80

#Se agregan las reglas para la comunicación mediante el protocolo Telnet
entre el host V12 (Cliente) y el host V22 (Servidor)

openflow.interfazEntrada4=3
openflow.interfazSalida4=2
openflow.ipOrigen4=192.168.3.6
openflow.ipDestino4=192.168.3.5
openflow.tipoPaquete4=6
openflow.puertoOrigen4=23
openflow.puertoDestino4=None

openflow.interfazEntrada5=2
openflow.interfazSalida5=3
openflow.ipOrigen5=192.168.3.5
openflow.ipDestino5=192.168.3.6
openflow.tipoPaquete5=6
openflow.puertoOrigen5=None
openflow.puertoDestino5=23

#Se agregan las reglas para la transmisión de video streaming entre el host
V13 (Cliente) y el host V23 (Servidor) en el puerto 8081

openflow.interfazEntrada6=2
openflow.interfazSalida6=3
openflow.ipOrigen6=192.168.3.9
openflow.ipDestino6=192.168.3.10
openflow.tipoPaquete6=6
openflow.puertoOrigen6=None
openflow.puertoDestino6=8081

openflow.interfazEntrada7=3
openflow.interfazSalida7=2
openflow.ipOrigen7=192.168.3.10
openflow.ipDestino7=192.168.3.9
openflow.tipoPaquete7=6
openflow.puertoOrigen7=8081
openflow.puertoDestino7=None
```

Archivo de Configuración 3.2 openflow.properties

Desde el entorno de desarrollo en donde se importó el código de Beacon, de acuerdo a lo señalado en el Anexo E, en este caso Eclipse, y con el proyecto Beacon agregado, se ejecuta el controlador con el componente desde la pestaña *Run->Debug->Configuration*. En el menú de exploración a la izquierda se abre la pestaña llamada *OSGI Framework* y se selecciona el archivo beaconTutorialLearningSwitch y se da un clic en el botón *Debug* de la parte inferior derecha, como se muestra en la Figura 3.30.

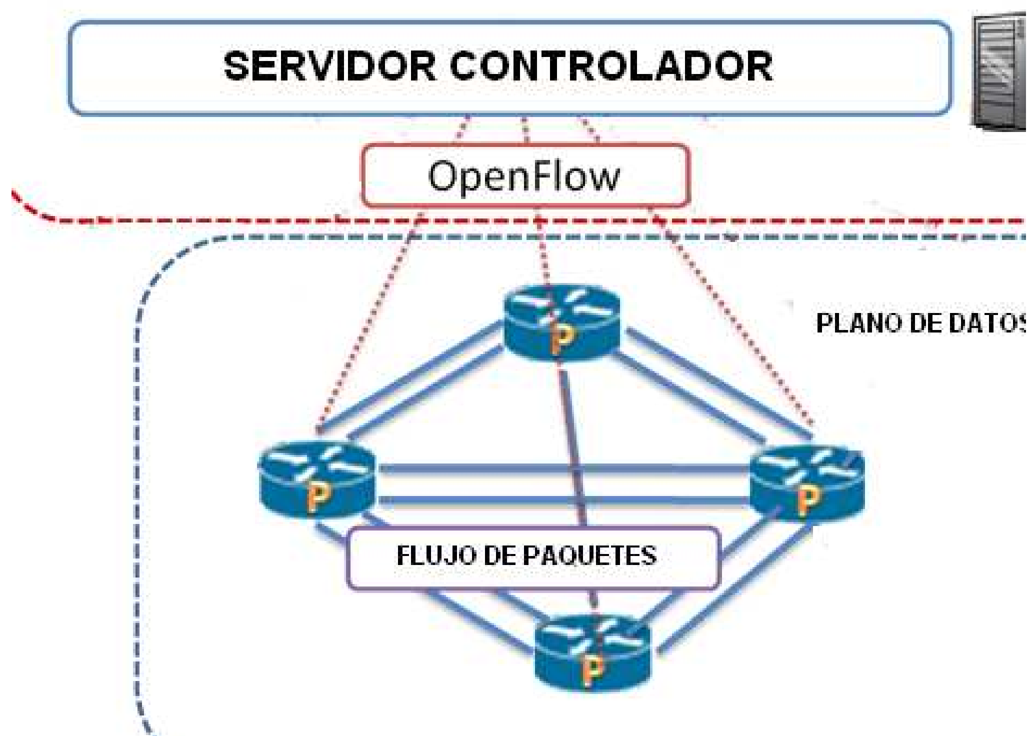


Figura 3.30 Ejecución de Beacon con el componente personalizado

El controlador empezará a ejecutarse y en unos instantes se conectarán los conmutadores de manera automática. Ahora se realizan las mismas pruebas que en POX y el resultado será prácticamente idéntico, por lo que se omiten las capturas de pantalla al ser bastante similares, lo único que difiere es el envío de mensajes ICMP, en el cual los mensajes son enviados con RTT diferentes. Desde el host *dpctl* se ejecuta nuevamente el comando *dump-flows* hacia el conmutador S1, cuyo resultado se muestra en la Figura 3.31.

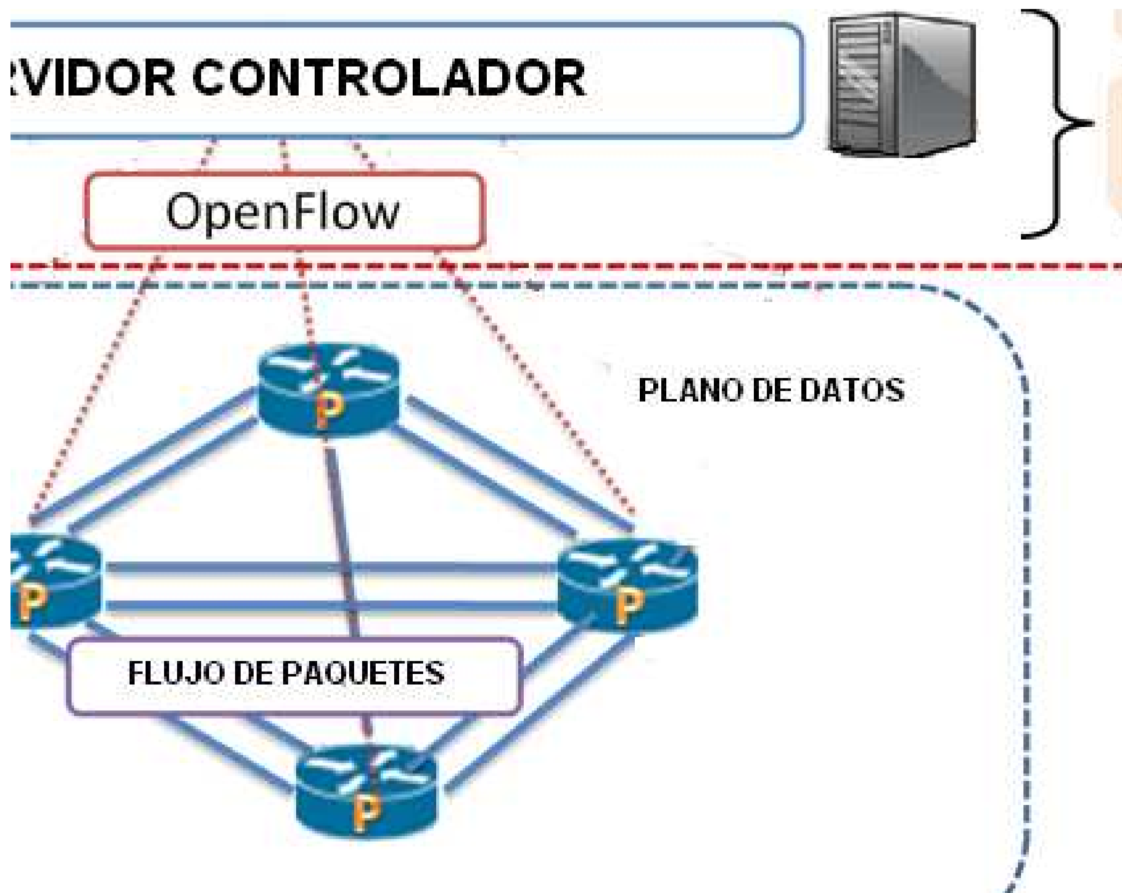


Figura 3.31 Reglas de flujos en el conmutador S1

3.3.2.3 Floodlight

Para el componente Floodlight programado anteriormente se presenta el archivo de configuración creado para definir las características de las reglas de flujo. Este archivo se ubica dentro del directorio /floodlight, se denomina parametros y su contenido se puede observar en el Archivo de Configuración 3.3.

```
#De igual manera que los archivos anteriores se define el número de reglas de flujos
numeroFlujos=8

#Cada atributo de cada flujo tiene su identificador, así el primer flujo se denota con un cero en la parte final, el segundo con un uno y así sucesivamente. Este identificador se ubica entre corchetes [ ].
```

Archivo de Configuración 3.3 parametros (continúa...)

```
#interfazOrígen e interfazDestino deben ser enteros, ipOrigen e
ipDestino direcciones IP en formato decimal, tipoPaquete debe ser
entero y puertoOrigen y puertoDestino pueden ser o enteros o la
palabra None.
#Se especifica “None” cuando no se desea agregar información de los
puertos origen o destino capa 4 del modelo OSI.

#La primera y segunda regla permiten el intercambio de mensajes
ICMP entre los hosts V11 y V21

interfazOrigen[0]=2
interfazDestino[0]=3
ipOrigen[0]=192.168.3.1
ipdestino[0]=192.168.3.2
tipoPaquete[0]=1
puertoOrigen[0]=None
puertoDestino[0]=None

interfazOrigen[1]=3
interfazDestino[1]=2
ipOrigen[1]=192.168.3.2
ipDestino[1]=192.168.3.1
tipoPaquete[1]=1
puertoOrigen[1]=None
puertoDestino[1]=None

#Se agregan las reglas para la comunicación mediante el protocolo
HTTP entre el host V11 (Servidor) y el host V21 (Cliente)

interfazOrigen[2]=2
interfazDestino[2]=3
ipOrigen[2]=192.168.3.1
ipdestino[2]=192.168.3.2
tipoPaquete[2]=6
puertoOrigen[2]=80
puertoDestino[2]=None

interfazOrigen[3]=3
interfazDestino[3]=2
ipOrigen[3]=192.168.3.2
ipDestino[3]=192.168.3.1
tipoPaquete[3]=6
puertoOrigen[3]=None
puertoDestino[3]=80
```

Archivo de Configuración 3.3 parametros (continúa...)

```
#Reglas para la comunicación mediante Telnet entre el host V12 (Cliente) y
el host V22 (Servidor)

interfazOrigen[4]=3
interfazDestino[4]=2
ipOrigen[4]=192.168.3.6
ipDestino[4]=192.168.3.5
tipoPaquete[4]=6
puertoOrigen[4]=23
puertoDestino[4]=None

interfazOrigen[5]=2
interfazDestino[5]=3
ipOrigen[5]=192.168.3.5
ipdestino[5]=192.168.3.6
tipoPaquete[5]=6
puertoOrigen[5]=None
puertoDestino[5]=23

#Se agregan las reglas para la transmisión de video streaming entre el host
V13 (Cliente) y el host V23 (Servidor)

interfazOrigen[6]=2
interfazDestino[6]=3
ipOrigen[6]=192.168.3.9
ipdestino[6]=192.168.3.10
tipoPaquete[6]=6
puertoOrigen[6]=None
puertoDestino[6]=8081

interfazOrigen[7]=3
interfazDestino[7]=2
ipOrigen[7]=192.168.3.10
ipDestino[7]=192.168.3.9
tipoPaquete[7]=6
puertoOrigen[7]=8081
```

Archivo de Configuración 3.3 parametros

Ahora se ejecuta Floodlight como se indicó en el subcapítulo anterior y se ejecuta, paralelamente, el *script* que se programó también en el subcapítulo anterior, con el siguiente comando:

```
./crearFlujos.sh
```

El resultado de la ejecución del *script* se muestra en la Figura 3.32.

Figura 3.32 Ejecución del *script* para la creación de reglas de flujos

Ahora, aprovechando la interfaz web de Floodlight se pueden observar las reglas de flujos añadidas entrando a la página http://localhost:8080, haciendo clic en la pestaña *Switches* y luego en uno de los conmutadores. El resultado de esta operación en el conmutador S1 se muestra en la Figura 3.33.

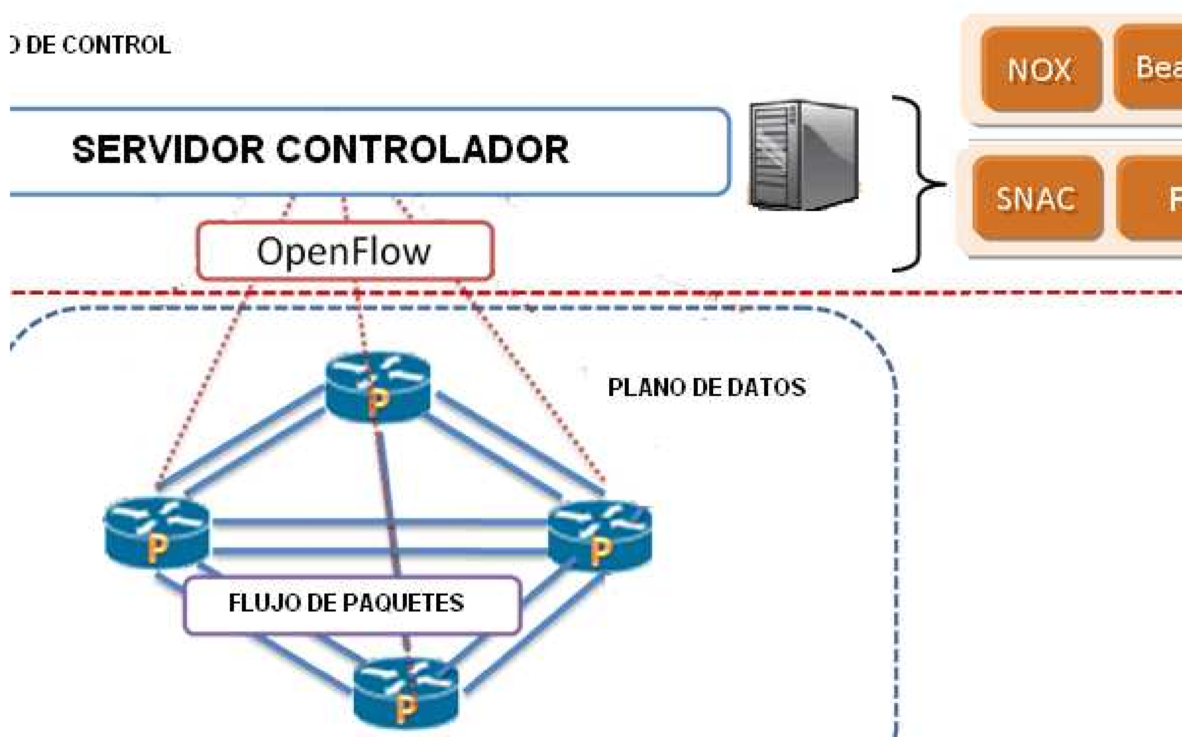


Figura 3.33 Visualización en Floodlight de las reglas añadidas en S1

En la Figura 3.3 se visualiza el parámetro *cookie*, explicado anteriormente, la prioridad de cada regla de flujo, la estructura de comparación: el puerto de entrada, el tipo de paquete Ethernet, el identificador de protocolo de la cabecera IP (1=ICMP, 6=TCP), las direcciones IP origen y destino y los puertos origen y destino. Se visualizan además las acciones para cada regla y sus estadísticas.

Finalmente, al igual que los controladores anteriores, se verifican los flujos con *dpctl* en el conmutador S1, cuyo resultado se muestra en la Figura 3.34.

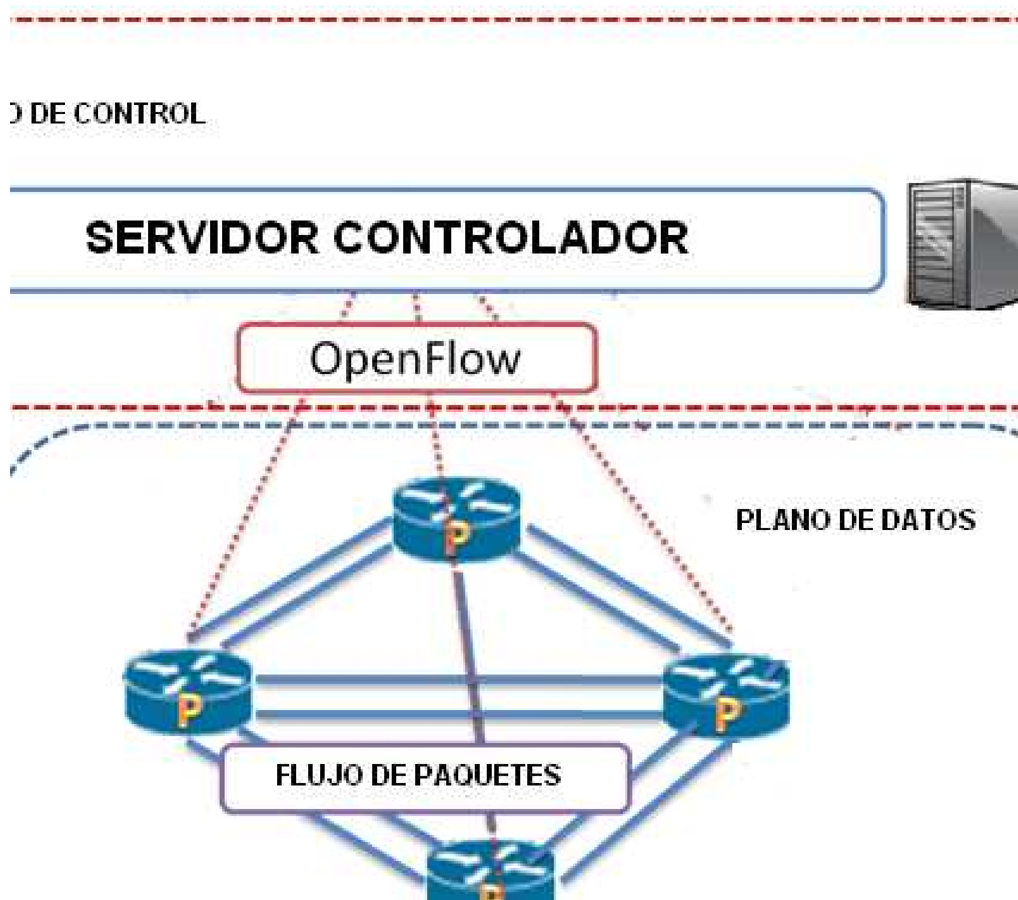


Figura 3.34 Reglas de flujos en el conmutador S1

Se hacen las pruebas con las aplicaciones y el resultado es el mismo que en los dos controladores anteriores. Las capturas de pantalla se omitieron por ser prácticamente iguales a la primera captura.

3.3.2.4 Análisis de resultados

Para el análisis de resultados, se puede observar que la programación de los tres componentes es prácticamente la misma, lo único que cambia son los nombres de ciertos métodos o variables y la manera como se llama a los archivos de configuración.

En la programación de los componentes se puede observar que el usuario tiene el control completo sobre todo el tráfico que pasa por la red y lo puede encaminar de acuerdo a sus necesidades. Un conmutador OpenFlow, haciendo una analogía con las redes tradicionales, puede ser un conmutador, un enrutador y un *firewall* a la vez, lo que representa una ventaja tanto de rendimiento, económica y de espacio, al tener todo en un mismo equipo.

Las reglas que se definieron pueden modificarse para cualquier necesidad que tenga el administrador. Se puede bloquear hosts con determinadas direcciones, dar permisos a segmentos de una compañía, permitir el acceso desde Internet únicamente a los servidores web de una compañía, mientras se permite el acceso de los usuarios de la red de la compañía a todos los recursos, etc.

El limitante es solamente la capacidad de programación de reglas; que, como se pudo observar anteriormente, usan estructuras determinadas dependiendo del tipo de controlador, así que es posible hacer cualquier tarea que el administrador desee en la red que administra.

Con relación a la simulación en Mininet, se puede constatar que la programación en la simulación y en la implementación del prototipo es prácticamente igual. Sin embargo, cabe recalcar que la simulación tiene varios limitantes, como el tipo de tráfico que se maneja, ya que el tráfico por defecto es ICMP y si se requiere otro tipo de tráfico se debe recurrir a programarlo. Otra limitante es la interfaz de *loopback* que maneja, la cual no permite diferenciar tráfico de los conmutadores y del controlador cuando se inicia una captura en Wireshark.

Como se indicó en la sección NOX, se realizaron varias pruebas con el controlador y los conmutadores que presentaron problemas de conexión intermitente. Este problema se debe a la incompatibilidad entre el *firmware* del conmutador y el software del controlador. Sin embargo, en el capítulo anterior se pudo implementar un componente personalizado en NOX sin ningún tipo de problema de manera virtual. Pueden existir problemas de compatibilidad con estos conmutadores, pero esto no quiere decir que el componente programado

para NOX en el capítulo anterior no tenga utilidad, es más se lo puede llevar a un ambiente físico con conmutadores que no tengan estos problemas y tendrá un funcionamiento similar a la simulación. Además, puede ser que los problemas se puedan solucionar con un nuevo *firmware* que se introduzca al mercado.

A pesar de que se realizaron las acciones correctivas indicadas en la literatura [14] [15], el controlador nunca tuvo un comportamiento estable, por lo que se llevaron a cabo pruebas en la simulación que no pudieron ser comprobadas en el prototipo.

3.4 PRESUPUESTO REFERENCIAL DEL PROTOTIPO

El presupuesto referencial tomando en cuenta los equipos detallados anteriormente se indica a continuación:

- Tres computadores con las características de hardware incluidas anteriormente, uno de ellos para el servidor controlador y dos para trabajar como hosts. Cada computador cuesta 1138 dólares americanos con cincuenta centavos, lo que da un total de 2277 dólares americanos
- Dos Access Point Linksys WRT54GL. Puesto que estos equipos ya no se encuentran disponibles en el mercado, en su lugar se ha cotizado los equipos TP-LINK TL-WR1043, que tienen las mismas características que los equipos Cisco y su precio es similar. Cada Access Point cuesta 89 dólares americanos con cincuenta centavos, lo que da un total de 179 dólares americanos.
- Rubros por instalación y configuración de los Access Point y el controlador: 20 dólares americanos la hora. El tiempo de capacitación, implementación y pruebas es de 200 horas. Lo que da un total de 4000 dólares americanos.

Tomando en cuenta estas consideraciones, el presupuesto referencial asciende a 6456 dólares americanos. En el Anexo H se detallan las proformas en las cuales se ha basado este presupuesto referencial. En el próximo capítulo se presentarán

las conclusiones y recomendaciones obtenidas del presente Proyecto de Titulación.

REFERENCIAS:

[1] VMware vSphere Hypervisor. http://www.vmware.com/products/vsphere-hypervisor/overview.html (Consultado el 5 de Mayo de 2013)

[2] Linksys. http://support.linksys.com/en-us/support/routers/WRT54GL (Consultado el 20 de Mayo de 2013)

[3] OpenWRT. https://openwrt.org/ (Consultado el 20 de Mayo de 2013)

[4] Openflow OpenWRT. http://www.openflow.org/wk/index.php/OpenFlow_1.0_for_OpenWRT (Consultado el 20 de Mayo de 2013)

[5] http://mytestbed.net/projects/openflow/wiki/Installing_a_NOX_controller (Consultado el 3 de Junio de 2013)

[6] http://networkstatic.net/nox-openflow-controller-install-on-ubuntu-12-04-precise-lts/ (Consultado el 3 de Junio de 2013)

[7] NOX Repo. https://github.com/noxrepo/nox-classic/wiki/Installation.DebianUbuntu (Consultado el 3 de Junio de 2013)

[8] POX Stanford. https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-InstallingPOX (Consultado el 9 de Junio de 2013)

[9] Beacon Stanford. https://openflow.stanford.edu/display/Beacon/Configuration (Consultado el 12 de Junio de 2013)

[10] Floodlight. http://www.projectfloodlight.org/getting-started/ (Consultado el 17 de Junio de 2013)

[11] Project Floodlight. http://docs.projectfloodlight.org/display/floodlightcontroller/ (Consultado el 17 de Junio de 2013)

[12] Tutorial OpenFlow. http://www.openflow.org/wk/index.php/OpenFlow_Tutorial (Consultado el 17 de Junio de 2013)

[13] Guía del comando Curl. http://curl.haxx.se (Consultado el 19 de Junio de 2013)

[14] https://mailman.stanford.edu/pipermail/openflow-discuss/2011-July/002483.html (Consultado el 22 de Junio de 2013)

[15] https://mailman.stanford.edu/pipermail/openflow-discuss/2010-December/001704.html (Consultado el 22 de Junio de 2013)

CAPÍTULO IV

4. CONCLUSIONES Y RECOMENDACIONES

4.1 CONCLUSIONES

- Al finalizar el presente Proyecto de Titulación se tiene disponible la implementación de un prototipo de una SDN operacional, compuesto por un servidor controlador, dos conmutadores y seis hosts, mediante el cual se pudieron cumplir los objetivos que se plantearon originalmente en el Plan de Proyecto de Titulación. Mediante esta red se pudo constatar los beneficios de una SDN; en particular, la capacidad que le brinda al administrador de moldear la red según sus necesidades. En el prototipo de red se definieron reglas para flujos de paquetes determinados y se constató que estas reglas se pueden alterar de una manera rápida para adaptarse a los cambios que surjan en la red, mediante la modificación de archivos de configuración.

- La arquitectura de las SDN es prácticamente la misma que se tendría en una red tradicional, salvo que tiene un servidor o grupo de servidores controladores en donde básicamente se concentra la toma de decisiones, cosa que en una red tradicional se realizaba en cada equipo de conmutación o enrutamiento, lo que abría la puerta para tener huecos y políticas de seguridad inconsistentes. Cabe recalcar que se puede tener un conjunto de servidores que actúen de manera coordinada como una entidad controladora. Esto se realiza para evitar centralizar toda la toma de decisiones en un solo equipo físico. A partir de un manejo centralizado de las políticas de seguridad se puede controlar de manera más eficiente

la red en caso de incursiones o ataques malintencionados porque existe una menor probabilidad de inconsistencia en las reglas de flujos. Durante las pruebas con el prototipo se permitió solo cierto tipo de tráfico, por ejemplo tráfico correspondiente al protocolo HTTP entre dos hosts determinados. Esto se puede ampliar y adaptar a las exigencias de la red cada vez que el administrador así lo considere.

- La arquitectura SDN tiene como principio fundamental la separación de planos de control y de datos y se basa en el protocolo OpenFlow para la comunicación entre el o los servidores controladores y los conmutadores. Durante las pruebas del prototipo se pudo constatar esta aseveración que se realizó en el marco teórico. Por un lado, se establecieron conexiones entre los conmutadores y el controlador que solo manejan tráfico de administración, encapsulado en el protocolo OpenFlow. A este tipo de tráfico corresponden los mensajes de modificación de flujos, de establecimiento del enlace OpenFlow, etc. Por otro lado se establecieron enlaces entre los hosts y los conmutadores que solo manejan los datos de las aplicaciones. De esta situación se puede comprobar que los enlaces de datos no se ocupan para la gestión de la red y por tanto tienen una mayor capacidad de canal para datos, lo que supone una gran ventaja en redes altamente congestionadas.

- Durante la simulación del prototipo se utilizó la herramienta Mininet, que permite simular redes SDN en un solo host o máquina virtual como si fuera una red física. En el presente Proyecto de Titulación se usó una máquina virtual, en la que se encuentra incluido el controlador NOX, para el cuál se programó un módulo o componente de software; en él, se definieron reglas para ser transmitidas a cada conmutador para que el tráfico ICMP se envíe entre dos hosts y el tráfico restante se elimine. No se amplió a otro tipo de tráfico porque se debe recurrir a programarlo, situación que sale fuera del alcance del presente Proyecto de Titulación. Sin embargo, esto fue suficiente para demostrar que se pueden programar reglas que moldeen el tráfico y verificar que el administrador

tiene el control absoluto de la red. La herramienta es muy versátil para propósitos de aprendizaje, pero tiene limitantes en cuanto al manejo de una mayor variedad de tipo de tráfico, como se lo mencionó anteriormente, y no permite una medición de retardos verídica, al no considerarse retardos de transmisión de los enlaces concretos de una implementación física.

- Existen varias alternativas de software que permiten la implementación de un servidor controlador: en primer lugar está NOX, en el cual se presentaron problemas de intermitencia en las conexiones entre los conmutadores y el controlador, debido a problemas de compatibilidad entre el *firmware* de los conmutadores y el software del controlador. Estos problemas surgen al momento de la conexión y el establecimiento de la comunicación mediante OpenFlow entre un conmutador y el controlador. Esto causa que el conmutador se conecte en ciertos instantes de tiempo y se desconecte casi inmediatamente. Esta situación se agudiza cuando se conectaron los dos conmutadores al controlador, ya que no se pudo establecer la conexión entre ellos en repetidas ocasiones. Se recurrió a la literatura y se implementaron varias soluciones recomendadas en Internet, pero ninguna solución dio resultados concretos. A pesar de que el *firmware* de los conmutadores no es compatible con el controlador NOX, el módulo programado puede ser adaptado a una implementación real con otros equipos de conmutación o con los mismos equipos, pero con un *firmware* mejorado, que no se encontraba disponible al momento de la implementación, pero como está en continuo desarrollo, puede corregir estos problemas eventualmente. En resumen, se implementó un módulo para el servidor controlador NOX en Mininet y se comprobó que su funcionamiento era adecuado, en base a las reglas definidas. El componente se puede adaptar para implementaciones físicas, para tener una mayor cobertura de filtrado, ya que en ellas se puede usar una mayor gama de tráfico que la simulación no permite, al estar limitada a mensajes ICMP si no se recurre a programar otro tipo de tráfico.

En segundo lugar se encuentra POX, que es una herramienta muy versátil y que es recomendada para el desarrollo de componentes personalizados, como es el caso del componente que se creó en el capítulo tres del presente Proyecto de Titulación. La función del componente es la de leer parámetros de configuración de un archivo y enviar reglas basadas en estos parámetros a los conmutadores mediante mensajes de modificación de flujo. Además, se pudo constatar que este software es el que más documentación tiene en Internet y por ello su estudio y comprensión puede tomar menos tiempo que los otros controladores. Finalmente, el lenguaje Python es muy sencillo de entender, por lo que se puede manejar esta herramienta en poco tiempo leyendo la documentación incluida en el controlador.

En tercer lugar se encuentra Beacon, que presenta una mayor dificultad al momento de programar que su similar POX porque tiene estructuras más complejas y métodos que se deben analizar cuidadosamente. Por ejemplo, en la implementación del prototipo en el capítulo tres con el controlador Beacon se programaron ciertos parámetros adicionales que no fueron necesarios en POX, como por ejemplo w*ildcards,* que se encargan de definir que parámetros se toman en cuenta y que parámetros se ignoran al realizar una comparación entre un paquete de un flujo entrante y una regla de flujo (por ejemplo ignorar puerto origen de la comunicación entre un host A y un host B, pero si tener en cuenta el puerto destino), por lo que es necesario definir este parámetro para cada tipo de regla que se pretenda crear, y si no se define este parámetro, las reglas pueden llegar a ser inconsistentes. Esto, sin embargo; puede ser beneficioso para el administrador, ya que permite tener un mayor control sobre la estructura de comparación para las reglas de flujos; por ejemplo, el usuario puede definir parámetros y transmitirlos del controlador al equipo, pero no tomarlos en cuenta al momento de la comparación, simplemente modificando el valor de *wildcards.*

Finalmente, se encuentra Floodlight, que tiene una gran aceptación a nivel comercial debido a su gran facilidad de uso. Además, cuenta con una interfaz web bastante amigable, a través de la cual se pueden definir reglas de manera mucho más sencilla que el resto de controladores mediante el comando *curl,* como se constató en el capítulo tres, en donde se presentó el componente personalizado que envía reglas de flujo a cada conmutador, de manera similar a lo que se hizo con NOX, POX y Beacon.

De los resultados obtenidos durante la implementación del prototipo, se puede constatar que todos los componentes programados para cada controlador tienen sus ventajas y sus desventajas, siendo importante para una implementación definir que se va a hacer y como se lo va a hacer. En la implementación primero se definió qué hosts se podían comunicar entre sí y qué protocolo sería aceptado en esta comunicación. Además como requisito previo se definió que los mensajes ARP que llegaran al conmutador debían ser transparentes para el administrador para no causar confusión y que siempre se permita hacer peticiones ARP para que los protocolos de capa superior funcionen con normalidad. Se puede decir que la programación en Floodlight es bastante amigable con el usuario, para POX se necesita un mayor conocimiento y Beacon es el que requiere un mayor conocimiento de todos, aunque a su favor, le da al administrador un control más estricto que los anteriores.

- Durante la implementación de la SDN se modificó el *firmware* de los conmutadores Linksys WRT54GL, ya que fueron adaptados para que funcionen con el protocolo OpenFlow. Esto se logró mediante el *firmware* OpenWRT con soporte para OpenFlow. La actualización de *firmware* es un proceso en el que se debe tener mucho cuidado, porque existen ocasiones en las que la imagen se corrompe, por lo que se debe regresar a un modo de CLI básico mediante el cual se puede descargar una nueva imagen y realizar la actualización del *firmware.* Sin embargo, muchas veces esto no es posible, por lo que se debe desarmar el equipo y usar un cable JTAG para realizar este proceso. Estos equipos no son diseñados

para soportar este protocolo, solo han sido adaptados, por lo que el rendimiento frente a un equipo diseñado para este propósito es mucho menor y las funcionalidades son diferentes. Se menciona en la literatura específicamente el caso de un conmutador creado para OpenFlow, en el cual se debe definir una VLAN para la conexión con el controlador, como por ejemplo un conmutador HP con soporte de fábrica para este protocolo, situación que no es requerida en equipos adaptados para OpenFlow, como se pudo comprobar en el capítulo tres en donde se aplicó la configuración de la conexión con el servidor controlador pero nunca se especificó una VLAN. Esto puede causar confusión al momento de la implementación. Como los conmutadores son adaptados para el manejo del protocolo OpenFlow, se pueden presentar problemas de compatibilidad, como se mencionó anteriormente.

- En todas las pruebas llevadas a cabo en el presente Proyecto de Titulación se pudieron obtener prácticamente los mismos resultados para todos los controladores empleados cuando se implementaron componentes personalizados para el envío de reglas de flujos, la única diferencia radica en el lenguaje y paradigma de programación con la que los componentes fueron creados.

- Finalmente, se hicieron pruebas con Wireshark que permitieron comprender el intercambio de mensajes que se presentaron en el fundamento teórico, capturando los mensajes que se envían durante el establecimiento de la conexión OpenFlow y el envío de mensajes de modificación de flujos en la implementación del prototipo, con lo que se pudo constatar de manera práctica lo aseverado en el primer capítulo.

4.2 RECOMENDACIONES

- Para los administradores que deseen implementar una SDN existen varias alternativas. Se pueden usar los componentes que se encuentra incluidos por defecto en el controlador, o si se tiene el conocimiento suficiente en el

lenguaje de programación que usa el controlador, se pueden crear componentes personalizados para el envío de reglas de flujos hacia los conmutadores. Se recomienda que los administradores de redes que deseen crear su componente elijan un controlador programado en un lenguaje que ellos tengan amplio conocimiento, ya que así será más fácil la definición de flujos.

- Para evitar problemas de compatibilidad es necesario verificar que el software del servidor controlador sea compatible con los conmutadores elegidos para realizar la implementación. Esto es vital ya que caso contrario la SDN no podrá ser implementada. Esta recomendación surge de los problemas de incompatibilidad que se presentaron en el capítulo tres y cuyos síntomas son la desconexión inmediata del enlace establecido entre los conmutadores y el controlador NOX.

- Para redes pequeñas los controladores analizados presentan las mismas características básicas y su rendimiento ha sido dejado a un lado por el tamaño de la red y la cantidad de flujos definidos. Sin embargo, para implementaciones de mayor tamaño es recomendable analizar el rendimiento de cada conmutador en las condiciones en las que se implementará la red. Cabe recalcar que las SDN han sido diseñadas para ser implementadas en redes de gran tamaño, en donde existe un gran volumen de tráfico. En redes pequeñas esta arquitectura no tiene sentido, ya que se desperdiciaría recursos al usar un servidor que controle a uno o dos conmutadores, por lo que en este caso convendría usar un conmutador tradicional. Por ello, las redes SDN pequeñas son recomendadas sólo para implementar prototipos y no redes de producción.

- Como se pudo comprobar en la implementación del prototipo de SDN, este tipo de redes presentan ventajas sobre las redes tradicionales en redes de gran tamaño. Debido a este factor se sugiere que se introduzcan estos

conceptos en la enseñanza de redes y se presente esta tecnología como una alternativa para las redes tradicionales.

- Si se desea estudiar las SDN se recomienda empezar usando la herramienta Mininet, con la cual se trabajó a lo largo del capítulo dos. Mediante esta herramienta se puede simular redes SDN y es posible familiarizarse con conceptos y procedimientos propios de OpenFlow y de las SDN. Esta herramienta es muy práctica, ya que puede ser descargada como un *appliance* que incluye todo lo necesario para trabajar con SDN en un nivel básico. Como limitante no posee una interfaz gráfica, por lo que se sugiere previamente tener conocimientos básicos del sistema operativo Linux y de Python para la programación de *scripts* que permitirán la creación de topologías o componentes personalizados del controlador incluido en la máquina virtual.

- Para la definición de reglas en todos los componentes de los controladores, es imprescindible tener un buen conocimiento del modelo de referencia OSI, especialmente las capas dos, tres y cuatro que son las encargadas de la comunicación. Se hace especial énfasis en protocolos, direcciones lógicas y físicas y números de puerto para comunicaciones TCP y UDP. Además es necesario conocer y entender protocolos básicos como ARP. En el capítulo tres se definieron reglas para la comunicación de hosts con protocolos específicos basándose en el conocimiento de cómo estos protocolos realizan el intercambio de sus mensajes.

- Es recomendable que la empresa o institución en la que se va a implementar una SDN defina en primer lugar políticas de seguridad para posteriormente implementar de una manera más eficaz y segura las reglas que reflejen estas políticas. En la implementación del prototipo se definieron unas reglas a manera de muestra. Estas reglas pueden ampliarse y adaptarse según sea requerido.

- El computador o los computadores en los que se va a instalar la entidad controladora requieren características especiales, ya que van a ser equipos que van a estar en constante actividad durante todo el día. Así que, es recomendable que tengan buenos recursos en cuanto a memoria, procesamiento e interfaces de red, además de respaldo en elementos sensibles como ventiladores y fuentes de alimentación. Haciendo una analogía, este equipo viene a ser como los enrutadores o conmutadores de núcleo en una red tradicional. El hardware del o de los controladores usados requiere de mayores características, al tener varios controladores ejecutándose en diferentes máquinas virtuales.

- Por el motivo mencionado anteriormente es recomendable tener más de un servidor controlador trabajando conjuntamente, ya que este equipo es crítico al ser el punto central de la red. Por ello se alienta al administrador a usar más de uno de estos dispositivos y usar un software como FlowVisor para gestionar el manejo de servidores y poder administrar cuando un determinado servidor entra en funcionamiento. Esto también se lo puede hacer sin FlowVisor, pero se debe tener en cuenta que las reglas transmitidas por cada controlador no deben ser contradictorias para que todo funcione correctamente.

- El soporte para este tipo de redes y la documentación son escasos, ya que existe poca literatura pero existen amplias referencias en Internet, por lo que se sugiere que si se va a administrar una red de este tipo se tenga el conocimiento suficiente para responder a cualquier falla o eventualidad. Se alienta a que se compile todo el conocimiento necesario para entender una SDN en un libro, que sería de gran utilidad para futuros usos e implementaciones de este tipo de red.

- Con este prototipo de SDN, se tiene una base para realizar otras implementaciones, ya que el tema de SDN es bastante amplio. Se alienta a los administradores a adaptar aplicaciones de las redes tradicionales a este tipo de redes, como por ejemplo, efectuar traducciones NAT o

implementar *firewalls* para el manejo de la seguridad en una SDN. Las aplicaciones que se pueden realizar son muy amplias, ya que estas redes son muy versátiles.

REFERENCIAS BIBLIOGRÁFICAS

[1] Opennetworking. /www.opennetworking.org/images/stories/downloads/white-papers/wp-sdn-newnorm.pdf (Consultado el 6 de Noviembre de 2012)

[2] OpenFlow. http://www.openflow.org (Consultado el 6 de Noviembre de 2012)

[3] NOX. http://www.noxrepo.org/nox (Consultado el 9 de Noviembre de 2012)

[4] Web Service. http://en.wikipedia.org/wiki/Web_service (Consultado el 15 de Noviembre de 2012)

[5] POX. http://www.noxrepo.org/pox (Consultado el 15 de Noviembre de 2012)

[6] OpenFlow Stanford. https://openflow.stanford.edu (Consultado el 14 de Noviembre de 2012)

[7] Mininet. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet (Consultado el 23 de Noviembre de 2012)

[8] DREIER, Gustavo, Tutorial Mininet, Documento Electrónico http://www.cedia.org.ec/images/stories/documentos_descarga/CursoOpenFlowDeRedClara/ 06a-mininet.pdf (Consultado el 3 de Diciembre de 2012)

[9] DREIER, Gustavo, Introducción SDN y OpenFlow, Documento Electrónico http://www.cedia.edu.ec/images/stories/documentos_descarga/CursoOpenFlowDeRedClara/ 01-introduccion%20sdn.pdf (Consultado el 3 de Diciembre de 2012)

[10] DREIER, Controladores para OpenFlow, Documento Electrónico http://www.cedia.org.ec/images/stories/documentos_descarga/CursoOpenFlowDeRedClara/ 05a-controladores.pdf (Consultado el 3 de Diciembre de 2012)

[11] FlowVisor. https://openflow.stanford.edu/display/DOCS/Flowvisor (Consultado el 8 de Diciembre de 2012)

[12] Floodlight. http://www.projectfloodlight.org (Consultado el 8 de Marzo de 2013)

[13] Documentación de Floodlight. http://docs.projectfloodlight.org /display/floodlightcontroller (Consultado el 8 de Marzo de 2013)

[14] Especificación OpenFlow. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf (Consultado el 8 de Marzo de 2013)

[15] ANÓNIMO, Tutorial, Documento Electrónico http://www.cs.princeton.edu/courses/archive/fall10/cos561/assignments/ps2-tut.pdf (Consultado el 12 de Marzo de 2013)

[16] VirtualBox. https://www.virtualbox.org/ (Consultado el 15 de Marzo de 2013)

[17] Xming. http://www.straightrunning.com/XmingNotes/ (Consultado el 15 de Marzo de 2013)

[18] ANÓNIMO, Manual de referencia Red Hat Linux 4, Capítulo 20: Protocolo SSH, Documento Electrónico http://www.gb.nrao.edu/pubcomputing/redhatELWS4/RH-DOCS/rhel-rg-es-4/ch- ssh.html (Consultado el 15 de Marzo de 2013)

[19] PuTTY. http://www.putty.org/ (Consultado el 15 de Marzo de 2013)

[20] ANÓNIMO, HP Networking OpenFlow Workshop, Documento Electrónico http://www.cedia.org.ec/images/stories/documentos_descarga/CursoOpenFlowDeRedClara/05e-dpctl.pdf (Consultado el 2 de Abril de 2013)

[21] Tipo de topología. http://www.gobiernodecanarias.org/educacion/

Conocernos_mejor/paginas/topologia.html (Consultado el 10 de Abril de 2013)

[22] Manual del comando dpctl en Linux.

[23] Mininet. http://mininet.org (Consultado el 2 de Abril de 2013)

[24] VMware vSphere Hypervisor. http://www.vmware.com/products/vsphere-hypervisor/overview.html (Consultado el 5 de Mayo de 2013)

[25] Linksys. http://support.linksys.com/en-us/support/routers/WRT54GL (Consultado el 20 de Mayo de 2013)

[26] OpenWRT. https://openwrt.org/ (Consultado el 20 de Mayo de 2013)

[27] OpenFlow OpenWRT. http://www.openflow.org/wk/index.php/OpenFlow_1.0_for_OpenWRT (Consultado el 20 de Mayo de 2013)

[28] http://mytestbed.net/projects/openflow/wiki/Installing_a_NOX_controller (Consultado el 3 de Junio de 2013)

[29] http://networkstatic.net/nox-openflow-controller-install-on-ubuntu-12-04-precise-lts/ (Consultado el 3 de Junio de 2013)

[30] Tutorial OpenFlow. http://www.openflow.org/wk/index.php/OpenFlow_Tutorial (Consultado el 17 de Junio de 2013)

[31] Guía del comando Curl. http://curl.haxx.se (Consultado el 19 de Junio de 2013)

[32] https://mailman.stanford.edu/pipermail/openflow-discuss/2011-July/002483.html (Consultado el 22 de Junio de 2013)

[33] https://mailman.stanford.edu/pipermail/openflow-discuss/2010-December/001704.html (Consultado el 22 de Junio de 2013)