Escalabilidad de Redes Definidas por Software en la Red Académica

Proyecto de grado

Santiago Vidal

Supervisores:
Dr. Eduardo Grampín
MSc. Martín Giachino

Instituto de Computación

Facultad de Ingeniería, Universidad de la República

Junio 2016

Tabla de Contenidos

Tabla de Figuras

Tabla de Cuadros

Capítulo 1

Introducción

La Red Académica Uruguaya (RAU) es un emprendimiento de la Universidad de la República que ya lleva 28 años, y es administrado por el Servicio Central de Informática Universitario (SeCIU). Al igual que las redes académicas de muchos otros países, tiene como objetivo proveer a docentes e investigadores una infraestructura aislada de las redes comerciales, que les ofrezca altas velocidades para diversas aplicaciones de enseñanza e investigación. La RAU actualmente consta de 64 nodos y brinda servicios a 31 instituciones, abarcando un total aproximado de 8.000 docentes, 4.000 técnicos y 100.000 estudiantes[1].

En la actualidad se está trabajando en renovar la RAU con una nueva infraestructura que permita proveer más y mejores servicios. Este esfuerzo se denominó RAU2. Los principales objetivos de esta renovación son: 1) extender y mejorar la calidad del servicio, y 2) implementar servicios e infraestructuras que maximicen la sinergia entre las instituciones. Para esto, se plantean tres áreas principales de trabajo: re-diseñar la topología de la red y aumentar el ancho de banda, mejorar la controlabilidad de la red, y desarrollar una plataforma integrada y global para los servicios académicos.

En 2015 culminó un proyecto llamado Routers Reconfigurables de Altas Prestaciones (RRAP) [43] que consistió en la construcción de un prototipo para la RAU2. Este prototipo propone una arquitectura de red denominada RAUFlow, y está basada en un concepto llamado Redes Definidas por Software (SDN por sus siglas en inglés). SDN es un paradigma de red que ha tomado fuerza en los últimos años, que plantea desacoplar el plano de control (donde se determina qué hacer con los paquetes de datos) del plano de datos (donde se toman las acciones sobre los paquetes). Propone una infraestructura enfocada en otorgar, de forma más económica y eficiente, un mayor control a los operadores de red para adaptar y optimizar sus redes para los servicios y capacidades que necesitan proveer.

[1] Cifras obtenidas del informe del proyecto RRAP

A pesar de la fuerza que han ganado las redes definidas por software en el último tiempo, están lejos de ser el estándar, y las redes legadas aún son muy utilizadas en la actualidad. Por esta razón es necesario estudiar soluciones que aprovechen las ventajas que ofrece SDN, y al mismo tiempo puedan coexistir con las redes legadas y beneficiarse de las mismas. RAUFlow es una propuesta que sigue ese mismo principio de redes híbridas, y es la base sobre la que se construye este trabajo.

1.1 Proyecto RRAP

El prototipo para la RAU2 construido por el proyecto RRAP [43] está compuesto 4 routers y una arquitectura para el control de la red basada en SDN, llamada RAUFlow. Cada dispositivo se denomina RAUSwitch y tiene la capacidad de funcionar en modo SDN y en modo tradicional como router IP. Los aspectos técnicos, tanto de RAUFlow como de RAUSwitch, se analizarán más adelante. El prototipo fue verificado con una serie de pruebas funcionales que comprueban la validez del enfoque y que se proveen correctamente los servicios de red que fueron implementados. Sin embargo, debido a las limitaciones físicas del prototipo, el enfoque propuesto aún no ha sido validado con pruebas de escala que aseguren que el mismo podría ser adoptado para una infraestructura con una dimensión y carga como la que tiene la RAU.

1.2 Objetivos

El objetivo principal de este trabajo es estudiar la escalabilidad de la arquitectura RAUFlow mediante simulaciones con grandes cantidades de nodos. Dada la ausencia de trabajos relacionados a la virtualización de redes híbridas que soporten SDN y los protocolos IP tradicionales, previamente es necesario un estudio a fondo de las tecnologías de virtualización disponibles y un posterior trabajo de construcción de la plataforma deseada.

1.3 Resultados esperados

Se espera que este trabajo produzca los siguientes resultados:

- El estado del arte en lo que refiere a la implementación de aplicaciones en SDN, así como las herramientas de virtualización disponibles, prestando especial atención a aquellas que pueden ser aplicables al enfoque híbrido SDN/Legacy y al mismo tiempo sean escalables.

- Una herramienta que permita virtualizar la arquitectura RAUFlow, y que sea razonablemente escalable. Es importante considerar dicha herramienta como un resultado que pueda mantenerse útil incluso afuera del contexto de RAUFlow, como herramienta de investigación autocontenida.
- Diseñar y llevar a cabo una serie de pruebas de escala sobre RAUFlow que detecten posibles errores, y permitan hacer un análisis sobre su escalabilidad.

1.4 Estructura del documento

La estructura de lo que resta de este documento se explica a continuación. En el capítulo 2 se muestran los resultados de la investigación del estado del arte. En el mismo se explican algunos conceptos claves para entender este trabajo, entre ellos los aspectos técnicos de RAUFlow y RAUSwitch. Luego se estudia lo investigado con respecto a las aplicaciones de SDN y las herramientas de virtualización disponibles. En el capítulo 3 se presentan los requerimientos, diseño, implementación y problemas encontrados para el entorno virtual construido. En el capítulo 4 se presentan las pruebas de escala realizadas y un análisis de los resultados que arrojan. Por último, en el capítulo 5 se presentan las conclusiones de la realización de este trabajo y posibles líneas de trabajo futuro.

Además, se agrega un capítulo con la bibliografía utilizada y un apéndice con el manual de usuario para el entorno virtual.

Capítulo 2

Estado del arte

En este capítulo se presentan los resultados de la investigación del estado del arte. La sección 2.1 está dedicada a introducir conceptos previos que son claves para entender el trabajo. La sección 2.2 estudia las diferentes aplicaciones que se le puede dar a las redes definidas por software, haciendo hincapié en la implementación de redes privadas virtuales. Por último, la sección 2.3 hace un estudio de las diferentes herramientas que se pueden utilizar para virtualizar SDN, y en particular, la arquitectura RAUFlow.

2.1 Conceptos previos

A continuación se explican algunos conceptos que son fundamentales, ya que son sobre los que se basa el proyecto RRAP [43] y este trabajo. Además, se hace un resumen de los aspectos técnicos de RAUFlow y RAUSwitch.

2.1.1 Software Defined Networking

Las Redes Definidas por Software (o Software Defined Networking en inglés) es una arquitectura de red emergente que propone separar el plano de control del plano de datos. Esto significa que toda la inteligencia de la red, que en las redes tradicionales se encuentra en cada dispositivo de red, se extrae de los mismos y se pasa a ubicar en una entidad central.

La arquitectura de SDN está compuesta por tres capas, y se puede ver en la figura 2.1. La inferior es la capa de infraestructura (Infrastructure Layer), que contiene los dispositivos de red como routers o switches. En el paradigma SDN, la única inteligencia que poseen estos dispositivos es la necesaria para llevar a cabo las tareas que sean indicadas por el plano de control. Por encima se encuentra la capa de control (Control Layer). Aquí se encuentra el controlador de la red, quien se encarga de comunicarse con los dispositivos y

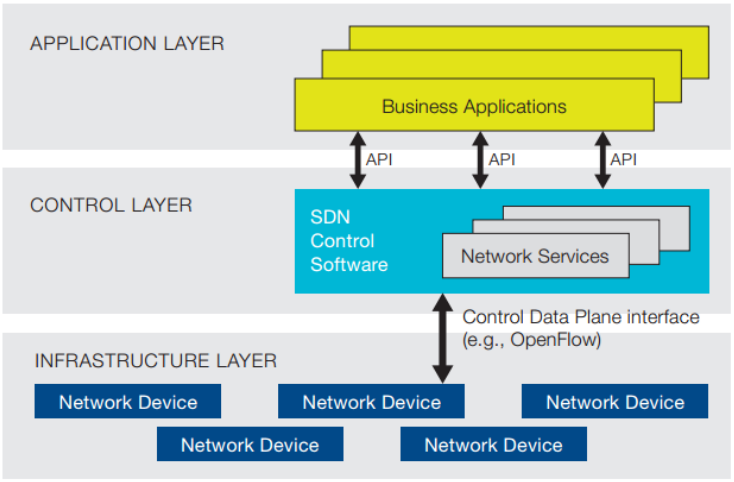


Fig. 2.1 Capas de la arquitectura de SDN. Imagen extraída de [28]

hacer disponibles un conjunto de servicios básicos a las aplicaciones, además de proveer una visión abstracta de la red. La capa superior es la de aplicación (Application Layer). En ella se encuentran las aplicaciones de negocio que escriben los administradores de red para implementar los requerimientos que necesitan de sus redes, como protocolos de ruteo, ingeniería de tráfico, mecanismos de seguridad, etc.

La comunicación entre las capas se da mediante interfaces (o APIs). La comunicación entre la capa de infraestructura y la de control se implementa con la llamada **interfaz sur**. Por otro lado, la **interfaz norte** especifica como las aplicaciones de negocio deben utilizar la visión abstracta de la red y los servicios provistos por el controlador.

2.1.2 OpenFlow

OpenFlow [51] es un estándar desarrollado por la Universidad de Stanford en 2008, y desde entonces es mantenido por la ONF (Open Networking Foundation). Es una implementación de la Interfaz Sur de la arquitectura de SDN, es decir, implementa la comunicación entre la capa de control y la capa de infraestructura (los dispositivos de red). Existen algunas diferencias entre las distintas versiones de OpenFlow. A pesar de que la última versión de OpenFlow al momento de realizar este trabajo es la 1.5, se explicará la versión 1.3.1 ya que es la utilizada por RAUFlow.

La estructura de un switch OpenFlow se puede ver en la figura 2.2. Se pueden identificar tres componentes principales: a) un conjunto de tablas de flujos, b) un canal seguro de comunicación entre el switch y el controlador para intercambiar comandos y paquetes, c) el protocolo OpenFlow, que define la comunicación entre el switch y el controlador.

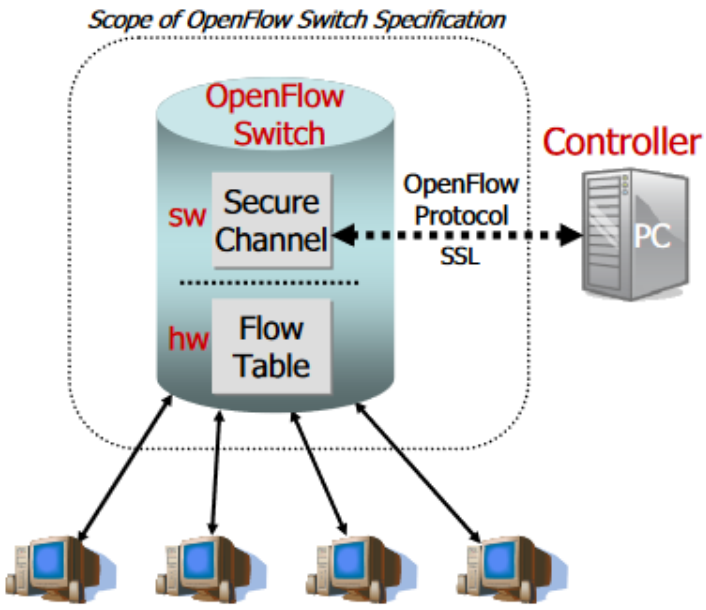


Fig. 2.2 Estructura de un switch OpenFlow. Imagen extraída de [51]

Table 2.1 Estructura de un flujo OpenFlow 1.3.1 [21]

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Las capacidades de un switch OpenFlow se abstraen mediante un concepto denominado **flujo**. Un flujo es una instrucción que le indica al switch qué acciones debe tomar cuando recibe un paquete de un determinado tipo. La tabla 2.1 muestra cómo está compuesto un flujo OpenFlow. Los campos son los siguientes:

- **Match Fields**. Este campo indica a qué tipo de tráfico aplica el flujo. Esto lo logra especificando valores para los cabezales de los paquetes. Como lo muestra la figura 2.3, OpenFlow 1.3.1 puede trabajar con los cabezales de protocolos de las capas 1-4 del modelo OSI. Esta capacidad le permite a un switch OpenFlow "agrupar" el tráfico en grupos (o flujos), y tomar las mismas acciones para tráfico similar.

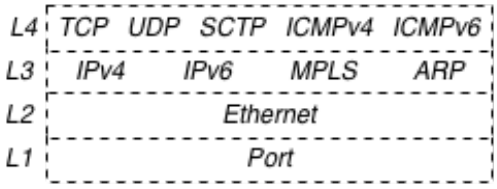


Fig. 2.3 Match fields de OpenFlow 1.3.1. Imagen extraída de [43]

- **Priority**. Indica la prioridad del flujo.
- **Instructions**. Indica las acciones que se deben tomar para los paquetes que le correspondan al flujo. Algunos ejemplos de acciones son:
  - **Output:port** permite reenviar el paquete por uno de los puertos del switch.
  - **Output:controller** permite enviarle el paquete al controlador para que él decida que acciones tomar. El controlador luego puede elegir descartar el paquete o instalar un nuevo flujo para contemplar ese tipo de tráfico.
  - **Drop** descarta el paquete.
  - **Push-Tag/Pop-Tag** permite agregar o quitar etiquetas MPLS o VLAN.
- **Timeouts**. Tiempo que debe pasar para que el switch elimine al flujo por inactividad.
- **Counters**, **Cookie**. Son campos auxiliares. Si el lector desea, puede leer sobre ellos en [21].

Cuando un paquete llega a un switch OpenFlow, el switch analiza los cabezales del mismo, y los compara con los Match Fields de los flujos que tiene almacenados, y así determina si el paquete pertenece a alguno de ellos. En caso positivo, realiza las acciones que indique el flujo correspondiente. En caso negativo, se aplican las acciones de un flujo especial llamado "Table-miss Flow Entry". Este flujo es el que se utiliza para el tráfico que no se corresponde con ninguno de los otros flujos de un switch, y en general descarta el paquete o lo reenvía al controlador.

OpenFlow 1.3.1 permite utilizar múltiples tablas de flujos. Gracias a una instrucción especial llamada "GoTo Table" un flujo puede indicar que el paquete pase por otra tabla. Cuando el paquete termina de recorrer ese pipeline de tablas, se ejecuta el conjunto de instrucciones que se obtuvo de cada una. La figura 2.4 muestra como funciona el pipeline de OpenFlow.

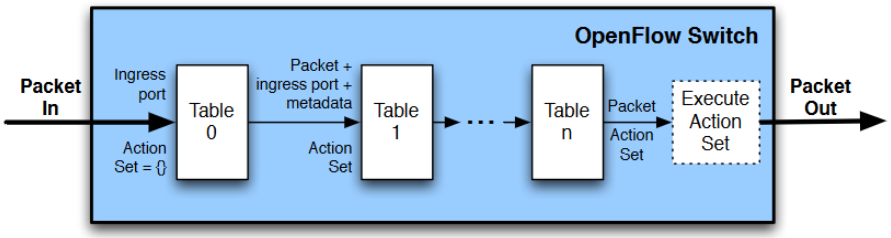


Fig. 2.4 Pipeline de procesado de paquetes de OpenFlow. Imagen extraída de [21]

2.1.3 Open vSwitch

Open vSwitch [Ben Pfaff] es una implementación en software de un switch virtual multicapa. Es open-source y está disponible para múltiples sistemas operativos, pero está mayoritariamente enfocado hacia Linux. Open vSwitch fue diseñado para actuar como switch virtual entre máquinas virtuales y para brindarles conectividad con la red física.

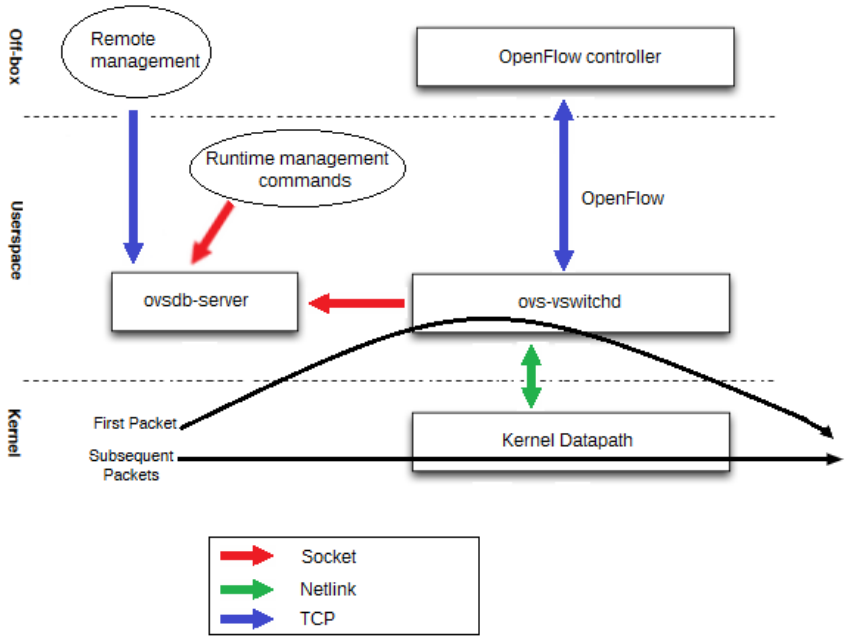


Fig. 2.5 Arquitectura de Open vSwitch.

Debido a que soporta OpenFlow, Open vSwitch es comúnmente utilizado como switch en despliegues SDN. La figura 2.5 muestra la arquitectura de Open vSwitch, y cómo ella interactúa con un controlador OpenFlow. Se identifican tres componentes principales:

- **ovsdb-server** [24]. Es un servidor de base de datos que almacena toda la información relacionada con los switches. Este servidor se puede usar tanto para hacer cambios en configuraciones como para hacer consultas. La interacción con este proceso se hace mediante un protocolo llamado OVSDB (RFC 7047 [20]). Esta comunicación se puede llevar a cabo de forma local y remota. Cuando se desea utilizar ovsdb-server localmente, desde el sistema operativo anfitrión, la comunicación se implementa con un archivo socket de Unix. Para gestión remota en general se utiliza el puerto TCP 6640.

- **ovs-vswitchd** [23]. Es un demonio que ejecuta en el userspace y se encarga de manejar y controlar los switches virtuales. Este demonio está pendiente y se actualiza automáticamente ante cambios de configuración efectuados en ovsdb-server. Para poder comunicarse con él, en general utiliza un Unix socket. Este demonio es quien se conecta con el controlador OpenFlow, e interactúa con el mismo de acuerdo al funcionamiento normal de OpenFlow.

- **Kernel Datapath**. Esto es un módulo que ejecuta en el kernel creado para aumentar la velocidad de los switches. Mantiene una caché de flujos para que sólo algunos paquetes deban ser resueltos en el userspace. Una vez un flujo está en el kernel datapath, ya no hay necesidad de que ovs-vswitchd resuelva qué hacer con el paquete, y se puede resolver a nivel del kernel, lo cual aumenta significativamente la velocidad de procesamiento. La comunicación de este módulo con el demonio ovs-vswitchd se implementa con el protocolo Netlink [25].

2.1.4 Multiprotocol Label Switching (MPLS)

Multiprotocol Label Switching (MPLS) es un mecanismo de transporte de datos que utiliza conmutación de etiquetas para reenviar paquetes a través de una red. MPLS trabaja entre las capas de Red y Enlace del modelo OSI, y puede encapsular paquetes de varios protocolos (de ahí surge el nombre "multiprotocol").

Cuando un paquete ingresa a una red MPLS, es recibido por un router llamado Label Edge Router (LER), es decir, un router de borde. El LER le agrega al paquete un cabezal MPLS y lo reenvía. Cada nodo interno de la red (llamado Label Switching Router) que recibe el paquete le extrae el cabezal y lo analiza, y basándose en el valor de la etiqueta decide cuál es el próximo nodo al que debe reenviarse. Luego le asigna un nuevo cabezal con la nueva etiqueta y lo reenvía. Este proceso se repite hasta que el paquete llega a un LER de salida, donde sale de la red sin etiquetas.

Un cabezal MPLS está compuesto por cuatro campos: Label, TC (Traffic Class), S (bottom-of-stack) y TTL (time-to-live). Label es el campo principal, y su valor decide el camino que toma el paquete. El campo TC es utilizado para implementar Quality of Service. TTL indica el tiempo de vida (en saltos) que le queda al paquete. MPLS permite utilizar múltiples etiquetas solapadas en una pila o stack. El campo S indica si la etiqueta es la última en el stack o no.

Algunos términos relacionados a MPLS son:

- Forwarding Equivalence Class (FEC). Clase de equivalencia que agrupa a los paquetes "similares" y que pueden ser reenviados por la misma ruta, es decir, pueden ser etiquetados con la misma etiqueta MPLS.
- Label Switched Path (LSP). Ruta sobre una red MPLS. En general es establecida por un protocolo de distribución de etiquetas como LDP, y se establece para encaminar tráfico de una más FECs.
- Label Distribution Protocol (LDP). Protocolo de distribución de etiquetas MPLS entre los nodos de una red.

2.1.5 Red Privada Virtual (VPN)

Una Red Privada Virtual (VPN por sus siglas en inglés) es una red privada que se extiende a través de una red pública, como lo es Internet. Permite conectar redes ubicadas en distintos lugares geográficos de modo transparente, es decir, como si estuvieran directamente conectadas.

Los servicios de VPNs en general proveen autenticación, integridad y confidencialidad de los datos, y en ocasiones ofrecen funcionalidades avanzadas como priorizar tipos de tráfico y Quality of Service (QoS).

Una VPN puede ser punto a punto o multipunto. La primera se utiliza para conectar sólo dos extremos de una organización, mientras que la segunda se utiliza para brindar conectividad a una organización dispersa en múltiples lugares.

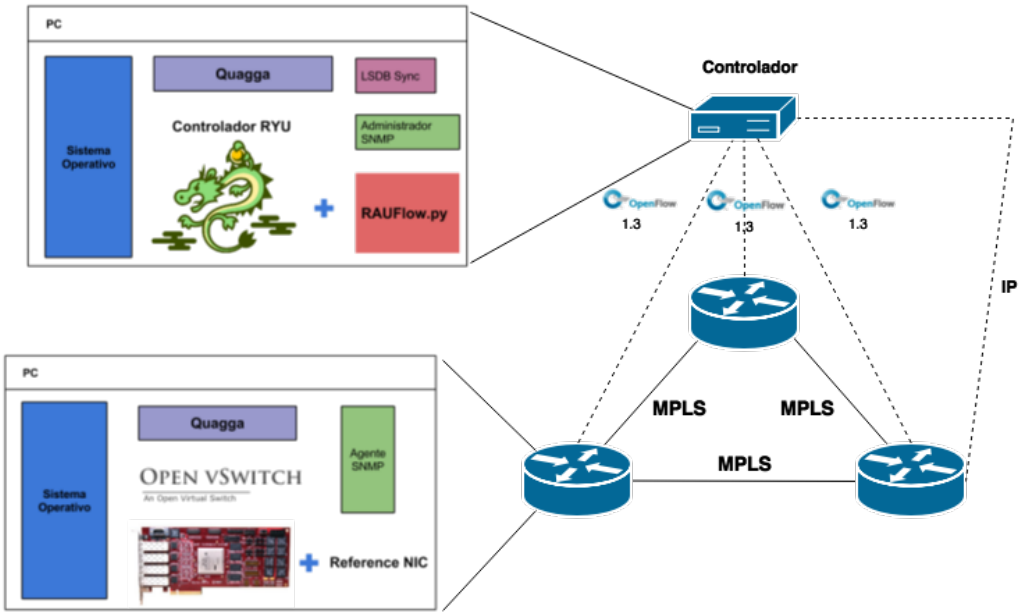
Otra forma de clasificar una VPN es de acuerdo a la capa en la que opera. Una VPN de capa 3 en general conecta subredes IP distintas, y el tráfico que pasa a través de ella consiste de paquetes IP. Por otro lado, una VPN de capa 2 permite que dos o más redes dispersas geográficamente compartan un dominio de difusión Ethernet, y actúen como si estuvieran en la misma LAN. Esta última comúnmente recibe el nombre de Virtual Private LAN Service (VPLS).

2.1.6 RAUFlow

El propósito de esta sección es resumir y explicar los aspectos técnicos de RAUFlow, ya que es la base sobre la que se construye el presente trabajo. Si el lector desea profundizar en este tema se recomienda la lectura del capítulo 5 del informe del Proyecto RRAP [43].

RAUFlow es una aplicación para control de red basada en el controlador SDN Ryu [4], que combina OpenFlow y MPLS para implementar servicios de VPN. El plano de control de la red no depende únicamente de la aplicación que ejecuta en el controlador, ya que existen

Fig. 2.6 Visión general de la arquitectura de RAUFlow. Imagen extraída de [43]



más componentes. Por lo tanto, el término RAUFlow también se utiliza para referirse a la arquitectura global de la solución.

RAUFlow utiliza OpenFlow y MPLS para conectar dos extremos de la red con una VPN. Además, utiliza herramientas adicionales como Quagga y SNMP para que el controlador tenga información de la topología y de los dispositivos. La figura 2.6 muestra la arquitectura global de RAUFlow. A continuación se explican sus principales características:

- **RAUSwitch** es el nombre que llevan los dispositivos de red. Se componen de la siguiente manera:
  - **Open vSwitch**. Se utiliza como implementación (en software) de OpenFlow. Es lo que permite que los dispositivos se comporten como switches OpenFlow.
  - **Quagga**. Este software de ruteo se utiliza para que los RAUSwitch puedan utilizar el protocolo de ruteo OSPF. El objetivo de utilizar este protocolo se explicará mas adelante, cuando se explique el módulo LSDB Sync del controlador. Esta característica hace que se refiera al RAUSwitch como un switch híbrido, ya que se comporta como switch OpenFlow y al mismo tiempo como router tradicional.
  - **Agente SNMP**. El controlador necesita conocer las direcciones IP de las interfaces de los RAUSwitch, pero OpenFlow no provee una manera de comunicarle esto al controlador, ya que está orientado a esquemas SDN puros donde los switches OpenFlow no tienen direcciones IP. Por lo tanto, se utiliza un agente

SNMP en cada RAUSwitch, el cual es consultado desde un administrador SNMP en el controlador para obtener las direcciones IP de cada RAUSwitch.

- **Controlador**. Esta entidad es la que ejecuta la aplicación RAUFlow y se comunica con todos los componentes de la arquitectura para implementar el plano de control. Se compone de la siguiente manera:
  - **Controlador Ryu + aplicación RAUFlow**. Aquí es donde se encuentra la mayoría de la inteligencia de la red. Provee una interfaz gráfica Web para que el administrador de la red cree, modifique y elimine VPNs. Cuando se crea una nueva VPN, se encarga de computar el camino óptimo entre un par de nodos utilizando una versión modificada del algoritmo Dijkstra. Luego de computar el camino para la VPN, instala los flujos OpenFlow necesarios en los RAUSwitch involucrados para que los mismos puedan brindar conectividad a la VPN mediante conmutación de etiquetas MPLS.
  - **LSDB Sync** se encarga de tomar la información de la base de datos topológica de OSPF, procesarla y enviarla a la aplicación RAUFlow. Esta componente se encarga de detectar cambios en la topología (escuchando los mensajes del protocolo OSPF) y cuando la base de información topológica local del controlador está actualizada, la procesa y luego la envía a la aplicación en el controlador. RAUFlow toma esta información y se actualiza con la nueva topología, y luego utiliza el módulo Administrador SNMP para obtener información adicional sobre los dispositivos (como las direcciones IP de cada puerto).
  - **Quagga**. El controlador ejecuta una instancia de Quagga, obteniendo de esta forma acceso local a la información de la base de datos topológica construida por OSPF (Link-State-Database).
  - **Administrador SNMP**. El administrador SNMP es utilizado para consultar al agente SNMP instalado en cada nodo, para así obtener información adicional que no puede ser obtenida por el protocolo OpenFlow, como la correspondencia entre números de puerto OpenFlow y direcciones IP. Este componente es utilizado por RAUFlow cada vez que se actualiza la topología, y es necesario obtener la información de cada RAUSwitch.

En RAUFlow, un servicio se refiere a una clase de tráfico a la que se brindará conectividad. Para crear un servicio desde la interfaz Web, es necesario indicar las siguientes propiedades:

- Nodo e interfaz de entrada.

- Nodo e interfaz de salida.
- Tipo de servicio: si es de capa 2 o 3.
- Ethertype. Indica el protocolo al que pertenece el tráfico del servicio. Es necesario indicar esto sólo si se trata de un servicio de capa 3.
- Opcionalmente, se puede definir el servicio de forma más específica, utilizando puertos TCP o UDP, direcciones IP, direcciones IPv6, etiquetas VLAN, direcciones MAC, y muchos otros.

Definiendo dos servicios iguales pero en sentidos inversos (nodos opuestos de entrada y salida) efectivamente se está definiendo una VPN.

2.2 Aplicaciones de SDN

El paradigma SDN, por definición, está basado en el software. Esto implica que el rango de aplicaciones o modos de uso que se le puede dar al paradigma es enorme. En esta sección se estudiarán algunas aplicaciones existentes sobre SDN, haciendo foco en la implementación de redes privadas virtuales, debido a que es el servicio que provee RAUFlow.

2.2.1 Implementación de VPNs

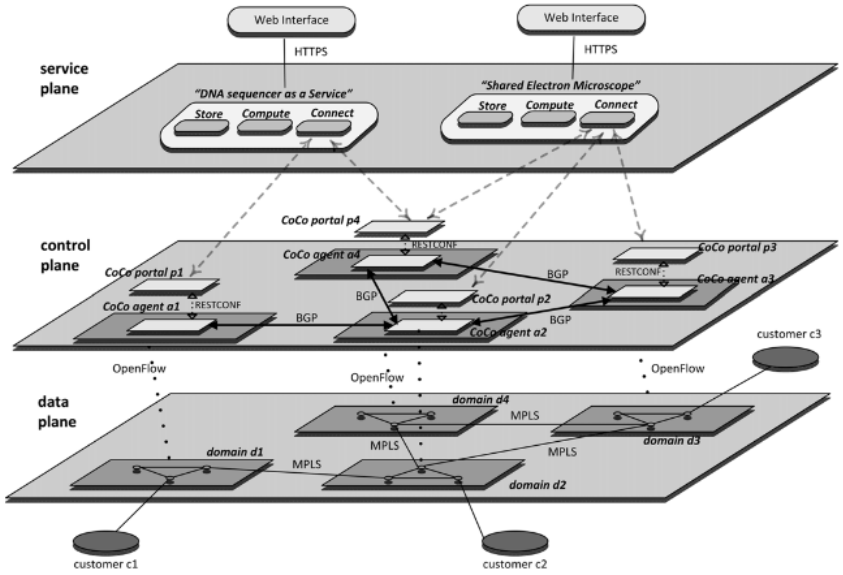
El desarrollo de redes privadas virtuales (VPN) es un tema de investigación relativamente común en SDN, ya que es un servicio de gran demanda en el mundo actual. A continuación se estudiarán algunas implementaciones existentes de VPN sobre SDN.

Proyecto CoCo

CoCo [56] es un proyecto que estudia el desarrollo de un servicio de VPN multipunto enfocado hacia investigadores, para permitirles intercambiar información de modo confiable y seguro. Uno de sus principales enfoques es la facilidad de uso, lo cual es una ventaja frente a las implementaciones tradicionales (no SDN) de este servicio. Sólo requiere que el administrador haga una configuración inicial de la red, y luego los usuarios finales pueden crear, modificar y eliminar VPNs a demanda mediante un portal web.

La arquitectura de CoCo, que se ilustra en la figura 2.7, está diseñada para aplicarse a muchos dominios. Cada dominio es una red OpenFlow administrada por un controlador OpenDaylight, llamado agente CoCo. Este último tiene dos grandes tareas. La primera es

Fig. 2.7 Arquitectura de CoCo. Imagen extraída de [56]



controlar los switches OpenFlow de su dominio, haciendo descubrimiento de la topología y configurando las reglas de reenvío de los switches. La segunda tarea es en el plano de control de la arquitectura, y consiste en utilizar el protocolo BGP para intercambiar información de accesibilidad con otros agentes CoCo (por ende, otros dominios).

Como ya se mencionó, la red de cada dominio está compuesta por switches OpenFlow. Éstos últimos pueden ser internos o de borde, también llamados Provider Edge (o PE). Los de borde son los que se conectan con otros dominios o con las redes cliente (Customer Edge). Las reglas de reenvío están basadas en MPLS, y se utilizan dos niveles de etiquetas. La etiqueta exterior sirve para identificar al switch PE al cual se debe enviar un paquete. La etiqueta interior identifica a la VPN a la cual pertenece el paquete. Esto quiere decir que el tráfico que se recibe por la red CE es etiquetado apropiadamente por el switch PE que recibe dicho tráfico. Cuando el tráfico sale de la red interna hacia la red CE, el switch PE se encarga de remover las etiquetas MPLS del tráfico saliente.

OpenFlow para mejorar la escalabilidad de un servicio IP-VPN

En [59] se estudia el problema de la escalabilidad al proveer un servicio de IP-VPN, y propone una solución basada en SDN y OpenFlow. Si un proveedor de servicios de telecomunicaciones ofrece un servicio de IP-VPN, es posible que lo haga utilizando el protocolo BGP. Dicho protocolo se utiliza para intercambiar la información de ruteo correspondiente

a cada red cliente conectada al servicio. En un esquema no SDN, el procesamiento del plano de control (en este caso BGP) queda a cargo de los routers de la red. Esto presenta un posible problema de escalabilidad. Cuando se agregan nuevos clientes a la VPN, la nueva información de ruteo debe ser propagada. Los recursos de CPU y memoria de los routers serán consumidos de acuerdo a lo que exijan los nuevos clientes. Por lo tanto, es necesario confirmar que hay margen en la capacidad de los dispositivos antes de aceptar nuevos clientes.

La solución que se propone para atacar este problema es utilizar OpenFlow. Se despliega un controlador OpenFlow que además ejecuta múltiples demonios BGP, uno por cada cliente. Los demonios BGP se encargan del intercambio de información de ruteo con los routers de las redes cliente. Un cliente puede tener múltiples redes conectadas a la VPN. Cuando el controlador recibe información de ruteo correspondiente a una determinada red, el demonio BGP encargado de ese cliente notifica a las demás redes del mismo cliente. Además, la misma información de ruteo recibida es procesada para generar las reglas de forwarding (entradas de flujos en los switches OpenFlow) que aseguren la conectividad. Dado que distintos clientes pueden compartir direcciones IP, es necesario que la red OpenFlow pueda distinguir de algún modo a qué cliente pertenece el tráfico. Esto lo logra asignando etiquetas de VLAN a los paquetes cuando entran a la red, que identifican a cada cliente. En el trabajo se menciona, como objetivo futuro, utilizar MPLS en lugar de etiquetas de VLAN.

Con la utilización de OpenFlow se resuelve en gran medida el problema de la escalabilidad, ya que se saca el procesamiento del protocolo BGP de los routers y se concentra todo en el controlador. Si bien agregar nuevos clientes a la VPN aumentará la carga de procesamiento y memoria, el controlador puede manejarlo más fácilmente que los dispositivos de red, que tienen recursos mucho más limitados.

SDxVPN

SDxVPN [36] es una propuesta de implementación de VPNs de capa 2 y 3, basadas en MPLS y OpenFlow. Intenta atacar tres problemáticas comunes que experimentan los proveedores de servicios al ofrecer VPNs: 1) complejidad en la administración de las VPNs, 2) los dispositivos de red deben implementar un plano de control muy complejo, y eso los encarece, y 3) ejecutar el plano de control en los dispositivos puede ser muy costoso para el rendimiento cuando aumenta la cantidad de clientes, y esto presenta un problema de escalabilidad.

El diseño de la solución propuesta se puede ver en la figura 2.8. En la misma se observa la red de un hipotético proveedor de servicios conectada a dos clientes, cada uno con

Fig. 2.8 Diseño de SDxVPN. Imagen extraída de [36]

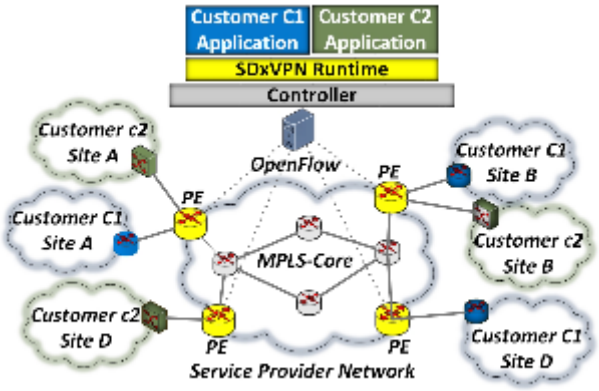
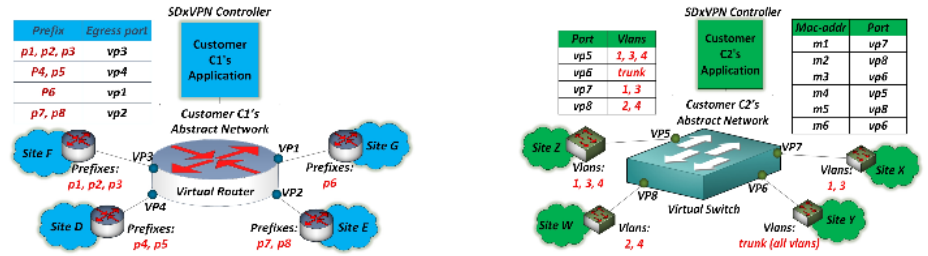


Fig. 2.9 Modo de operación de cada servicio de acuerdo a su capa. Imagen extraída de [36]



a) Router virtual implementando una VPN de capa 3

b) Switch virtual implementando una VPN de capa 2

tres redes. La red del proveedor está dividida en dos partes: la región *core* y los routers PE (Provider Edge). Este trabajo argumenta la necesidad enfoques híbridos SDN/Legacy, y es aquí donde lo aplica: si bien los routers PE son OpenFlow, el core de la red no utiliza SDN, y depende de LDP y Quagga para la distribución de etiquetas MPLS. Esto implica que los routers PE, además de utilizar OpenFlow, deben implementar Quagga y LDP.

Otro dato importante que muestra la figura 2.8 es la utilización de una aplicación por cada cliente. Esto permite proveer a los clientes un control personalizado sobre su servicio de VPN. Esas aplicaciones corren sobre una visión abstracta de la red, que depende de la capa del servicio. Si la VPN es de capa 3, el controlador SDxVPN ofrece un router virtual, y si es de capa 2 ofrece un switch virtual. Cada puerto en uno de estos dispositivos virtuales representa un puerto real de un dispositivo PE con un dispositivo CE (Customer Edge). Este enfoque le ofrece al cliente una visión abstracta de la red del proveedor, y se puede ver en la figura 2.9. En el caso de una VPN de capa 3, el cliente debe elegir un mecanismo

de ruteo. Para el caso que muestra la figura, puede alcanzar con ruteo estático, pero para casos más complejos, el router virtual puede utilizar Quagga como motor. Todo esto implica que se construirá una tabla de ruteo virtual, la cual será traducida a flujos OpenFlow en los dispositivos PE.
Para la VPN de capa 2, el cliente debe asociar cada puerto del switch virtual con un conjunto de identificadores de VLAN. A ese mapeo se suma un mecanismo para aprender direcciones MAC (ya que debe comportarse como un switch de capa 2), y todo esto (igual que en la VPN de capa 3) se traduce en flujos OpenFlow que serán instalados en los dispositivos PE.

2.2.2 Otras aplicaciones

En la sección anterior se presentaron algunas implementaciones de VPNs en SDN, que es la aplicación de más interés en este trabajo. A continuación se listan otras aplicaciones que se le puede dar a OpenFlow y SDN.

- **Ingeniería de tráfico** es una disciplina que tiene como objetivo principal optimizar el rendimiento de una red. Una parte crucial de esta disciplina es poder monitorear en tiempo real el estado de cada elemento de la red. Esto es algo que OpenFlow resuelve, ya que puede registrar estadísticas por cada flujo, en tiempo real. Por ejemplo, en [52] se presenta OpenNetMon, un módulo open source para el controlador POX que aprovecha las capacidades de monitoreo de OpenFlow y hace disponibles todas esas estadísticas para las aplicaciones que corren sobre el controlador. Una aplicación luego puede utilizar ese módulo y todas las estadísticas que provee para hacer ingeniería de tráfico, y tratar de optimizar los recursos de la red.
  El manejo de los flujos para lograr load balancing y la tolerancia a fallas son otros enfoques importantes, partes de la ingeniería de tráfico en SDN. En caso de que el lector desee profundizar en este tema, se recomienda la lectura de [44], que hace un estudio completo de los enfoques actuales para la ingeniería de tráfico en redes OpenFlow.

- **Quality of Service**. Un problema muy común hoy en día es el de proveer distintos niveles de calidad de servicio, y SDN puede ser utilizado con ese propósito. El objetivo es aplicar distintos niveles de servicio a un determinado tráfico de acuerdo al cliente o al tipo de aplicación al que pertenece. Para lograr eso se pueden aplicar mecanismos como la reserva de recursos y el enrutamiento dinámico para cada flujo.
  Algunos ejemplos de trabajos hechos en este campo son: 1) FlowQoS [47], una herramienta que permite a redes hogareñas reservar ancho de banda para cada tipo de tráfico mediante OpenFlow, 2) un framework basado en EuQoS [58] para redes a gran

escala, que utiliza OpenFlow para asignar prioridades a demanda a flujos de clientes. Si el lector desea profundizar en el tema de QoS sobre SDN, se recomienda la lectura de [49].

- **Detección y mitigación de ataques DoS**. Un controlador OpenFlow podría aprovechar las métricas y estadísticas ofrecidas por los switches OpenFlow para detectar, en tiempo real, ataques de denegación de servicio. El controlador también podría reaccionar ante el ataque, aplicando las entradas de flujos que correspondan para bloquear el tráfico utilizado, y así mitigar el efecto del ataque. Por ejemplo, Dossy [63] es una aplicación que corre sobre el controlador Beacon y permite detectar y mitigar ataques DoS. Analiza los mensajes packet_in que recibe y las estadísticas de los flujos para detectar los ataques, y una vez los detecta, instala flujos para bloquear el tráfico atacante.

2.3 Simuladores y emuladores

El avance de las redes definidas por software ha motivado el desarrollo de nuevas herramientas que permitan la virtualización de este paradigma, para ayudar a los investigadores y administradores en el desarrollo y testing de sus arquitecturas. Como se explica en el capítulo 1, uno de los objetivos del proyecto es mover la arquitectura RAUFlow a una plataforma virtual. Por lo tanto, es importante hacer un estudio del estado del arte en lo que respecta a simuladores y emuladores de redes, con énfasis en los que contemplan el paradigma SDN.

La principal observación que se debe hacer es que ninguna de las herramientas estudiadas permite o contempla una naturaleza híbrida SDN/Legacy como la que plantea RAUFlow, donde los switches OpenFlow puedan comportarse como routers tradicionales. La tabla 2.2 muestra las herramientas estudiadas. La segunda columna indica si la herramienta está orientada al concepto SDN. En las últimas dos columnas se pueden ver dos propiedades que aplican sólo a las herramientas orientadas a SDN. La penúltima indica la versión de OpenFlow que soporta, y la última indica si dicha herramienta puede ser utilizada con un controlador Ryu externo, en particular, el controlador RAUFlow.

La cuarta columna especifica si se trata de un emulador o simulador (en algunos casos se da que es un híbrido) y la quinta columna detalla cómo es el modo de virtualización que aplica la herramienta. Esta columna es de especial interés para analizar la escalabilidad de la herramienta. Si se trata de un motor de simulación o una emulación con virtualización ligera, es esperable que la herramienta pueda realizar experimentos con muchos nodos. Por otro lado, si utiliza máquinas virtuales completas para representar a los nodos, probablemente se trate de una herramienta diseñada para hacer experimentos chicos, con pocos nodos.

Table 2.2 Lista de emuladores y simuladores más relevantes para este trabajo.

Herramienta	**SDN**	**Open Source**	**Simulador o emulador**	**Modo de virtualización**	**OpenFlow**	**Permite Ryu/RAUFlow**
Cloonix [3]	No	Si	Emulador	Máquinas virtuales completas	-	-
CORE [45]	No	Si	Emulador	Virtualización ligera (containers)	-	-
DOT [35]	Si	Si	Emulador	Máquinas virtuales (hosts) y Open vSwitch (switches)	1.3 (con Open vSwitch)	Si
EstiNet [61]	Si	No	Ambos	Simulación con mecanismos para ejecutar aplicaciones reales	1.1.0	Si
FlowSim [7]	Si	Si	Simulador	Simulación	1.3	No
fs-sdn [50]	Si	Si	Simulador	Simulación	No especifica versión	No
GNS3 [8]	No	Si	Ambos	Simulación con soporte para máquinas virtuales completas y ligeras	-	-
IMUNES [40]	No	Si	Emulador	Virtualización ligera (containers)	-	-
Mininet [38]	Si	Si	Emulador	Virtualización ligera (containers)	1.3 (con Open vSwitch)	Si
MLN [16]	No	Si	Emulador	Máquinas virtuales completas	-	-
Netkit [17]	No	Si	Emulador	Máquinas virtuales completas	-	-
NS-3 (extensión OpenFlow) [6]	Si	Si	Simulador	Simulación con mecanismos para ejecutar aplicaciones reales	0.8.9 (con soporte para MPLS)	Posiblemente, pero sin confirmar
Omnet++ (extensión OpenFlow) [42]	Si	Si	Simulador	Simulación	1.2	No
SDN Troubleshooting System [31]	Si	Si	Simulador	Simulación	1.0	Si
Shadow [27]	No	Si	Ambos	Simulación con mecanismos para ejecutar aplicaciones reales	-	-
VNX [33]	No	Si	Emulador	Máquinas virtuales completas / Virtualización ligera	-	-

Las siguientes secciones estudian con más detalle las herramientas que se consideran más apropiadas para virtualizar la arquitectura RAUFlow/RAUSwitch. En la última sección se hace una mención aparte para un tipo de herramientas especial, que no sirve para hacer pruebas funcionales de una red como conjunto, sino que para hacer pruebas de estrés sobre cada elemento de la red por separado.

2.3.1 DOT

DOT (Distributed OpenFlow Testbed) [35] es un emulador open source distribuido de redes OpenFlow. Está diseñado para experimentos de gran escala, es decir, con muchos nodos. La idea principal es utilizar múltiples máquinas físicas para distribuir la carga de cómputo.

En la figura 2.10 se puede ver la arquitectura de gestión de DOT. Se pueden distinguir dos tipos de componentes:

Fig. 2.10 Arquitectura de gestión de DOT. Imagen extraída de [35]

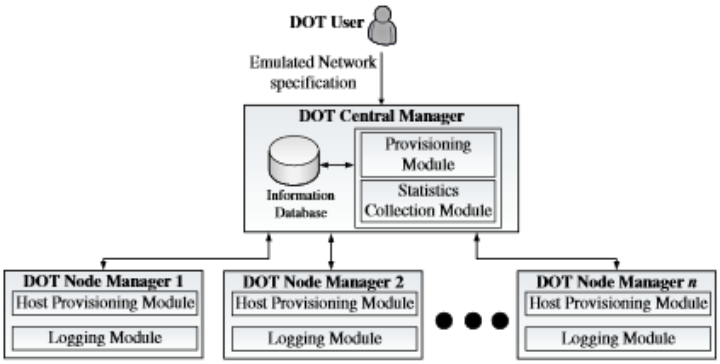


Fig. 2.11 Arquitectura de cada nodo DOT físico. Imagen extraída de [35]

- Central Manager. Este manejador (que puede estar en su propia máquina física) es responsable de gestionar los recursos para la red emulada indicada por el usuario. Está compuesto por dos módulos. El *Provisioning Module* se encarga de ejecutar un algoritmo que decide cómo utilizar los recursos físicos (las máquinas físicas disponibles) de la manera más óptima para lograr emular la red deseada. Luego le comunica los resultados del algoritmo a cada nodo físico, para que cada uno sepa los recursos debe crear. El *Statistics Collection Module* recolecta la información estadística que recibe de los nodos.

- Node Manager. Cada máquina física tiene uno, y está compuesto por dos módulos. El *Host Provisioning Module* es responsable de instanciar y configurar los recursos que indica el manejador central. Esos recursos serán los hosts, switches y links virtuales. El *Logging Module* recolecta estadísticas locales a cada nodo, como utilización de recursos, throughput, delay y mensajes OpenFlow.

La arquitectura interna de cada nodo físico en DOT se puede ver en la figura 2.11. Los hosts virtuales (VM) son provistos por un hypervisor (KVM). En la figura también se observan múltiples switches virtuales (VS) que son implementados por Open vSwitch. Cada nodo físico también tiene un Gateway Switch (GS), que se encarga de reenviar paquetes entre

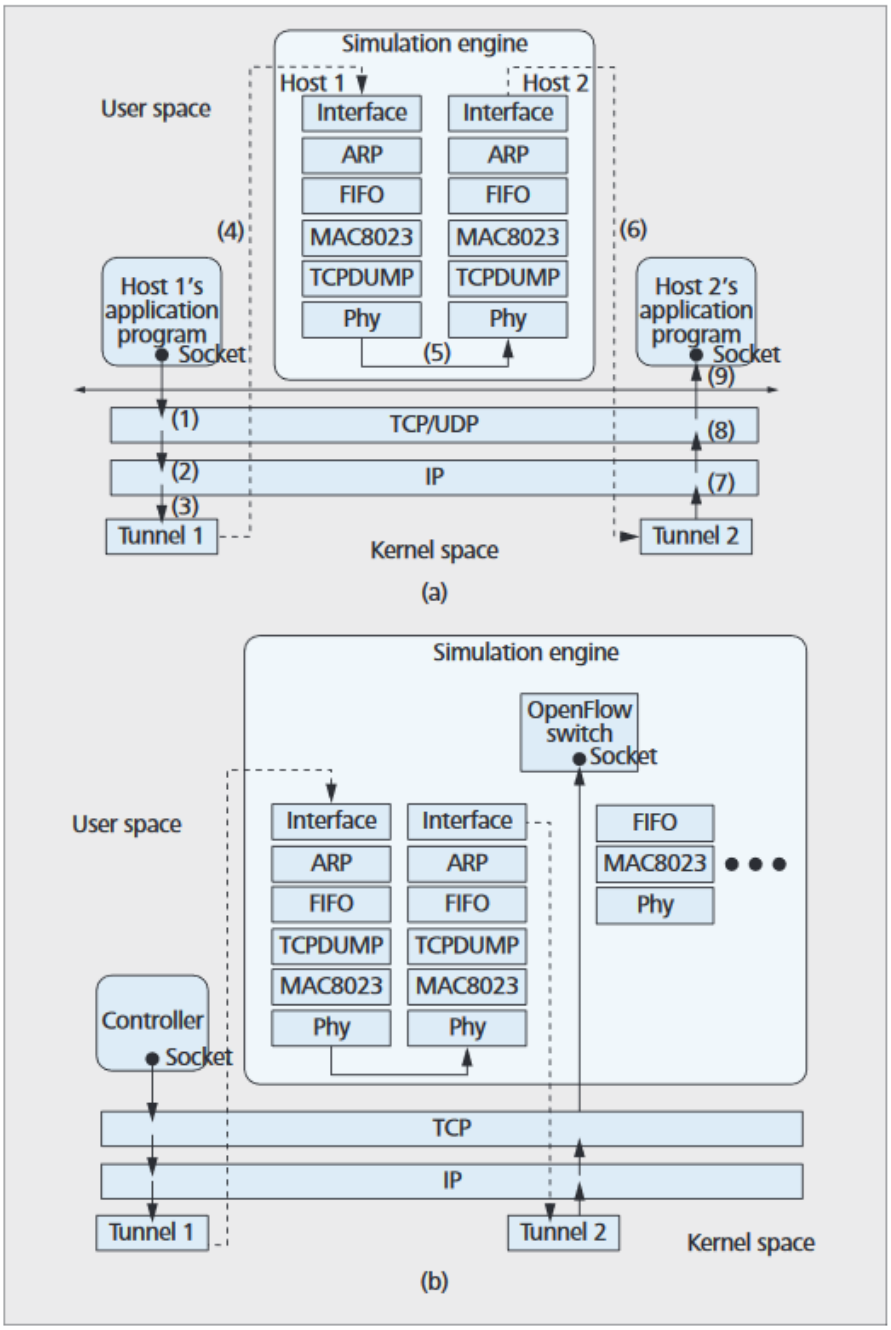
switches virtuales alojados en distintas máquinas físicas. El Gateway Switch es transparente para el usuario.

2.3.2 Estinet

EstiNet [61] es un emulador y simulador para redes OpenFlow, desarrollado por EstiNet Technologies Inc [1].

[1] http://www.estinet.com/ns/

Fig. 2.12 Arquitectura de simulación de EstiNet. Imagen extraída de [61]



Gracias a una técnica llamada *kernel re-entering*, EstiNet intenta combinar lo mejor de la simulación y emulación, para aprovechar sus ventajas y no sufrir las desventajas de cada una. El motor de simulación es capaz de simular hosts y switches OpenFlow, y utiliza interfaces de red de túnel para interceptar paquetes enviados entre aplicaciones reales. En la figura 2.12.a se puede ver como dos aplicaciones reales de Linux pueden ejecutar (desde cierto

punto de vista) en dos hosts simulados, y conectarse entre sí. Cada host simulado es dotado con una interfaz y un stack de protocolos de capa 1 y 2. Las aplicaciones envían los paquetes a los hosts simulados mediante los túneles de Linux, luego los paquetes son enviados de un host a otro en el simulador y finalmente son entregados a la aplicación destinataria mediante su túnel correspondiente.

La figura 2.12.b muestra como se puede usar la misma técnica para permitir que controladores OpenFlow reales puedan comunicarse con los switches simulados. Un controlador es una aplicación que corre sobre Linux, por lo tanto se puede correr en un host simulado como muestra el ejemplo anterior. Un switch OpenFlow simulado puede utilizar una interfaz de túnel (en el ejemplo sería *Tunnel 2*) para iniciar una conexión TCP OpenFlow con el controlador.

Este modo dual de EstiNet permite que se pueda integrar con aplicaciones reales, particularmente controladores OpenFlow reales, (algo que otros simuladores no pueden) y al mismo tiempo posea las ventajas de un simulador, como por ejemplo un reloj de simulación que garantiza la correctitud y precisión de los resultados que genera. Este enfoque también tiene ventajas en la escalabilidad, ya que se pueden conducir experimentos grandes a pesar de la limitación en el poder de cómputo de la computadora; el reloj del simulador se adecua al poder de cómputo disponible para asegurar la correctitud del experimento.

Una desventaja importante que tiene EstiNet es que, de acuerdo a [61], la versión máxima de OpenFlow que soporta es la 1.1.0, mientras que la versión que utiliza RAUFlow es la 1.3.1.

2.3.3 Mininet

Mininet es un emulador de redes SDN muy utilizado en la actualidad, que permite emular hosts, switches y enlaces. Utiliza *virtualización ligera* en Linux para levantar una red completa en un único kernel. La virtualización ligera, también llamada virtualización en el nivel de sistema operativo, consiste en un conjunto de funcionalidades de Linux que le permiten a un sistema ser separado en múltiples *containers* (contenedores) más chicos. Cada container tiene sus propios recursos, pero todos ejecutan sobre el mismo kernel, lo cual los hace más rápidos y livianos. Esto se combina con la capacidad de crear links virtuales entre dichos containers. Mininet no provee ni emula un controlador; el mismo se debe ejecutar como una aplicación común y corriente.

Este modo de funcionamiento logra que Mininet pueda ser mucho más rápido y escalable que emuladores que utilizan máquinas virtuales completas. Analizando con un poco más de detalle, a continuación se listan los componentes de Mininet y sus características:

- Hosts. Un host emulado por Mininet es un grupo de procesos de usuario que ejecutan dentro de un *network namespace*. Un network namespace le provee a un grupo de procesos sus propias interfaces de red, puertos, tablas de ruteo y tablas ARP.

- Enlaces. Los enlaces son emulados con virtual Ethernet (*veth*), que actúa como un cable que conecta dos interfaces virtuales. Los paquetes enviados por una interfaz son recibidos por la otra, y cada interfaz virtual es vista como un puerto Ethernet totalmente funcional, tanto para el sistema como para las aplicaciones. El ancho de banda de los enlaces virtuales es controlado por Linux Traffic Control (*tc*).

- Switches. Mininet puede utilizar el bridge de Linux o Open vSwitch para emular a los switches. El segundo es el que se utiliza para emular switches OpenFlow. Para emular múltiples switches OpenFlow se define un bridge por cada switch, todos ejecutándose en una única instancia de Open vSwitch. A diferencia de los hosts, que están en su propio network namespace, los switches de Open vSwitch comparten el root namespace, es decir, el namespace del sistema operativo. Como el controlador se ejecuta como una aplicación normal (no se ejecuta en un container ni tiene su propio network namespace), se puede comunicar con los switches mediante la interfaz de loopback del root namespace.

Mininet tiene una API en Python (el lenguaje en el que está escrito el emulador) que permite crear todo tipo de topologias. También tiene una línea de comandos que permite realizar acciones una vez la red virtual está levantada. Esas acciones pueden ser ejecutar un determinado comando en un host o switch, hacer una prueba de ping entre todos los hosts, etc.

Debido a su buena documentación y la forma en que está desarrollado, Mininet puede ser fácilmente extensible para lograr experimentos que la versión estándar no contempla. Existen diversos trabajos donde se presentan extensiones a Mininet, para lograr objetivos específicos:

- Mininet-HiFi [53] intenta mejorar el realismo de la emulación en Mininet, agregando mecanismos de reserva de recusos y monitoreo para los mismos.

- MaxiNet [55] y Mininet CE [60] son extensiones para hacer emulaciones distribuidas, es decir, con múltiples máquinas físicas.

- Mininet DC [46] es un trabajo que extiende Mininet y el controlador POX para hacer experimentos orientados a data centers en la nube.

2.3.4 NS-3

NS-3 es un simulador de redes open source basado en eventos discretos. Es una herramienta desarrollada para ser extensible por módulos escritos por los usuarios. Es por esta característica que puede ser utilizada para usar OpenFlow, a pesar de no ser una herramienta enfocada hacia el paradigma SDN.

Existe una implementación de OpenFlow [6] (llamada OFSID - *OpenFlow software implementation distribution*) que se puede integrar con ns-3 en modalidad de librería externa. Esta extensión implementa una versión antigua de OpenFlow, la 0.8.9, pero agrega soporte para MPLS.

Un problema de esta extensión es que la entidad controlador es modelada internamente en el simulador. Esto implica que no es posible que el simulador funcione con un controlador externo. Esto podría resolverse con una funcionalidad llamada *Direct Code Execution* (DCE), que permite la ejecución de aplicaciones externas dentro de ns-3. Sin embargo, DCE sólo soporta, de forma oficial, aplicaciones C/C++, y el soporte para Python por el momento es experimental.

Otro detalle importante es que no es posible modelar la conexión SSL entre un switch OpenFlow y el controlador (ya que es interno). Esto baja la calidad de la simulación ya que podría ser un punto interesante de estudio.

En resumen, NS-3 es un simulador muy usado en la actualidad pero su soporte para redes OpenFlow podría etiquetarse como experimental por el momento. Es necesario hacer un estudio muy detallado para determinar si se puede utilizar con la arquitectura RAUFlow, pero no se descarta que así sea.

2.3.5 IMUNES y otros emuladores de propósito general

IMUNES [40] es un emulador open source para Linux y FreeBSD. Funciona con nodos virtuales ligeros (o containers) conectados entre sí por enlaces emulados.

Para implementar redes virtuales en Linux usa dos tecnologías: Docker y Open vSwitch. Docker es una herramienta open source que utiliza virtualización a nivel de sistema operativo (o virtualización ligera) para crear *containers*. Estos nodos virtuales o *containers* son muy similares a los que implementa Mininet, y tienen como gran ventaja que son mucho más rápidos y utilizan menos recursos que las máquinas virtuales completas.

IMUNES es similar a otras herramientas que usan virtualización ligera como VNX [33], CORE [45] y GNS3 [8]. Son opciones bastante más escalables que otros emuladores como Cloonix [3], MLN [16] y Netkit [17], que utilizan máquinas virtuales completas.

2.3.6 Herramientas para pruebas de estrés y benchmarking

Existen múltiples herramientas de testing aplicables en el paradigma SDN. De ellas se puede destacar un grupo, que no tienen como objetivo verificar aspectos funcionales, sino que apuntan a algo mucho más específico: las pruebas de estrés y el benchmarking. Ayudan a los administradores de red e investigadores a conocer los niveles de rendimiento de los cuales sus dispositivos y controladores son capaces. Asimismo, pueden ser elementos de investigación útiles en el contexto de la nueva Red Académica, ya que podrían utilizarse, por ejemplo, para validar ciertos aspectos de rendimiento de los RAUSwitch.

En esta sección se estudiarán algunos ejemplos de estas herramientas, agrupándolas en dos grupos, de acuerdo a la entidad que intentan probar: switches y controladores. Para mantener la relevancia con el contexto de este trabajo, se limitará a herramientas enfocadas a OpenFlow.

Testing de switches

El testing y benchmarking de switches OpenFlow tiene como objetivo analizar cuál es el nivel máximo de rendimiento que puede alcanzar un switch. Esto por lo general se logra simulando las condiciones de una red que está bajo mucha carga, y sometiendo al switch a dichas condiciones, monitoreando su comportamiento. A continuación se listan algunas herramientas que hacen esto:

- **OFLOPS** [39] (OpenFlow Operations Per Second) es un framework open source para el testing y benchmarking de switches OpenFlow, tanto físicos como virtuales. Está desarrollado en el lenguaje C, y utiliza librerías de manipulación de paquetes para emular un controlador OpenFlow y tráfico de uso. Fue diseñado con un enfoque multi-thread para aprovechar las arquitecturas multi-core y así aumentar la potencia de la plataforma. Consiste de cinco threads paralelos, cada uno cumpliendo una función específica: 1) generación de paquetes, 2) captura de paquetes, 3) administración del canal de control (mensajes OpenFlow), 4) administración de un canal SNMP para hacer consultas asíncronas, y 5) un manejador de tiempo. Todo esto lo ofrece mediante una API, permitiendo a los usuarios crear sus propios módulos para escribir pruebas que se adapten a su realidad.

- **Spirent OpenFlow Controller Emulation** [29] es una herramienta desarrollada por la empresa Spirent[2] que, igual que OFLOPS, tiene como propósito el testing y benchmarking de switches OpenFlow. Puede emular un controlador OpenFlow, definir millones de flujos y aplicar patrones de tráfico a esos flujos, y de ese modo medir el

[2] http://spirent.com/

rendimiento, disponibilidad, seguridad y escalabilidad del switch. Entre sus funcionalidades, se destaca que puede probar todos los aspectos de OpenFlow 1.3, y que puede trabajar con switches híbridos. Estos dos puntos lo hacen una valiosa herramienta para el testing del RAUSwitch.

Testing de controladores

Similar al caso de los switches, las pruebas de estrés sobre controladores OpenFlow consisten en someterlos a condiciones de mucha carga y estudiar determinadas métricas de su comportamiento. En general los principales aspectos que se buscan estudiar son 1) la cantidad de sesiones paralelas con switches OpenFlow que puede mantener el controlador y 2) el ritmo de mensajes packet_in que puede manejar. No sólo es útil conocer esos umbrales, sino que también es muy valioso saber cual es el comportamiento esperado si se exceden dichos umbrales. Al ser RAUFlow un controlador de estilo proactivo, el aspecto número dos no es relevante aquí, ya que la red no genera paquetes de tipo packet_in. A continuación se listan algunas de las herramientas disponibles:

- **Cbench** [2] es una herramienta para benchmarking de controladores, y es parte del proyecto OFLOPS. Su funcionamiento es muy simple: el usuario indica una cantidad *n* de switches, la herramienta crea *n* sesiones OpenFlow paralelas con el controlador, y luego comienza a enviar mensajes de tipo packet_in y mide el tiempo que demora el controlador en responder a esos mensajes.

- **Spirent OpenFlow Switch Emulation** [30] es la propuesta de Spirent para el stress-testing de controladores OpenFlow. Igual que Cbench, en esencia consiste en emular múltiples switches y generar mensajes de tipo packet_in. Sin embargo, es una solución un poco más sofisticada que Cbench, ya que permite trabajar con múltiples topologias y protocolos como ARP y LLDP.

Capítulo 3

Entorno virtual

Uno de los principales objetivos de este trabajo es realizar pruebas funcionales y de escala sobre la arquitectura del prototipo. Es de interés generar distintas realidades, y así detectar puntos de falla o variables clave en el rendimiento de la arquitectura. Para modelar las distintas realidades se puede utilizar dos parámetros: topología y servicios. Es importante poder aplicar topologias complejas y relativamente grandes a la arquitectura, así como grandes cantidades de servicios, y de esta forma encontrar posibles problemas con la arquitectura, y su respectiva solución. Dado que no es realista hacer este tipo de pruebas con un prototipo físico, por temas económicos y prácticos, se observa la necesidad de un entorno virtual capaz de simular las características del prototipo. Es importante remarcar que también tendría un gran valor como herramienta de investigación, para trabajar sobre la arquitectura de RAUFlow pero también para futuros estudios sobre esquemas híbridos SDN/Legacy.
En este capítulo se estudian los requerimientos que debe cumplir este entorno y los detalles de diseño e implementación de la solución construida. También se explican los principales problemas o dificultades encontradas para lograr un correcto funcionamiento, así como el desarrollo de un módulo que permite cargar topologias automáticamente desde archivos con formato GraphML.

3.1 Requerimientos del entorno virtual

El primer paso en la construcción del entorno virtual es analizar cómo debería comportarse. Se podría hacer este análisis en dos partes separadas. En primer lugar, se deben cumplir los aspectos funcionales de la arquitectura de RAUFlow. No es un requerimiento que se utilicen las mismas herramientas que utiliza el prototipo físico pero es deseable que así sea. Cuanto más similar sea el entorno a la arquitectura, más relevantes serán las pruebas que se lleven

a cabo. Otra ventaja es que se pueden reutilizar recursos, como por ejemplo, archivos de configuración. Este primer grupo de requerimientos se detalla a continuación.

1. Se debe poder simular múltiples RAUSwitch virtuales, y los mismos deben tener las mismas capacidades funcionales que sus pares físicos. A partir de esto, se desprenden los siguientes sub-requerimientos.

   (a) Deben poder utilizar el protocolo de enrutamiento OSPF. Esto es necesario ya que la base de datos topológica de RAUFlow se construye a partir de la base de datos local de OSPF (Link-State Database). Es deseable que lo hagan mediante el software de enrutamiento Quagga.

   (b) Resulta trivial que los RAUSwitch virtuales soporten OpenFlow, ya que es el cimiento de RAUFlow. Específicamente, deben soportar la versión 1.3. Esto se debe a que la implementación de las VPNs depende de que los nodos tengan soporte para MPLS, y OpenFlow ofrece esta funcionalidad de forma completa a partir de la versión 1.3. Es muy deseable que lo hagan mediante Open vSwitch, ya que es la herramienta utilizada por los RAUSwitch físicos.

   (c) En la arquitectura de RAUFlow se usan agentes SNMP para que los RAUSwitch envíen información que no es soportada por el protocolo OpenFlow acerca de sus interfaces. Como requerimiento para este entorno, es importante que los nodos puedan enviar esa información de algún modo, pero no es necesario que sea a través de SNMP, ya que no es una parte vital de la arquitectura.

2. Se debe poder simular múltiples hosts, ya que son los agentes que se conectan a la red y utilizan la misma para enviarse datos entre sí. De esta forma se corrobora que el funcionamiento de la red es el correcto.

3. La aplicación RAUFlow debe ejecutarse y comunicarse correctamente con los RAUSwitch. Esto también implica que el controlador Ryu debe ser soportado por el entorno.

   En el segundo grupo de requerimientos, se consideran los que son inherentes a cualquier herramienta de virtualización que se utilizará para pruebas de escala como las que se pretenden.

4. Facilidad de configuración. Es importante que el entorno pueda generar distintas topologias y escenarios sin demasiado esfuerzo de configuración.

5. Escalabilidad. El entorno debería ofrecer buena escalabilidad en la cantidad de nodos que puede simular. Esto se traduce a que una computadora promedio de uso personal pueda levantar algunas decenas de nodos virtuales como mínimo.

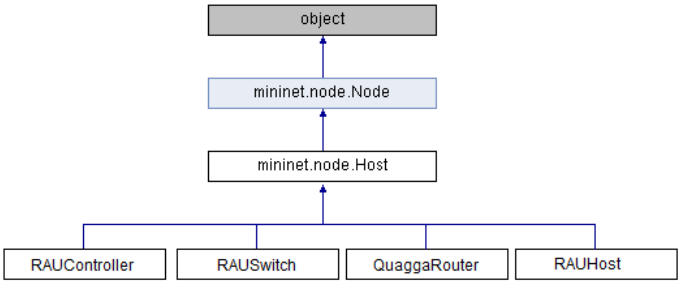
3.2 Elección de la herramienta

Teniendo definidos los requerimientos, el siguiente paso es obtener una herramienta de virtualización que cumpla dichos requerimientos. Se descarta una construcción "desde cero" de la misma, ya que se considera un esfuerzo demasiado grande para el alcance de este trabajo. Por lo tanto, se debe utilizar una herramienta existente. En el capítulo 2 se presentó un estudio del estado del arte de los emuladores y simuladores de red disponibles en la actualidad. La tabla 2.2 muestra todas las herramientas estudiadas, y se las puede separar en dos grupos: las orientadas específicamente para SDN y las de propósito general. Muchas de las orientadas a SDN fallan en requerimientos básicos, como la versión de OpenFlow o poder utilizar un controlador personalizado (requerimientos 1.b y 3), y por lo tanto se descartan. Por otro lado, Mininet cumple todos los requerimientos excepto el relacionado al enfoque híbrido (el 1.a) ya que es una herramienta diseñada para el paradigma SDN puro y sus switches no soportan los protocolos IP legados.

Otra opción podría ser utilizar una de las herramientas de propósito general, como IMUNES. Ese tipo de enfoque tiene la ventaja de que los aspectos funcionales (requerimientos 1-3) no son un problema. Esto se debe a que al ser herramientas de propósito general, en definitiva lo que proveen son nodos virtuales Linux genéricos. Por lo tanto se podría replicar lo hecho en los nodos físicos del prototipo en dichas máquinas virtuales. El problema de esa opción radica en los últimos dos requerimientos (4 y 5). En primer lugar, muchas de esas herramientas no tienen formas fáciles y rápidas de configurar distintas topologias. En segundo lugar, la escalabilidad de esas herramientas puede ser una limitación. En particular, las herramientas que utilizan máquinas virtuales completas no son un buena opción, ya que cada nodo necesita demasiados recursos de cómputo y no sería viable hacer experimentos de escala. Podrían ser viables las que utilizan virtualización ligera (o basada en containers) ya que cada nodo virtual utiliza menos recursos.

Tomando todo esto en cuenta, la herramienta que se elije para construir el entorno es Mininet, pero de un modo no tradicional. Como se explica en el capítulo 2, Mininet, además de ofrecer switches, ofrece hosts que son containers reducidos. Por lo tanto, la solución que se propone es utilizar a Mininet como emulador de propósito general y basarse en dichos hosts para construir los RAUSwitch virtuales. De este modo, tomando como base lo hecho

Fig. 3.1 Diagrama de clases del entorno.



para los nodos físicos se cubren los requerimientos funcionales, y también se aprovecha la configurabilidad y escalabilidad de Mininet.

3.3 Diseño e implementación del entorno

Como explica la sección anterior, el entorno está construido alrededor de Mininet, y se podría pensar como una extensión de la misma. *Out of the box*, Mininet ya cumple la mayoría de los requerimientos estudiados anteriormente. Está diseñada para ser escalable, ya que usa containers reducidos, tiene soporte para OpenFlow 1.3 mediante Open vSwitch, y gracias a su API en Python es muy fácil de configurar. El aspecto en el que falla es en el soporte para Quagga. Dado que Mininet es una herramienta de prototipado para SDN puro, no está pensado para un esquema híbrido como el que se propone. Los switches compatibles con Open vSwitch que ofrece no pueden tener su propio network namespace, por lo tanto, no pueden tener su propia tabla de ruteo ni interfaces de red aisladas, así que no es posible que utilicen Quagga.

Por otro lado, los hosts de Mininet sí tienen su propio network namespace, y gracias a su capacidad de tener sus propios procesos y directorios, es posible ejecutar una instancia de Quagga y Open vSwitch para cada host. De esta forma es posible convertir un host en un RAUSwitch como el requerido por la arquitectura. Esta extensión de las funcionalidades de los hosts es posible ya que Mininet está programado con orientación a objetos y permite al usuario crear subclases propias de las clases que vienen por defecto. En la figura 3.1 se puede ver la estructura de clases del entorno construido. En las siguientes secciones se procederá a estudiar cada una de ellas.

3.3.1 RAUSwitch

La clase RAUSwitch es el núcleo del entorno virtual. Como se explica anteriormente, no es posible utilizar la clase Switch de Mininet, así que se utiliza la clase Host y se la extiende para cumplir los requerimientos del RAUSwitch.

Cada RAUSwitch está en su propio network namespace (igual que la clase Host) y ejecuta su propia instancia de Quagga y Open vSwitch. Esto se logra gracias a la funcionalidad de directorios privados de Mininet. Cada RAUSwitch tiene los siguientes directorios privados: /var/log/, /var/log/quagga, /var/run, /var/run/quagga, /var/run/openvswitch. Cada RAUSwitch también usa un directorio bajo /tmp, para almacenar sus archivos de configuración.

No es posible utilizar una única instancia de Open vSwitch (como en Mininet estándar, recordar sección 2.3.3 del estado del arte) ya que no es posible que una instancia defina múltiples switches, cada uno en su propio network namespace. Y se necesita que cada uno tenga su namespace para que pueda comportarse como router legado y tenga sus propias tablas de ruteo e interfaces. Aunque Quagga no presenta ningún problema para ejecutarse de este modo, es importante tener en cuenta que Open vSwitch no está diseñado para ejecutarse en múltiples instancias paralelas, y por lo tanto hacer que funcione correctamente es una tarea más compleja. Como la comunicación entre los componentes de Open vSwitch se efectúa mediante Unix Sockets, es necesario que cada instancia tenga su propio socket, de lo contrario si todas las instancias utilizaran el mismo socket ocurriría un problema de lockeo.

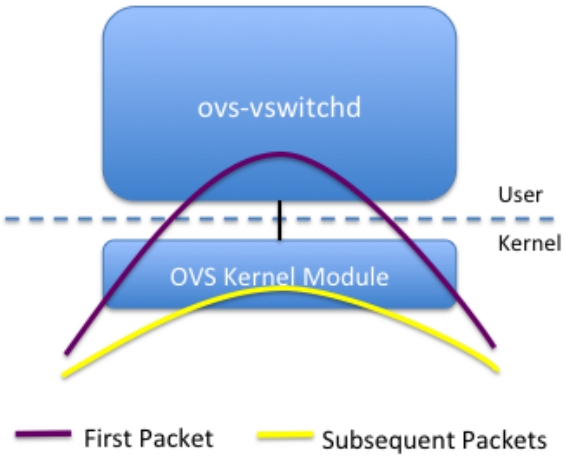
Si bien el enfoque de múltiples instancias de Open vSwitch funciona, lo hace con determinadas limitaciones. Como se explica en la sección 2.1.3 del capítulo del estado del arte, Open vSwitch utiliza un módulo en el kernel que actúa como cache para flujos recientes y que aumenta radicalmente la velocidad de procesamiento de paquetes. Debido a que Open vSwitch no contempla la existencia de múltiples módulos de kernel paralelos, y no se puede utilizar un único módulo para múltiples instancias, es necesario que cada instancia se ejecute completamente en el userspace. Esto implica que cada paquete que un RAUSwitch procese deberá ser evaluado por su demonio ovs-vswitchd que se encuentra en el userspace, lo cual reduce la velocidad significativamente.

Obviamente esto no es deseable, pero no significa una desventaja muy seria, ya que el objetivo del entorno no es ser performante al procesar paquetes. Cabe aclarar que en este modo Open vSwitch continúa haciendo cacheo de flujos, pero ahora lo hace en el userspace.

3.3.2 RAUController

En el uso típico de Mininet, la comunicación entre el controlador y el switch se da a través de la interfaz de loopback. Esto es así porque los switches no tienen su propio namespace. Para

Fig. 3.2 Arquitectura simplificada de Open vSwitch.



lograr dicha comunicación, no hace falta un objeto en Mininet que represente el controlador, ya que ejecutar la aplicación en el sistema operativo base ya habilita al switch a comunicarse con ella a través de la interfaz de loopback. Esta situación cambia en este diseño, porque los switches pasan a tener su propio network namespace. Esto lleva a la necesidad de crear un host virtual, que ejecute la aplicación de RAUFlow y se comunique con los switches a través de enlaces virtuales. Para satisfacer esta necesidad se usa la clase RAUController.

3.3.3 QuaggaRouter

Es una clase similar al RAUSwitch pero sin Open vSwitch, es decir, sólo usa Quagga. Apunta a representar el router CE que utilizaría una subred para conectarse a la red. Está conectado a un RAUSwitch de borde.

3.3.4 RAUHost

Representa a los hosts que serán clientes de la red. Con este propósito, se podría utilizar directamente la clase Host de Mininet, pero se construye esta clase auxiliar para evitar determinadas configuraciones manuales, como por ejemplo, el *default gateway*.

3.3.5 Sustitución del agente SNMP

Como se explicó en el capítulo (X), RAUFlow usa un agente SNMP en cada switch para hacer disponible al controlador, información acerca de sus interfaces de red que no puede enviar a través de OpenFlow. Específicamente, los datos que se envían a través de SNMP

para cada interfaz de red son dirección IP, dirección MAC y nombre de interfaz. Para lograr esto en el entorno virtual se debe crear una instancia de agente SNMP por cada RAUSwitch virtual que se ejecuta, y se las debe configurar de tal forma para que cada una sólo devuelva la información de su nodo virtual.
Si bien es posible, se observa que lograr esto es relativamente complejo, debido que se aleja mucho del uso tradicional de SNMP. A esto se suma el hecho de que si bien es necesario que esa información llegue al controlador de alguna forma, no es estrictamente necesario que sea mediante SNMP. Por lo tanto, se concluye que la información que RAUFlow envía al controlador a través de SNMP, sea enviada de otra forma en el entorno virtual.

La alternativa que se construye está basada en Open vSwitch. Mediante el comando 'ovs-vsctl list bridge', se pueden ver las distintas propiedades del switch, como el identificador de datapath, estado, etc. Entre esas propiedades, existe un campo llamado 'other_config', al que se le pueden agregar un número de configuraciones adicionales. Entonces, se utiliza ese campo para almacenar una propiedad llamada 'ports_info' que almacenará la información sobre todas las interfaces del nodo. Esta propiedad no tendrá significado para Open vSwitch, por lo tanto quedará intacta. El valor que almacenará la propiedad debe ser de tipo String, y probablemente almacene la información de múltiples interfaces de red, así que se debe crear un formato para esa información. Por ejemplo, si tenemos un switch con la siguiente información:

```
Interfaz de red 1:
  Nombre: eth1
  Direccion IP: 10.0.0.1
  Direccion MAC: 00:00:00:00:00:01
Interfaz de red 2:
  Nombre: eth2
  Direccion IP: 10.0.0.2
  Direccion MAC: 00:00:00:00:00:02
```

El valor de ports_info sería el siguiente:

```
eth1_10.0.0.1_00:00:00:00:00:01/eth2_10.0.0.2_00:00:00:00:00:02
```

Como se puede ver, el formato indica que se usa el carácter '/' para separar la información de cada interfaz, y el carácter '_' para dividir los campos de información de cada interfaz. Este formato no es el único posible, se puede utilizar cualquier otro siempre y cuando use los caracteres permitidos por Open vSwitch.

Después de configurar el campo ports_info, cada instancia de Open vSwitch almacena la información de las interfaces de su respectivo nodo virtual. Sin embargo, esta información todavía no es accesible desde afuera del nodo. Para lograr que Open vSwitch pueda enviar esta información por la red, se utiliza un comando llamado 'set-manager'. Con él, se le puede indicar a Open vSwitch que escuche en un determinado puerto (típicamente el puerto

TCP 6640) para que pueda ser gestionado de forma remota. Por lo tanto, si el controlador desea saber los datos de las interfaces de un determinado switch, alcanza con que le envíe el comando 'ovs-vsctl list bridge' a través de la red y parsear la respuesta para extraer los datos de interés.

Como se explicó anteriormente, esta alternativa se construye para evitar la adaptación de SNMP al entorno virtual. Sin embargo, se considera que es una buena solución al problema de los datos de las interfaces. En primer lugar, la arquitectura se simplifica, ya que elimina la necesidad de SNMP. Esta solución utiliza aspectos de Open vSwitch, que ya está integrada a la arquitectura, por lo que no agrega mucha complejidad. En segundo lugar, eliminar la necesidad de que cada nodo ejecute un agente SNMP ayuda a la performance de los mismos. Por estas razones, se sugiere esta solución como aporte a RAUFlow.

3.4 Módulo de carga automática - GraphML Loader

Como se explica en el Apéndice 1, cada topología se debe configurar en un script Python usando la API del entorno. Este proceso puede ser tedioso, especialmente si se trata de una topología grande. Todos los nodos se deben inicializar con los parámetros que requieren, también se deben crear los enlaces entre ellos, y cualquier problema de coherencia en los datos puede resultar en un funcionamiento inadecuado de la topología.

Tomando como base [48], se desarrolló un módulo llamado **GraphML Loader**. Su propósito es facilitar la tarea de configurar topologias nuevas. GraphML es un formato de archivo que sirve para detallar grafos (sus nodos, enlaces, etc), y está basado en XML. Este formato es usado, por ejemplo, en el dataset de Topology Zoo [32] para detallar las topologias. El propósito de este módulo es recibir como entrada un archivo de tipo graphml y producir como salida un script Python que configura la topología que se corresponde con el grafo de entrada. El módulo se encarga no sólo de crear una topología que se adapte al grafo, sino que también de asignar automáticamente las direcciones IP de cada nodo, indicar qué nodos son de borde y cuales no, entre muchas otras cosas. Este módulo aporta una gran mejora a la usabilidad del entorno virtual, ya que si el usuario dispone de un archivo de tipo graphml, puede tener una topología lista para usarse en pocos segundos. En caso que se desee hacer una modificación a dicha topología, alcanza con hacerla en el script que produjo el módulo. En el Apéndice 2 se explica más detalladamente como funciona este componente.

3.5 Validación funcional

Con el emulador para RAUFlow construido, es necesario llevar a cabo una serie de pruebas para asegurar que: a) el emulador funciona correctamente, b) la arquitectura RAUFlow no tiene problemas con topologias grandes. El punto b) es necesario ya que la validación funcional llevada a cabo por el proyecto RRAP (explicada en el capítulo 6 de [43]) sólo incluye una topología de 4 nodos.

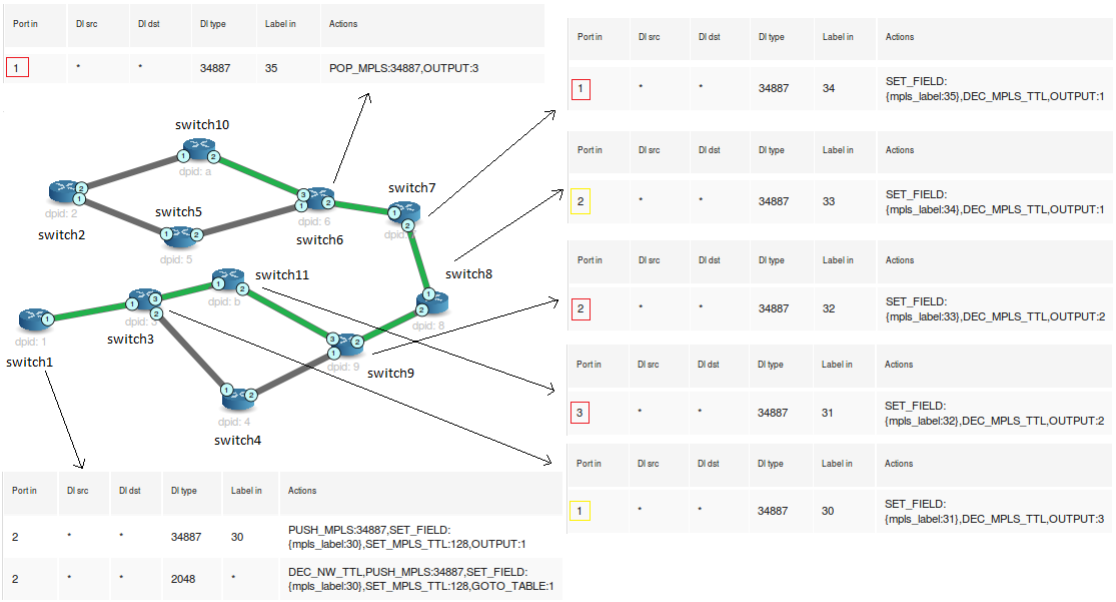
La metodología de las pruebas es relativamente simple. Se crea una VPN punto a punto de capa 3 entre dos extremos de la red, y se controla que tanto la creación de la VPN como el uso de la misma con tráfico funciona correctamente. Esto se repite con distintas topologias de prueba. Los puntos específicos que se busca verificar se detallan a continuación:

Algoritmo de ruteo
Se verifican dos aspectos claves: que el camino se corresponde con el camino esperado (calculado previamente de forma manual), y que el camino es correctamente instalado en forma de reglas de reenvío (en base a conmutación de etiquetas MPLS) en las respectivas tablas de flujos OpenFlow de cada nodo del camino. Todo esto se puede comprobar analizando las tablas de flujos de cada nodo, que se pueden ver utilizando el comando **dump-flows** de Open vSwitch. También se puede utilizar la interfaz gráfica de RAUFlow, aunque no se recomienda su uso con topologias grandes ya que la forma de presentar los nodos se vuelve demasiado caótica. Desde las tablas de flujos se puede reconstruir el camino que computó la aplicación, y también comprobar que los flujos manipulan correctamente las etiquetas MPLS.

Clasificación de tráfico
La idea es verificar que realmente se están asignando las etiquetas MPLS al tráfico entrante, así como comprobar que el mismo es reenviado por los nodos correctos. Para probar esto se utiliza tráfico con ethertype 0x0800, es decir, del protocolo IPv4. Ese es el ethertype que admite la VPN creada. Se genera tráfico de este tipo utilizando el comando **ping** y la herramienta **iperf**. Utilizando la herramienta tcpdump en cada nodo, se verifica que el tráfico pasa correctamente por cada nodo del camino.

En las siguientes sub-secciones se explican los principales problemas y errores detectados como consecuencia de las pruebas realizadas. Cada una explica en que consiste cada problema, estudiando sus síntomas, su explicación, y en caso de que exista una, su solución. Como se verá, algunos de ellos se manifestarían en un despliegue real de la arquitectura, y

Fig. 3.3 Escenario donde los flujos están mal configurados. Muestra la topología de la red y los flujos de interés para cada nodo.



otros son causados por el uso de un entorno virtual, y por lo tanto no se deberían tener en cuenta en un despliegue real.

3.5.1 Errores en el código de RAUFlow

Se descubrió que existían determinados errores (o "bugs") en el código de RAUFlow. Como son errores en el código del controlador, es importante remarcar que estos errores sin lugar a dudas se manifestarían en una red real.

Error en flujos de servicio con más de un salto

Se observó que cuando se trataba de crear un servicio que pasara por más de 2 nodos (es decir, con más de un salto), el controlador instalaba flujos en los nodos correctos, pero los flujos mismos no eran correctos. Esto indicó que el problema no se encontraba en el algoritmo de ruteo, sino que en el algoritmo encargado de configurar los flujos en cada nodo del camino computado. Específicamente, el problema es que los flujos en los nodos intermedios (es decir, los nodos donde no empieza ni termina el servicio) tenían un incorrecto puerto de entrada. En la figura 3.3 se examina este comportamiento. Los enlaces verdes muestran el camino del servicio que se intentó crear, y cada flecha indica la tabla de flujos (reducida) de cada nodo relevante. Si se presta atención a los flujos en los nodos intermedios,

se puede ver que los puertos de entrada de cada flujo coinciden con el puerto de salida del flujo en el nodo anterior. Los recuadros rojos muestran los puertos incorrectos y los amarillos indican los que podrían haber sido incorrectos pero no lo son por coincidencia. Por ejemplo, si se observa la tabla de flujos del switch11, se puede ver que el flujo indica que el puerto de entrada por el que se espera el tráfico es el número 3. Sin embargo, dicho switch no tiene un puerto número 3. El controlador asignó incorrectamente el puerto 3 ya que el switch3 (el anterior en el camino) tiene el número 3 como puerto de salida para el flujo. Para solucionar esto se creó un "fork"[1] del repositorio de RAUFlow, y se hizo el commit[2] correspondiente.

Error de tipos en el algoritmo de ruteo

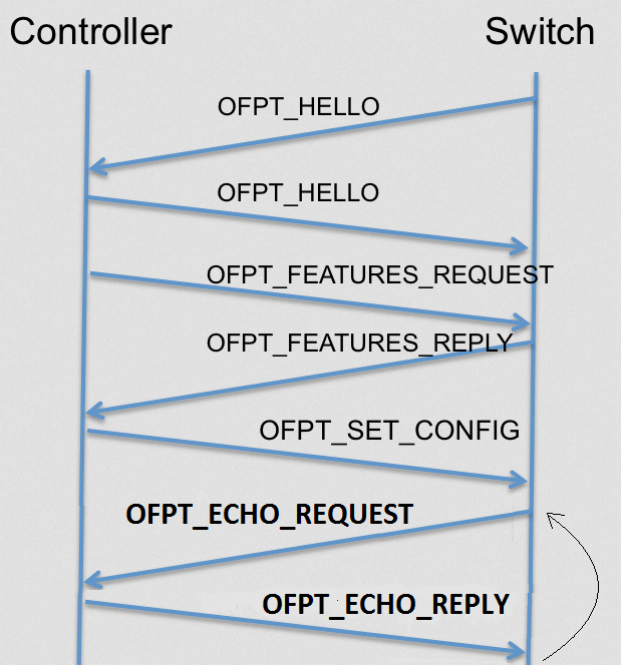
Se detectó que al intentar crear servicios en algunas topologias, se producía un error 500 de Python en RAUFlow. Luego de inspeccionar el código se concluyó que el problema radicaba en la implementación del algoritmo de ruteo, que está basado en Dijkstra. En el proceso de calcular el camino óptimo, el algoritmo de Dijkstra acumula iterativamente los costos desde el origen hasta los nodos intermedios. Dado que los costos de cada enlace son números enteros, la acumulación de costos debería implementarse simplemente aplicando suma entera a dichos costos. Sin embargo, la estructura interna usada para representar el grafo, almacena los costos de cada enlace con tipo "String". Al hacer la suma para acumular los costos, en vez de aplicar suma entera, el código hacía concatenación de strings. Dada la estructura interna del código, que no se mencionará para simplificar, esto generaba un error 500 sólo en algunas topologias. Para solucionar esto se realizó un casteo de String a Int en el código del algoritmo. Al igual que en el error mencionado en el punto anterior, se realizó un commit[3] con dicho arreglo.

3.5.2 Problemas de comunicación con muchos nodos

Cuando se iniciaba el entorno con una topología grande, de unos 100 nodos aproximadamente, ocurría que al principio los nodos se registraban correctamente con el controlador y la comunicación parecía ser la esperada, pero luego de unos segundos el controlador anunciaba que los nodos se habían desconectado. Igual que el prototipo físico, el entorno virtual necesita que cada topología también defina una red de gestión para que el controlador pueda comunicarse con los nodos en modo out-of-band. Dado que era probable que el problema se encontrara en dicha red, se utilizó la herramienta Wireshark (un analizador de protocolos ampliamente usado) para monitorear la comunicación entre los nodos y el controlador. Si

[1] https://github.com/santiagovidal/LiveCode
[2] https://github.com/santiagovidal/LiveCode/commit/aeb575a10eb241dc3980a4c37846af7551bb7060
[3] https://github.com/santiagovidal/LiveCode/commit/4128923efcff38768aefd2864e10bd1adb63df52

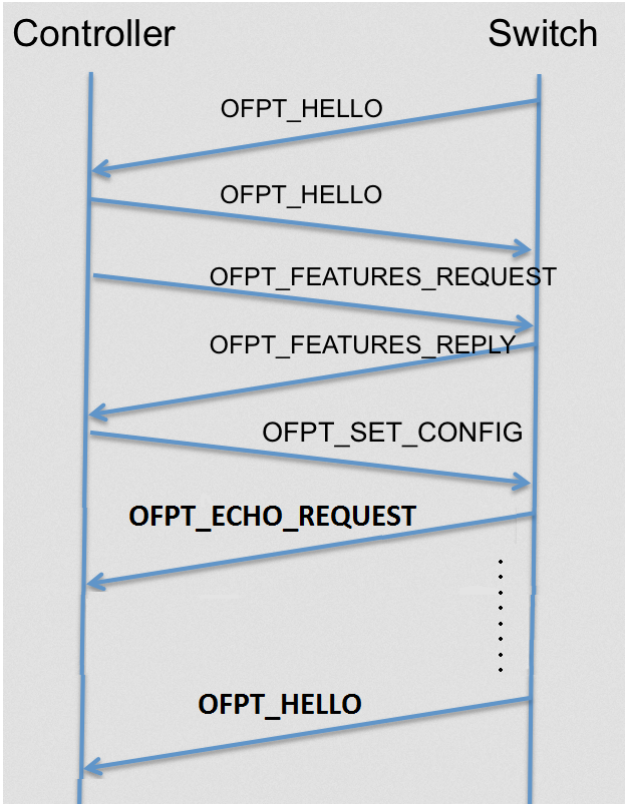
Fig. 3.4 Protocolo de control de OpenFlow.



la comunicación es correcta, se debería ver una interacción como la que muestra la figura 3.4. Luego del 'handshake' inicial, se entra en un ciclo en el que el switch envía un mensaje llamado Echo Request al controlador y espera la respuesta Echo Reply del mismo. Cuando se recibe dicha respuesta se empieza de nuevo el ciclo.
Sin embargo, lo que en realidad se observa en Wireshark se indica en la figura 3.5. El handshake se efectúa correctamente, y cuando se inicia el ciclo de Echo Request - Echo Reply, en algún punto el controlador no responde, o demora demasiado en mandar la respuesta. En la figura se muestra que eso ocurre con el primer Echo Request, pero no necesariamente es el caso en la práctica. Cuando el switch detecta que pasó demasiado tiempo esperando la respuesta a su Echo Request, asume que la conexión se ha finalizado y envía de nuevo mensajes Hello para intentar volver a conectarse.

De acuerdo al manual de Open vSwitch [54], la variable responsable de este comportamiento es la llamada **inactivity_probe**. Se define como el máximo número de mili-segundos que debe esperar el switch antes de enviar un mensaje sonda por inactividad (Echo Request). Si Open vSwitch no se comunica con el controlador por la cantidad especificada de segundos, enviará una sonda, es decir, un mensaje Echo Request. Si la respuesta no es recibida dentro la misma cantidad de mili-segundos adicional, Open vSwitch asume que la conexión se ha finalizado e intenta reconectarse. El valor por defecto para esta variable

Fig. 3.5 Error de comunicación en protocolo de OpenFlow.



depende de la implementación, y en el ambiente de trabajo tiene por defecto el número 5000 (5 segundos). Esto quiere decir que si el switch no recibe mensajes del controlador por 10 segundos (5 luego de mandar la sonda) se cerrará la conexión.
Lo interesante de esta situación es estudiar por qué el controlador no logra responder a tiempo las sondas de los nodos. Algunas posibles razones son las siguientes:

- Congestión en la red de gestión. Es posible que el exceso de nodos genere demasiado tráfico de control, y por ende haya retrasos y/o pérdidas en la red de gestión. Este problema posiblemente estaría presente en un despliegue real de la arquitectura.

- Falta de capacidad de cómputo. Dado que este comportamiento se detectó en el entorno virtual y con un número importante de nodos, es posible que el controlador no tenga acceso al poder de cómputo suficiente como para responder a tiempo a todas las sondas. Este sería un problema del entorno virtual, y no sería relevante en una red real.

- Incapacidad de Ryu para manejar muchos nodos. En [43] se menciona que Ryu es un controlador minimalista y académico. Por lo tanto, es posible que no esté diseñado para controlar tantos switches. Si ese fuera el caso, esto sería un obstáculo al desplegar

la arquitectura en una red real. TODO!: Mencionar CBench o otro benchmark para controllers como posible manera de analizar.

Como solución provisoria, con el propósito de realizar pruebas con las topologias que presentan este problema, se configuró el valor de inactivity_probe como un número muy alto (45 segundos) de modo de darle al controlador más que suficiente tiempo para responder a las sondas. Sin embargo, esta no es una buena solución ya que en una red real, cuanto más alto esté ese número, más demoraría el controlador en darse cuenta que un switch se desconectó. Se deja para trabajos futuros investigar si esto podría afectar una despliegue real, ya que si lo hace, tendrían que hacerse cambios radicales a la arquitectura.

3.5.3 Problema de concurrencia por muchas instancias de OpenVSwitch

Como se explica en el Proyecto RRAP [43], para que las interfaces de red de los nodos funcionen como puertos OpenFlow con dirección IP, no solo se debe crear un puerto en Open vSwitch para cada una, sino que también se debe crear una interfaz virtual con su respectivo puerto. Además, deben crearse flujos para que los paquetes que entran por la interfaz física salgan por su respectiva interfaz virtual, y viceversa. El siguiente código simplificado muestra como es el proceso de configuración:

```
ovs-vsctl add-port eth0   #deberia asignar nro de puerto OF 1
ovs-vsctl add-port eth1   #deberia asignar nro de puerto OF 2
ovs-vsctl add-port eth2   #deberia asignar nro de puerto OF 3
ovs-vsctl add-port veth0  #deberia asignar nro de puerto OF 4
ovs-vsctl add-port veth1  #deberia asignar nro de puerto OF 5
ovs-vsctl add-port veth2  #deberia asignar nro de puerto OF 6

ovs-ofctl add-flow in_port=1,output:4
ovs-ofctl add-flow in_port=4,output:1
ovs-ofctl add-flow in_port=2,output:5
ovs-ofctl add-flow in_port=5,output:2
ovs-ofctl add-flow in_port=3,output:6
ovs-ofctl add-flow in_port=6,output:3
```

Cuando se le agrega un puerto, Open vSwitch le asigna automáticamente un número de puerto OpenFlow. La manera en que Open vSwitch hace esa numeración es secuencial, empezando desde 1. Eso quiere decir que, en teoría, si se agregan 3 puertos a Open vSwitch, sus puertos OpenFlow tendrán los números 1, 2 y 3, de acuerdo al orden en que fueron creados. Como se ve en el pseudocódigo, las líneas que agregan los flujos que "conectan" las interfaces virtuales y físicas asumen que la numeración se hace de esa forma. Está configuración se hace para cada nodo (es decir, para cada instancia de Open vSwitch) y es equivalente a la forma en que se configuran los dispositivos físicos en el prototipo.
No se observaron problemas relacionados con esto en topologias chicas de 4 nodos aproxi-

madamente, pero sí en topologias más grandes. Se observó que con más nodos, la numeración de los puertos en Open vSwitch se comportaba de forma impredecible, y no seguía el esquema secuencial que se mencionó anteriormente. Como los flujos que se agregan posteriormente asumen ese determinado orden, eso causa que varias o todas las interfaces del nodo no funcionen. En topologias de entre 10 y 40 nodos ese comportamiento a veces afectaba sólo unos pocos nodos, y en ocasiones no ocurría. Con topologias de 100 nodos aproximadamente, esto ocurría siempre, con más de la mitad de los nodos. Esto llevó a creer que se trataba de un problema de concurrencia por tener muchas instancias de Open vSwitch iniciándose y configurándose al mismo tiempo.
Para solucionar este problema se hizo el siguiente cambio:

```
ovs-vsctl add-port eth0 ofport_request=1
ovs-vsctl add-port eth1 ofport_request=2
ovs-vsctl add-port eth2 ofport_request=3
ovs-vsctl add-port veth0 ofport_request=4
ovs-vsctl add-port veth1 ofport_request=5
ovs-vsctl add-port veth2 ofport_request=6

ovs-ofctl add-flow in_port=1,output:4
ovs-ofctl add-flow in_port=4,output:1
ovs-ofctl add-flow in_port=2,output:5
ovs-ofctl add-flow in_port=5,output:2
ovs-ofctl add-flow in_port=3,output:6
ovs-ofctl add-flow in_port=6,output:3
```

El parámetro opcional **ofport_request** permite indicarle a Open vSwitch que número debería asignarle al puerto que se está agregando. Esto soluciona completamente el problema y permite levantar topologias de cualquier tamaño sin problemas. Es importante remarcar que probablemente esto solo afecta al entorno virtual, y no es necesario realizar esta corrección al configurar nodos reales, ya que cada dispositivo ejecuta una única instancia de Open vSwitch y por lo tanto no es probable que se vea este comportamiento. Sin embargo, no se descarta del todo que exista una posibilidad de que esto ocurra, así que se propone el uso de **ofport_request** para la configuración de los nodos físicos para eliminar ese riesgo, por más pequeño que sea.

3.5.4 Problema de LSDB Sync con muchos nodos

Como se explicó en el capítulo X, el componente llamado LSDB Sync es el encargado de procesar la información de la base de datos topológica de OSPF y enviarla a las aplicaciones que se ejecutan en el controlador. Este proceso se lleva a cabo en un script escrito en Python, el cual se conecta mediante Telnet con el demonio OSPF, extrae la información topológica y la procesa, y luego la envía al controlador. Se observó que si se está simulando una topología grande (de 100 nodos aproximadamente), la ejecución de este script queda colgada, y no

termina. Por lo tanto, la información no se envía nunca al controlador. Para explicar la fuente del problema, es necesario mostrar un pseudocódigo del script:

```
conexion = telnetlib.iniciar_conexion("localhost", PUERTO_OSPFD)
conexion.write(password)
conexion.write("show ip ospf database router")
conexion.write("exit")
info_topologica = conexion.read_all()

procesar_info_topologica
enviar_info_controlador
```

La tarea de conectarse con el demonio OSPF mediante Telnet y extraer la información topológica se hace con una librería llamada Telnetlib. La parte que se observó era problemática es la llamada al método **read_all()**, usada para leer la información que devuelve el demonio. Al existir muchos nodos, la cantidad de información que deberá devolver el método será extensiva. Se calcula que para topologias de 100 nodos aproximadamente, esa información ronda los 92KB. Aunque no se encontró documentación que lo soporte, el diagnóstico que se hizo es que dicho método no está diseñado para devolver tanta información, y quizás queda colgado por limitaciones de memoria o buffers.

A diferencia de read_all, que se bloquea hasta terminar de leer, el método **read_very_eager()** lee toda la información que puede pero sin bloquearse [5]. Con el segundo método, el script no se cuelga, pero sí devuelve información incompleta en ocasiones. Por lo tanto, se adoptó read_very_eager como solución al problema, y se agregó una validación al script que compruebe que la información no está incompleta antes de enviarla al controlador.

Se propone un estudio más profundo sobre este tema para trabajos futuros por dos razones. En primer lugar, no se conoce con certeza la razón del comportamiento, y la solución que se adopta lo resuelve pero parcialmente. En segundo lugar, este defecto podría afectar a despliegues reales de la arquitectura RAUFlow.

3.5.5 Precaución con MTU

A diferencia de los problemas mencionados en las secciones anteriores, en esta sección no se discutirá un problema de la arquitectura ni del entorno virtual, sino de una precaución que se debe tener en cuenta a futuro. Como se explica en el capítulo X, las VPN se implementan agregando y quitando etiquetas MPLS a los paquetes que pasan por los switches. Esas etiquetas son de 5 bytes, y dependiendo el camino que debe recorrer, a un paquete se le puede agregar una o dos etiquetas. Es importante tener esto en cuenta ya que impacta en el MTU. Por ejemplo, si se tiene una red Ethernet con un MTU de 1500 bytes, y se envía tráfico TCP sin tener esto en consideración, probablemente se utilice un MSS (Maximum Segment Size) de 1460 bytes, dejando 20 bytes para el cabezal IP y 20 más para el TCP. En ese caso, el

tráfico no pasa por la red, ya que cuando el switch de borde recibe los paquetes y les asigna las etiquetas MPLS que corresponden, los mismos pasan a tener más de 1500 bytes, y por lo tanto no se envían. Para solucionarlo, se debe reducir el MSS a 1455 en caso de que el camino sea de un salto (se le asigna una etiqueta), y a 1450 en caso contrario (se asignan dos etiquetas).

Capítulo 4

Pruebas de escala

Con el entorno de simulación construido, el siguiente objetivo es realizar pruebas de escala sobre la arquitectura. Se realizan dos tipos de pruebas de escala: (a) verificar cómo funciona la arquitectura para topologias de escala, (b) realizar estudios de escala sobre la cantidad de servicios que soporta. En este capítulo se explicará el propósito de cada prueba, las condiciones bajo las cuales se ejecuta cada una de ellas (topologias, tipos de tráfico, etc) y por último, los resultados que arrojan. Las pruebas fueron realizadas en el servidor del Instituto de Computación, con una máquina virtual Lubuntu 14.04 sobre KVM, con 32GB de RAM y un procesador AMD Opteron 23xx.

4.1 Topologias de escala

El objetivo de estas pruebas es aplicar topologias de diferentes tamaños y características a la arquitectura y estudiar el proceso de creación de las redes privadas virtuales en cada realidad.

Las topologias utilizadas se listan a continuación:

- **Básica**: 4 nodos en topología de full mesh. Es la utilizada en el prototipo físico. Figura 4.1.
- **Chica**: topología arbitraria de 11 nodos (fuente: Topology Zoo). Figura 4.2.
- **Mediana**: topología arbitraria de 45 nodos (fuente: Topology Zoo). Figura 4.3.
- **Grande**: topología de tipo arborescente compuesta por 105 nodos. Figura 4.4

Todas las topologias tienen las siguientes características:

- Tienen un conjunto de RAUSwitch, conectados de acuerdo a lo que dicte la topología.

- Existen dos subredes cliente, implementadas por un QuaggaRouter y un RAUHost cada una (recordar las clases del entorno virtual). El RAUHost representa la computadora que utiliza el usuario final para conectarse a la red, y QuaggaRouter representa el router legacy que conecta la subred del usuario a la red SDN. Los RAUHost serán los remitentes y destinatarios del tráfico que pasará por la red. Esos datos se generarán con el comando *ping* y la herramienta *iperf*. Estas dos subredes tendrán una ubicación variable en cada topología, ya que se probará con distintos caminos entre ellas. Por esta razón, se omiten en las imágenes de las topologias.

- El controlador se conecta con un switch virtual genérico (implementado por la clase Switch de Mininet, en el modo *standalone*), el cual se conecta con los RAUSwitch. Esta será la red de gestión. Por simplicidad, dicha red se omite en las imágenes.

Fig. 4.1 Topología básica

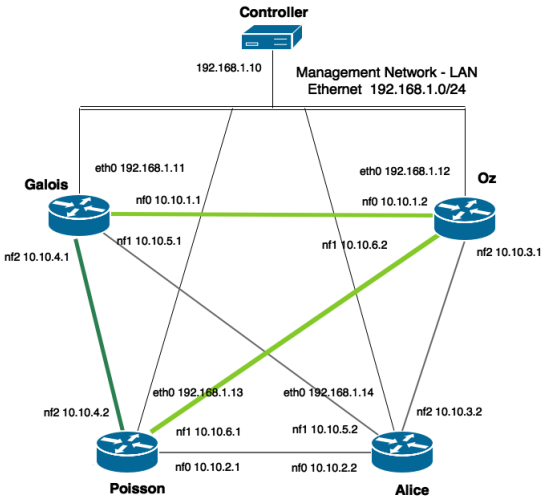


Fig. 4.2 Topología chica

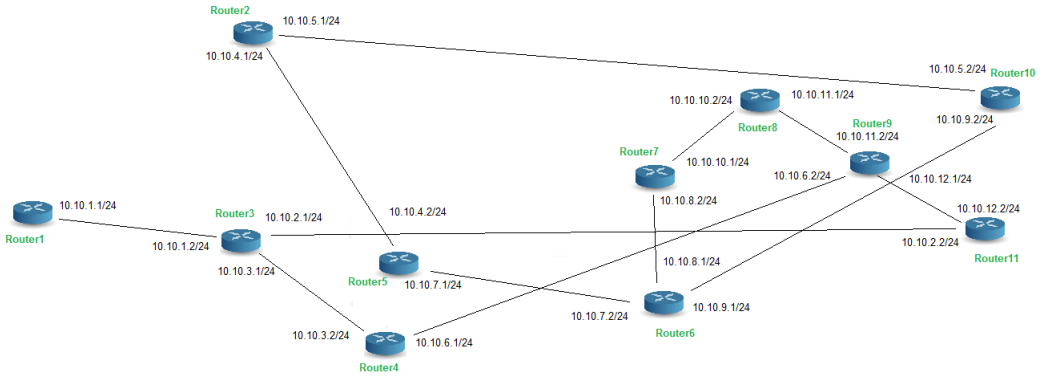


Fig. 4.3 Topología mediana

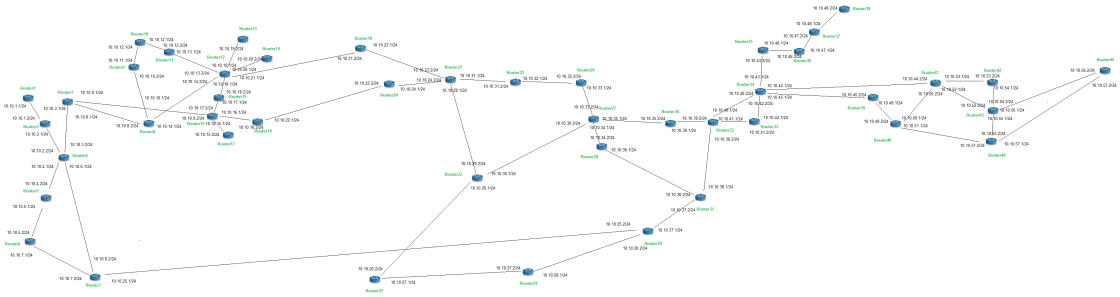
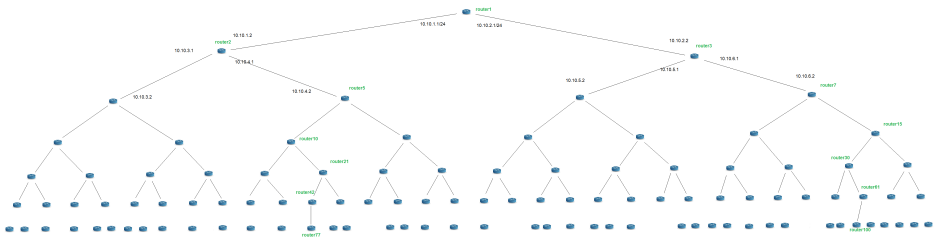


Fig. 4.4 Topología grande



Es importante tener en cuenta que realizar esta serie de pruebas es posible gracias al proceso de verificación funcional explicado en la sección 3.5. En ella se explica la detección y corrección de algunos problemas de escalabilidad tanto del entorno virtual como de la

arquitectura RAUFlow misma. Por lo tanto, estas pruebas se pueden considerar como una continuación del estudio de escalabilidad presentado en dicha sección.

4.1.1 Escenario

El objetivo es estudiar como impacta el tamaño de la topología y el largo del camino en el tiempo que demora la arquitectura en establecer una VPN entre dos subredes cliente. Se espera que ese tiempo sea influenciado en gran medida por dos factores: el tiempo que demora el controlador en calcular el camino óptimo y el tiempo que demora en configurar los flujos en cada nodo del camino. Dado que los servicios se crean enviando pedidos HTTP POST al controlador, el tiempo de creación de los mismos se medirá como el tiempo que demore el controlador en devolver las respuestas HTTP indicando que fueron creados con éxito (esta información está disponible en los logs). Para lograr resultados representativos y reducir el margen de error, en lugar de crear una VPN y medir su tiempo solamente, se realizan cuatro ejecuciones y se calculan las métricas estadísticas relevantes. Para agilizar la ejecución de esta prueba se utiliza un script en Python que manda los pedidos HTTP POST al controlador para crear las VPN, y de este modo no hay necesidad de hacerlo manualmente a través de la interfaz web.

La tabla 4.1 muestra los resultados obtenidos en el escenario. En ella se muestran los tiempos obtenidos para cada topología y largo de camino. Cada caso fue ejecutado cuatro veces, y para cada conjunto de ejecuciones se calcula la media, mediana, desviación estándar y coeficiente de variación (CV).

El primer dato que se puede extraer de los resultados obtenidos es que hay una diferencia de tiempo significativa entre una VPN de capa 2 y una de capa 3. Esto es esperable dado que un servicio de capa 2 debe instalar 42 flujos en cada nodo que compone el camino, porque debe instalar un flujo por cada ethertype posible (esto se explica en la sección 2.6). Por otro lado, un servicio de capa 3 solo debe instalar un flujo en cada nodo del camino.

También se puede observar que los tiempos tienden a incrementarse a medida que el camino por el que pasa la VPN es mayor. Esto se debe principalmente a que mientras más nodos haya en el camino, más flujos deben instalarse. Se observa una diferencia de tiempo más acentuada entre el camino de largo 1 y el de 2. Esto se debe a que cuando la VPN pasa por un camino de largo uno el controlador debe configurar sólo un nivel de etiquetas MPLS, mientras que si el camino es de largo mayor, corresponden dos niveles de etiquetas. Esto requiere más tiempo de computo, e instalar al menos un flujo adicional.

Si se comparan los resultados entre cada topología, también hay observaciones importantes. En primer lugar, se puede ver que para caminos de igual largo, si la topología es más

Table 4.1 Tiempo que demora el controlador en dar de alta VPNs. El tiempo se mide en ms. Los valores de color marrón corresponden a la VPN de capa 2, y los azules a la de capa 3.

Topología	Largo del camino	Ejecución 1 (ms)	Ejecución 2 (ms)	Ejecución 3 (ms)	Ejecución 4 (ms)	Media (ms)	Mediana (ms)	Desv. Estándar (ms)	CV
Básica	1	188 / 14	188 / 23	199 / 15	188 / 15	191 / 17	188 / 15	6 / 4	0.03 / 0.24
Chica	1	155 / 29	208 / 22	351 / 19	220 / 18	234 / 22	214 / 21	83 / 5	0.35 / 0.23
	2	241 / 25	309 / 27	317 / 31	317 / 29	296 / 28	313 / 28	37 / 3	0.13 / 0.11
	4	269 / 52	298 / 42	327 / 43	301 / 39	299 / 44	300 / 43	24 / 6	0.08 / 0.14
	6	257 / 42	320 / 55	348 / 56	326 / 57	313 / 53	323 / 56	39 / 7	0.12 / 0.13
	8	285 / 51	333 / 64	337 / 64	342 / 60	324 / 60	335 / 62	26 / 6	0.08 / 0.10
Mediana	1	254 / 101	318 / 92	355 / 110	318 / 110	311 / 103	318 / 106	42 / 9	0.14 / 0.09
	2	567 / 104	746 / 134	520 / 112	611 / 127	611 / 119	589 / 120	97 / 14	0.16 / 0.12
	4	372 / 129	504 / 105	498 / 135	438 / 145	453 / 129	468 / 132	62 / 17	0.14 / 0.13
	6	364 / 126	397 / 187	675 / 164	529 / 153	491 / 158	463 / 159	142 / 25	0.29 / 0.16
	8	436 / 130	515 / 271	529 / 202	787 / 179	567 / 196	522 / 191	152 / 59	0.27 / 0.30
	10	414 / 181	548 / 197	485 / 214	477 / 191	481 / 196	481 / 194	55 / 14	0.11 / 0.07
	12	441 / 289	519 / 152	613 / 154	571 / 178	536 / 193	545 / 166	74 / 65	0.14 / 0.34
Grande	1	425 / 483	940 / 325	573 / 581	428 / 282	592 / 418	501 / 404	242 / 139	0.41 / 0.33
	2	642 / 810	457 / 223	1044 / 319	2671 / 1346	1204 / 675	843 / 565	1009 / 516	0.99 / 0.76
	4	1692 / 1000	2457 / 2209	1543 / 1070	1613 / 668	1826 / 1237	1653 / 1035	425 / 672	0.23 / 0.54
	6	1669 / 722	1254 / 1016	1048 / 476	2678 / 569	1662 / 696	1462 / 646	725 / 236	0.44 / 0.34
	8	1516 / 2182	2273 / 608	3482 / 856	3496 / 748	2692 / 1099	2878 / 802	972 / 729	0.36 / 0.66
	10	493 / 449	775 / 638	784 / 563	966 / 570	755 / 555	780 / 567	195 / 78	0.26 / 0.14
	12	1843 / 594	3438 / 1422	1791 / 1018	2848 / 849	2480 / 971	2346 / 934	803 / 348	0.32 / 0.36

grande entonces se necesita más tiempo para crear la VPN. Ese comportamiento se explica con los siguientes puntos:

- Si la topología tiene más nodos, entonces el algoritmo Dijkstra que calcula el camino óptimo va a demorar más tiempo en converger. (poner referencia o explicar)
- A medida que hay más RAUSwitch en la topología, la red de gestión tiene más tráfico por la mensajería generada por OpenFlow. Esto puede llevar a congestión en dicha red. Los flujos correspondientes a cada nodo son instalados mediante mensajes OpenFlow que envía el controlador a través de la red de gestión, y si esos mensajes experimentan retrasos o pérdidas, entonces eso significa un retraso en la creación de la VPN. Este fenómeno también se explora en la sección 3.5.2.
- Como se está usando un emulador, tener más nodos implica que el procesador anfitrión debe repartir el tiempo de cómputo entre más procesos. Naturalmente, esto resulta en una lentitud generalizada que afecta a toda la red virtual. A diferencia de los puntos anteriores, este fenómeno no se observaría en un despliegue real de la red, así que es relevante solo en este ambiente de pruebas.

Los dos últimos puntos también ayudan a explicar otro detalle importante que muestran las tablas: el coeficiente de variación (CV). Se puede observar que este valor es mayor a medida que crece la topología, y que incluso llega a 0.99 (o 99%) cuando se realizaron las ejecuciones en la topología grande. Esto implica que a medida que la topología crece, la variabilidad de los tiempos que se miden es mayor, y por lo tanto son menos confiables. El segundo punto puede explicar esto porque si hay congestión en la red, entonces la variabilidad del RTT (round-trip time) va a ser mayor (referencia o explicacion?). El tercer punto también puede influir en un CV más alto, ya que es posible que una sobrecarga en la CPU introduzca variabilidad al tiempo de CPU que recibe cada proceso, o quizás también una variabilidad en las tareas que puede desempeñar un proceso dado un determinado tiempo de CPU.

Con el objetivo de hacer un análisis más fino sobre el tiempo de creación de las VPNs, se repitieron las pruebas pero agregando modificaciones al código del controlador que permitan analizar cuánto tiempo dedica a cada parte del proceso. Se identificaron tres partes fundamentales: el cálculo del camino óptimo, todo lo relacionado al cálculo y manejo de las etiquetas MPLS, y la instalación de los flujos OpenFlow. La ejecución de esas tres tareas abarca la mayoría (más del 90%) del tiempo de creación de una VPN. Las modificaciones en el código consistieron de agregar *timestamps*[1] en las partes claves del código. Luego,

[1] Marca temporal que indica la hora en la que se lleva a cabo un evento.

analizando el log del controlador se hace la resta de los diferentes timestamps para determinar cuánto tiempo se tomó para cada tarea. Es importante tener en cuenta que este método tiene un margen de error no despreciable, ya que está a la merced del tiempo de CPU que se le asigna al controlador. De hecho, algunas ejecuciones fueron descartadas de los resultados ya que se alejan demasiado del resto, probablemente porque la CPU fue asignada durante la creación de una VPN a otros procesos por demasiado tiempo.

Las gráficas en las figuras 4.5, 4.6 y 4.7 a continuación muestran, para cada topología, cómo se descompone el tiempo de ejecución para cada largo de camino. Se omite la de la topología básica ya que sólo permite crear VPNs de largo 1. Cada punto en la gráfica indica el porcentaje del tiempo total que se necesitó para llevar a cabo una de las tres tareas ya mencionadas (cálculo del camino, manejo de etiquetas, instalación de flujos OpenFlow). Cada porcentaje se mide en base al promedio de 8 ejecuciones (descartando las ejecuciones que arrojan valores muy alejados del resto).

Fig. 4.5 Distribución del tiempo de creación de VPNs en la topología chica

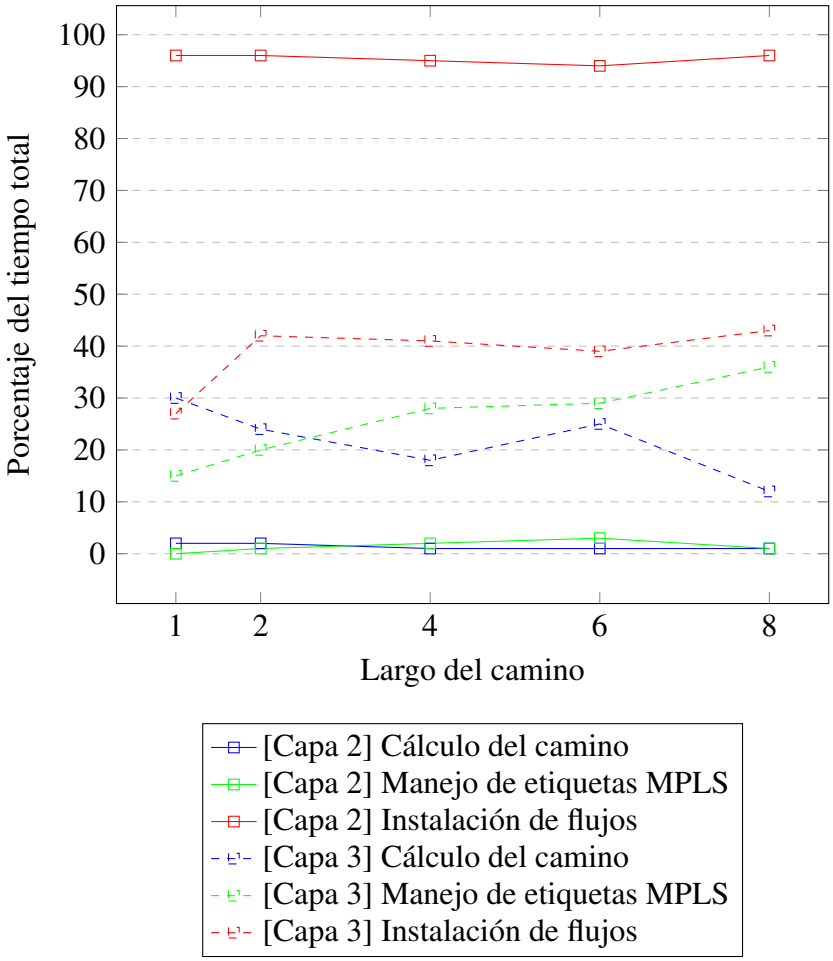


Fig. 4.6 Distribución del tiempo de creación de VPNs en la topología mediana

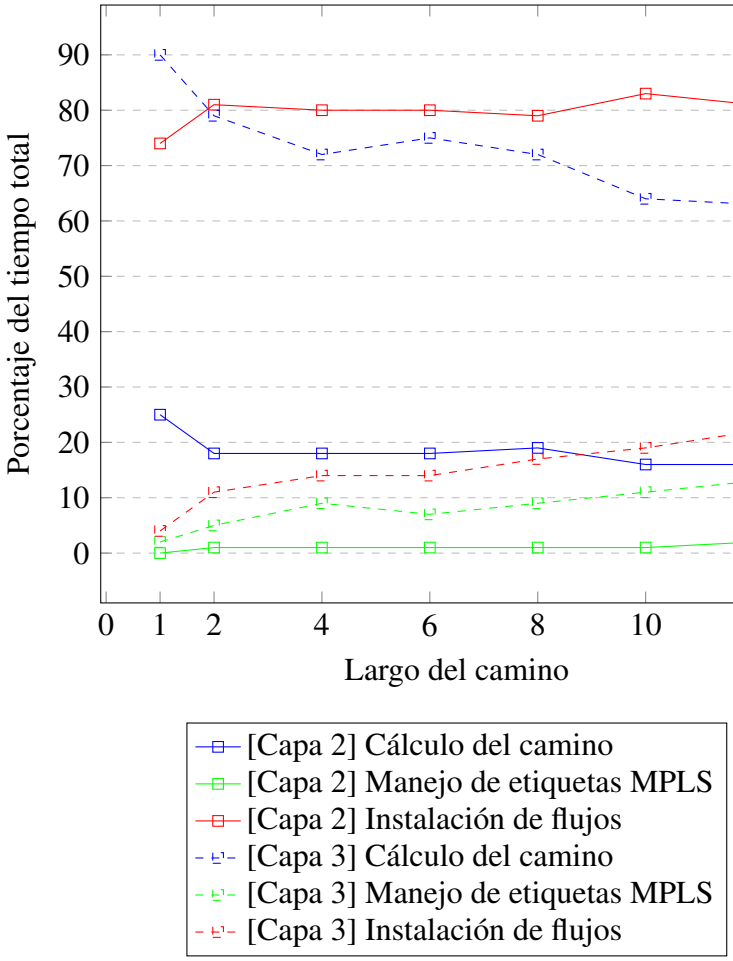
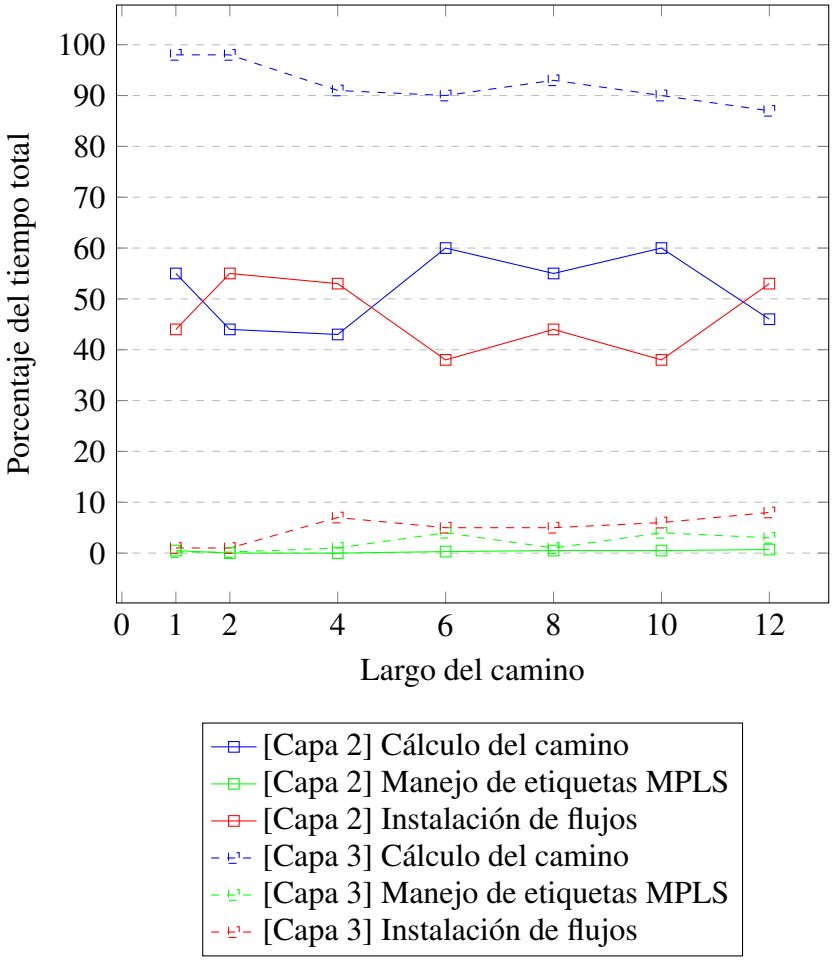


Fig. 4.7 Distribución del tiempo de creación de VPNs en la topología grande



Las líneas rojas muestran como evoluciona el porcentaje del tiempo que abarca la instalación de flujos. Se observa que la porción de tiempo requerida para la instalación de los flujos de una VPN de capa 2 (denotada por las líneas rojas continuas) queda siempre por encima de la requerida para instalar flujos de una VPN de capa 3 (líneas rojas punteadas). Esto se debe a que, como ya fue explicado anteriormente, una VPN de capa 2 debe instalar más flujos en los nodos, lo cual implica que se debe dedicar más tiempo a la etapa de instalación de flujos. Para ambos tipos de VPN se observa que la instalación de flujos abarca más tiempo de ejecución (las líneas suben) a medida que el camino es más largo. Esto es esperable ya que más nodos implican más flujos. Curiosamente, esta suba parece ser más pronunciada para la VPN de capa 3.

Las líneas verdes muestran qué porcentaje del tiempo total se dedica al cálculo y manejo de etiquetas MPLS. Igual que la instalación de flujos, esta etapa requiere más tiempo a medida que hay más nodos en el camino, ya que se aumenta la cantidad de etiquetas MPLS.

También se observa que la VPN de capa 3 requiere un mayor porcentaje del tiempo que la de capa 2 para esta tarea. Esto puede resultar extraño, ya que la de capa 2 debe manejar más etiquetas. Lo que ocurre es que la de capa 2 requiere más tiempo absoluto, pero el tiempo total de creación de la VPN también es mayor, por lo tanto el porcentaje del tiempo que se dedica a la tarea resulta mayor para la VPN de capa 3. Otra observación es que esta tarea es la que menos tiempo requiere de las tres.

Por último, el cálculo del camino óptimo. Lo primero a tener en cuenta es que esta etapa no es afectada por el tipo de VPN que se está creando, ya que lo único que debe hacer es determinar el camino óptimo entre dos nodos de un grafo. Tampoco es afectada por el largo del camino que está calculando. Lo único que la afecta es el tamaño de la topología. Las líneas azules denotan la participación del cálculo del camino óptimo en la distribución del tiempo. Lo primero que se observa es que el porcentaje de tiempo dedicado a esta tarea decrece a medida que se aumenta el largo del camino. Esto es esperable, ya que el tiempo requerido tiende a ser constante, mientras que las otras tareas requieren más tiempo, por lo tanto su porción del tiempo disminuye. Otra observación interesante es que a medida que se aumenta el tamaño de la topología, esta etapa ocupa una mayor parte del tiempo de ejecución. En la topología chica la etapa de instalación de flujos se lleva la mayor parte del tiempo de ejecución tanto para la VPN de capa 2 como la de capa 3. Sin embargo, en la topología mediana eso cambia, y se cumple sólo para la VPN de capa 2. Por último, en la topología grande, ya no se observa eso para ninguno de los dos tipos, y la VPN de capa 2 reparte de forma bastante equitativa el tiempo de ejecución entre el cálculo del camino y la instalación de flujos.

Desde el punto de vista técnico esto también es esperable, ya que el algoritmo que calcula el camino es de orden cuadrático en la cantidad de nodos, mientras que las otras etapas (manejo de etiquetas e instalación de flujos OpenFlow) aumentan linealmente con la cantidad de nodos en el camino. Esto lleva a creer que si se siguiera experimentando con topologias aún más grandes, el cálculo del camino óptimo eventualmente abarcaría la mayoría del tiempo de creación de las VPNs, tanto de capa 2 como de capa 3.

4.2 Escala de servicios y flujos

Entre los requerimientos de la RAU2 se encuentra el de la escalabilidad de usuarios. Específicamente, se espera alcanzar en un mediano plazo un total de 11.000 docentes, 7.000 funcionarios y 140.000 estudiantes (de acuerdo a los requerimientos relevados por el proyecto RRAP). Esto implica que la red será sujeta a importantes cantidades de servicios y flujos distintos. Para poder asegurar que la arquitectura escala sin problemas, se la debería someter

a experimentos que reflejen los niveles de actividad y tráfico que son esperados en un futuro. Sin embargo, dichos experimentos no son llevados a cabo por este trabajo por dos razones. En primer lugar, se considera que es una tarea que lleva demasiado tiempo como para ser incluida dentro del alcance de este trabajo. En segundo lugar, no están disponibles los modelos y patrones de tráfico que serían necesarios para esas pruebas.

También es importante recordar que el entorno virtual no generará resultados relevantes en lo que refiere al nivel real de rendimiento de la arquitectura. Recordar sección 3.3.1, donde se explica que cada instancia de Open vSwitch se ejecuta en modo user-space, y por ende procesa los paquetes de forma bastante lenta.

Se puede concluir entonces que las pruebas detalladas a continuación no aseguran la escalabilidad de RAUFlow, pero sí ayudan a investigar el comportamiento de la red y los dispositivos cuando se los somete a muchos servicios.

4.2.1 Descripción del escenario

El escenario de prueba consiste en crear una cantidad relativamente grande de VPNs en la topología básica, y analizar si esto degrada el rendimiento de la red de alguna forma. Se utiliza una VPN punto a punto de capa 3 para conectar dos subredes cliente, y se utiliza *iperf* para generar tráfico TCP y medir el ancho de banda entre los dos hosts. Para cargar al controlador con servicios, se crean múltiples VPN de capa 2 entre las subredes, variando los valores de los cabezales VLAN_ID y VLAN_PCP (pudiendo crear un total de 32.768 combinaciones distintas) para que toda VPN sea distinta de las demás. De esta forma, existirán múltiples VPN pero solo una (la de capa 3) será utilizada con tráfico.
Dado que cargar todas las VPN a mano en la interfaz web llevaría demasiado tiempo, se creó un servicio web que recibe como parámetro la cantidad de VPN que se desean. Cuando se hace un pedido GET a ese servicio web, se inicia el proceso de creación de las mismas. Este proceso puede tomar entre algunos minutos y varias horas, dependiendo de la cantidad.

Con este escenario de prueba se busca verificar los dos siguientes aspectos claves:

Escalabilidad interna del RAUSwitch
Se estudian posibles limitaciones internas que puedan tener los dispositivos cuando deben manejar grandes cantidades de flujos. Es posible que a medida que crecen sus tablas de flujos, demoren más en encontrar el flujo que se corresponde con cada paquete que reciben. Si pasa esto, el throughput debería ser afectado negativamente por la cantidad de flujos en sus tablas.
Escalabilidad en servicios
Se estudian posibles problemas que puedan tener la arquitectura de la red o, en particular,

la aplicación RAUFlow para manejar grandes cantidades de servicios o información. Es de particular interés medir la memoria que requiere el controlador para mantener los servicios.

4.2.2 Resultados y observaciones

Table 4.2 Throughput en Kbits/s medidos para distintas cantidades de VPNs.

# de VPNs	Throughput (Kbits/s)
1	893
3000	887
6000	887
9000	890
12000	885
15000	886

Como se explica en la descripción del escenario, se busca determinar si la existencia de muchos flujos en la tabla implica que un RAUSwitch demora más tiempo en encontrar el flujo OpenFlow que corresponde para un paquete entrante, y por lo tanto demora más en determinar la acción a tomar para ese paquete. Si esto fuera así, debería existir una relación inversamente proporcional entre la cantidad de flujos en la tabla de un nodo y su velocidad para procesar paquetes. En la tabla 4.2 se pueden observar los throughput promedio medidos para un flujo de datos sobre la topología básica, con distintas cantidades de VPN existiendo en la red. La principal conclusión que se puede sacar de la tabla es que el throughput no es afectado por la cantidad de VPNs existentes en el momento (se asume que las pequeñas diferencias numéricas entran en el margen de error).

Se puede ver que la máxima cantidad de VPNs con la que se probó fue de 15.000. Cada VPN de capa 2 está compuesta por dos servicios de capa 2, y cada uno de esos servicios introduce 42 flujos en cada nodo del camino. Esto quiere decir que cada uno contiene alrededor de 1.260.000 (15.000 * 2 * 42) flujos en su tabla. Se podría argumentar que hacen falta más flujos para impactar el throughput, pero en realidad la explicación se encuentra en la especificación de Open vSwitch, y ya fue mencionada en el capítulo 3. Open vSwitch realiza cacheo de flujos. Eso quiere decir que cuando un paquete de datos de un determinado flujo llega por primera vez a un nodo, este paquete es enviado al pipeline de OpenFlow para determinar qué acción se debe tomar. Luego de realizada, esta acción es escrita en la caché, y tiene un tiempo de vida de entre 5 y 10 segundos. Si en ese período de tiempo llega otro

paquete del mismo flujo, no hay necesidad de enviar el paquete al pipeline, porque ya se sabe cuales son las acciones a tomar para el mismo. Por lo tanto, si un flujo de datos es constante y rápido, el tamaño de la tabla de OpenFlow no afectará el tiempo de decisión, ya que sólo el primer paquete de ese flujo deberá ser comparado con dicha tabla.

Mediante el comando 'ovs-appctl dpctl/show' de Open vSwitch, se puede examinar las estadísticas de la cache de cada nodo. A modo de ejemplo, en la figura 4.8 se observan las estadísticas de un nodo llamado 'alice'. En la sección 'lookups' se detallan cuantos 'hits' y 'miss' de caché han ocurrido hasta el momento, y 'flows' indica cuantos flujos activos hay en el momento en la caché.

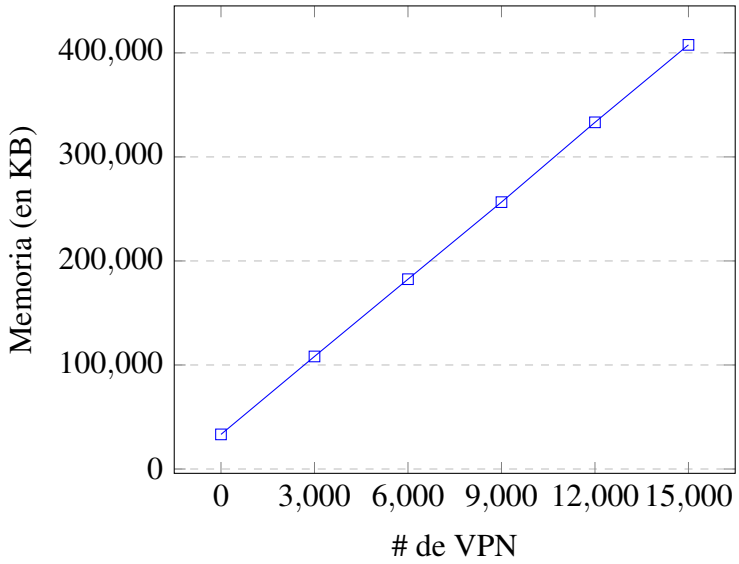
Fig. 4.8 Estadísticas de cache de flujos del nodo 'alice'.

Otro objetivo de la prueba es determinar si la arquitectura, y en particular el controlador, tienen algún problema para manejar muchos servicios. No se ha detectado ningún problema de esa índole. Sin embargo, es importante recordar que el controlador mantiene toda su información en memoria, por lo tanto es de interés realizar un estudio del consumo de memoria del mismo a medida que crece la cantidad de servicios. El comando de Linux llamado *pmap* permite estudiar el consumo de memoria del proceso que se le indique, y con el mismo se puede analizar la evolución del consumo de memoria del controlador a medida que se le agregan servicios. En la tabla 4.3 y la figura 4.9 se puede observar el resultado de estas mediciones.

Table 4.3 Evolución del consumo de memoria del controlador.

# de VPNs	Memoria (KB)
0	33280
3000	108196
6000	182460
9000	256528
12000	333244
15000	407696

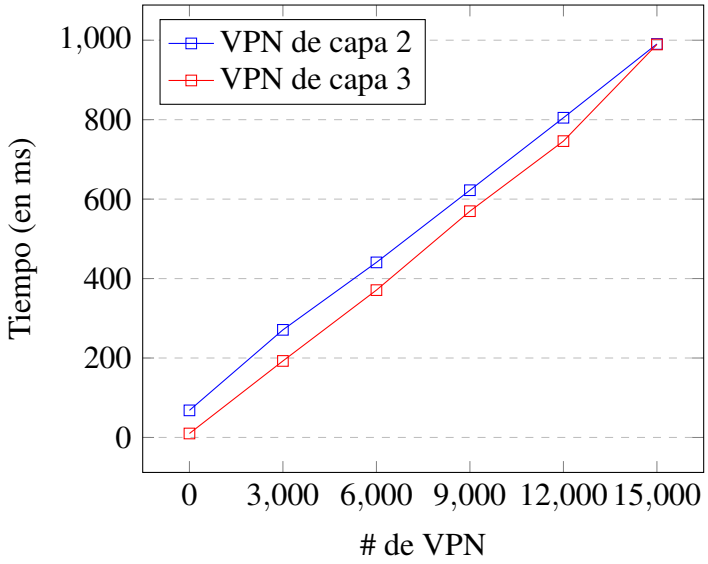
Fig. 4.9 Efecto de la cantidad de VPN sobre la memoria consumida



La principal observación que se puede hacer es que el consumo de memoria del controlador aumenta de forma lineal con la cantidad de servicios. El mismo se incrementa en 74916 KB cada 3000 servicios, por lo tanto se puede calcular que cada servicio ocupa alrededor de 25 KB (74916/3000). A modo de ejemplo, si extrapolamos ese número a una computadora que puede dedicar 4 Gb de RAM para mantener los servicios, llegamos a que el controlador podrá mantener alrededor de 167.772 servicios. A pesar de que no es ideal mantener tantos datos en memoria, se puede concluir que es un consumo aceptable.

En el proceso de realizar las pruebas ya mencionadas también se observó un comportamiento que no se esperaba. Se detectó que a medida que hay más VPNs creadas, la red demora más tiempo en crear una nueva VPN. Con el propósito de entender más ese comportamiento, se hizo un experimento cuyos resultados se observan en la figura 4.10.

Fig. 4.10 Efecto de la cantidad de VPNs sobre el tiempo de carga de una VPN nueva



Cada punto indica el tiempo que demora la red en dar de alta una nueva VPN con una determinada cantidad de VPN ya existentes. Estos tiempos se miden de la misma forma que los obtenidos en la sección de pruebas anterior: se toma el tiempo que demora la aplicación en devolver la respuesta HTTP indicando que el servicio se creó con éxito (disponible en los logs). En la gráfica se puede ver que el tiempo de carga aumenta de forma lineal a medida que hay más VPN en la red, y esto se cumple para la de capa 2 como la de capa 3. Una posible explicación inicial para esto puede ser que al tener más flujos, cada nodo demora más en insertar nuevos flujos en su tabla. Esa teoría se descarta con el siguiente razonamiento. En la gráfica se observa que toma más tiempo crear una VPN de capa 2 que una de capa 3. Esto en gran medida se explica porque un servicio de capa 2 debe insertar 42 flujos en los nodos del camino, mientras que uno de capa 3 solo inserta 1. Pero también se observa que las lineas son virtualmente paralelas, es decir, esa diferencia de tiempo se mantiene constante a pesar de las VPNs que existan. Si insertar un flujo nuevo cada vez tomara más tiempo, ese incremento se debería multiplicar por 42 para la VPN de capa 2, y la linea correspondiente a la VPN de capa 2 debería tener una pendiente más inclinada que la de capa 3.

Otra posible explicación para este comportamiento, y quizás la más probable, es que la aplicación RAUFlow se vuelve más lenta a medida que sus estructuras de datos crecen. Sin embargo, no se observa ninguna característica en el código que indique esto.

Capítulo 5

Conclusiones

Se realizó una investigación del estado del arte de las diferentes aplicaciones que se le puede dar al paradigma SDN, prestando especial atención a las diferentes maneras de implementar redes privadas virtuales. También se hizo una investigación profunda sobre las diferentes opciones de virtualización para las redes definidas por software.

Tomando como base al popular emulador Mininet, se construyó una herramienta que permite emular dispositivos que pueden actuar como switches OpenFlow y al mismo tiempo correr protocolos distribuidos legados como OSPF. Esta herramienta permite la virtualización de la arquitectura RAUFlow, pero también se la puede ver como un resultado valioso incluso afuera del contexto de RAUFlow, ya que hasta el momento de realizar este trabajo no se encontró ninguna herramienta que tenga esas capacidades. Se creó un repositorio público en Github [26] con el código fuente del emulador con la intención de hacerlo disponible para la comunidad. También se produjo un manual de usuario para facilitar el uso futuro de la herramienta.

Se trabajó en una verificación funcional del entorno construido, que permitió detectar y corregir problemas, y validar el correcto funcionamiento del mismo con diversas topologias. Dicha verificación también permitió corregir dos defectos en el código fuente de la aplicación RAUFlow, y detectar algunos posibles problemas de escalabilidad de la arquitectura. También se implementó una mejora a RAUFlow, que permite eliminar la necesidad de agentes SNMP en cada RAUSwitch, reduciendo la complejidad y posiblemente aumentando el rendimiento de los mismos.

Usando la herramienta construida se realizó una serie de pruebas para estudiar la escalabilidad de RAUFlow. En primer lugar, mediante diferentes topologias de prueba se estudió el tiempo requerido para la creación de VPNs. Para profundizar el análisis, se estudió la distribución del tiempo de ejecución entre las principales tareas que componen dicho proceso de creación. En segundo lugar, se llevó a cabo una serie de pruebas que estudian el

comportamiento de RAUFlow y los RAUSwitch cuando existen muchos servicios. Desde el punto de vista de los RAUSwitch, un estudio a fondo de la herramienta Open vSwitch determinó que la existencia de muchos servicios no afecta el rendimiento de los mismos. Desde el punto de vista del controlador, se verificó que no tiene problemas para mantener muchos servicios, y se estudió la evolución de su consumo de memoria.

Por último, este trabajo contribuyó a la realización de una publicación científica llamada "RAUflow: building Virtual Private Networks with MPLS and OpenFlow" [Santiago Vidal], la cual fue presentada recientemente en la conferencia ACM SIGCOMM Workshop on Fostering Latin-American Research in Data Communication Networks (LANCOMM 2016). La publicación también ha sido aceptada en formato poster en Spring School on Networks (SSN 2016) a llevarse a cabo en noviembre.

5.1 Trabajo futuro

Si bien en este trabajo se ha desarrollado un emulador completamente funcional, aún existe lugar para mejorarlo. Por ejemplo, la configuración de nuevas topologias se debe hacer mediante scripts en Python. Si bien la API para hacerlo es relativamente simple, la tarea puede ser tediosa si se trata de topologias grandes. Existe una propuesta de interfaz gráfica para Mininet llamada MiniEdit [13] que puede ser utilizada con el propósito de mejorar la usabilidad del entorno virtual. Es probable que sea necesario modificar MiniEdit para que funcione con el entorno construido, pero esto no debería ser una tarea muy compleja ya que el código fuente está disponible y bien documentado.

El entorno construido también puede ser mejorado para proporcionar experimentos más reales, o fieles a la realidad. Existe una extensión de Mininet llamada Mininet-HiFi [53] que apunta a mejorar Mininet para proveer experimentos más realistas y reproducibles. Utiliza conceptos como límites de CPU, control de tráfico y aislamiento de recursos para intentar que el experimento que se lleve a cabo sea lo más fiel posible al mismo escenario pero con hardware real. Otra posible línea de trabajo con el objetivo de aumentar el realismo de los experimentos es investigar la herramienta Open vSwitch para encontrar maneras de hacer que múltiples switches OpenFlow en distintos network namespaces puedan procesar paquetes a nivel del kernel. Esto aumentaría sustancialmente la velocidad de los switches que ofrece el entorno virtual actualmente, y ayudaría a que los switches emulados se asemejen más a los físicos.

Se han hecho diversas pruebas sobre la arquitectura RAUFlow que muestran que posee una buena escalabilidad. Sin embargo, es necesario hacer más pruebas para poder asegurarlo. Desde el punto de vista de la escalabilidad en las topologias, en la sección 3.5 se detallaron

algunos problemas funcionales que se sospecha pueden afectar un despliegue real de la arquitectura, y por ende se sugiere una investigación a fondo sobre los mismos. Desde el punto de vista de la escalabilidad en el uso, no se hicieron pruebas con modelos de tráfico que reflejen el volumen y características del tráfico que se espera en la red. Esto es un paso clave en la determinación de la escalabilidad de RAUFlow.

Sería interesante implementar nuevas funcionalidades sobre la arquitectura RAUFlow que incorporen nociones como Ingeniería de tráfico, Quality of Service y seguridad de la red.

Se ha mencionado que el controlador de RAUFlow almacena todos sus datos en memoria. Implementar un método de persistencia que no dependa de la memoria del sistema podría mejorar la escalabilidad y también la robustez, ya que el controlador se podría recuperar ante fallas.

También se discutió que mantener comunicación con demasiados switches OpenFlow puede ser un problema para el controlador, lo cual presentaría un serio problema de escalabilidad. La existencia de un único centro de cómputo para el control de la red también puede generar un efecto cuello de botella. Estos problemas pueden ser solucionados utilizando múltiples controladores, y estableciendo una relación de jerarquía entre ellos.

References

[1] (2014). Network simulators and virtualization - the list. url: http://nil.uniza.sk/network-simulation-and-modelling/network-simulators-list/, último acceso Mayo 2016.

[2] (2016). Cbench. url: https://github.com/mininet/oflops/tree/master/cbench.

[3] (2016). Cloonix. url: http://cloonix.fr/build-30/html/index.html, último acceso Julio 2016.

[4] (2016). Controlador ryu. url: https://osrg.github.io/ryu/, último acceso Setiembre 2016.

[5] (2016). Documentación de Telnetlib. url: https://docs.python.org/2/library/telnetlib.html, último acceso Mayo 2016.

[6] (2016). Extensión openflow para ns-3. url: https://www.nsnam.org/docs/release/3.13/models/html/openflow-switch.html, último acceso Julio 2016.

[7] (2016). Flowsim. url: https://flowsim.flowgrammable.org/, último acceso Julio 2016.

[8] (2016). Gns3. url: https://www.gns3.com/, último acceso Julio 2016.

[9] (2016a). Manual de ovs-appctl. url: http://openvswitch.org/support/dist-docs/ovs-appctl.8.txt, último acceso Mayo 2016.

[10] (2016b). Manual de ovs-dpctl. url: http://openvswitch.org/support/dist-docs/ovs-dpctl.8.txt, último acceso Mayo 2016.

[11] (2016c). Manual de ovs-ofctl. url: http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt, último acceso Mayo 2016.

[12] (2016d). Manual de ovs-vsctl. url: http://openvswitch.org/support/dist-docs/ovs-vsctl.8.txt, último acceso Mayo 2016.

[13] (2016a). Miniedit - interfaz gráfica para mininet. url: https://github.com/mininet/mininet/blob/master/examples/miniedit.py, último acceso Agosto 2016.

[14] (2016b). Mininet, sitio web oficial. url: http://mininet.org/, último acceso Mayo 2016.

[15] (2016c). miniNExT, extensión de MiniNet. url: https://github.com/USC-NSL/miniNExT, último acceso Agosto 2015.

[16] (2016). Mln. url: http://mln.sourceforge.net/index.php, último acceso Julio 2016.

[17] (2016a). Netkit. url: http://wiki.netkit.org/index.php/Main_Page, último acceso Julio 2016.

[18] (2016a). Open networking foundation. url: https://www.opennetworking.org/about/onf-overview, último acceso Setiembre 2016.

[19] (2016). Open Source Network Simulators. url: http://www.brianlinkletter.com/open-source-network-simulators/, último acceso Mayo 2016.

[20] (2016e). The open vswitch database management protocol. url: https://tools.ietf.org/html/rfc7047, último acceso Setiembre 2016.

[21] (2016a). Openflow 1.3.1 especificación. url: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf, último acceso Setiembre 2016.

[22] (2016b). Openflow 1.5 especificación. url: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf, último acceso Setiembre 2016.

[23] (2016f). ovs-vswitchd - open vswitch daemon. url: http://openvswitch.org/support/dist-docs/ovs-vswitchd.8.txt, último acceso Setiembre 2016.

[24] (2016g). ovsdb-server - open vswitch database server. url: http://openvswitch.org/support/dist-docs/ovsdb-server.1.txt, último acceso Setiembre 2016.

[25] (2016b). Protocolo netlink. url: http://man7.org/linux/man-pages/man7/netlink.7.html, último acceso Setiembre 2016.

[26] (2016). Repositorio con el código del entorno virtual. url: https://github.com/santiagovidal/P2015_44, último acceso Mayo 2016.

[27] (2016). Shadow. url: https://shadow.github.io/, último acceso Julio 2016.

[28] (2016b). Software-defined networking (sdn) definition. url: https://www.opennetworking.org/sdn-resources/sdn-definition, último acceso Setiembre 2016.

[29] (2016a). Spirent testcenter openflow controller emulation. url: http://www.spirent.com/-/media/Datasheets/Broadband/PAB/SpirentTestCenter/Spirent_TestCenter_OpenFlow_Controller_Emulation_datasheet.pdf.

[30] (2016b). Spirent testcenter openflow switch emulation. url: http://www.spirent.com/-/media/Datasheets/Broadband/PAB/SpirentTestCenter/SpirentTestCenter_OpenFlow-Switch-Emulation_datasheet.pdf.

[31] (2016). STS - SDN Troubleshooting System. url: http://ucb-sts.github.io/sts/, último acceso Mayo 2016.

[32] (2016). The Internet Topology Zoo. url: http://topology-zoo.org/, último acceso Mayo 2016.

[33] (2016). Vnx. url: http://web.dit.upm.es/vnxwiki/index.php/Main_Page, último acceso Julio 2016.

[34] (2016). What's software defined networking (sdn)? definition. url: https://www.sdxcentral.com/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/, último acceso Setiembre 2016.

[35] Arup Raton Roy, Faizul Bari, M. F. Z. R. A. R. B. (2014). Design and management of dot: A distributed openflow testbed.

[36] Behzad Mirkhanzadeh, Naeim Taheri, S. K. (2016). Sdxvpn: A software-defined solution for vpn service providers.

[Ben Pfaff] Ben Pfaff, Justin Pettit, T. K. E. J. J. A. Z. J. R. J. G. A. W. J. S. P. S. K. A. M. C. The design and implementation of open vswitch.

[38] Bob Lantz, Brandon Heller, N. M. (2010). A network in a laptop: Rapid prototyping for software-defined networks.

[39] Charalampos Rotsos, Nadi Sarrar, S. U. R. S. y. A. W. M. (2012). Oflops: An open framework for openflow switch evaluation.

[40] Denis Salopek, Valter Vasi, M. Z. M. M. M. V. V. K. (2014). A network testbed for commercial telecommunications product testing.

[41] Diego Kreutz, Fernando M. V. Ramos, P. E. V. C. E. R. S. A. S. U. (2015). Software-defined networking: A comprehensive survey.

[42] Dominik Klein, M. J. (2013). An openflow extension for the omnet++ inet framework.

[43] E. Viotti, R. A. (2015). Routers reconfigurables de altas prestaciones.

[44] Ian F. Akyildiz, Ahyoung Lee, P. W. M. L. W. C. (2014). A roadmap for traffic engineering in sdn-openflow networks.

[45] Jeff Ahrenholz, Claudiu Danilov, T. R. H. J. H. K. (2008). Core: A real-time network emulator.

[46] José Teixeira, Gianni Antichi, D. A. A. D. C. S. G. A. S. (2013). Datacenter in a box: test your sdn cloud-datacenter controller at home.

[47] M. Said Seddiki, Muhammad Shahbaz, S. D. S. G. M. P. N. F. Y.-Q. S. (2014). Flowqos: Qos for the rest of us.

[48] Marcel Großmann, S. J. S. (2013). Auto-mininet: Assessing the internet topology zoo in a software-defined network emulator.

[49] Mirchev, A. (2015). Survey of concepts for qos improvements via sdn.

[50] Mukta Gupta, Joel Sommers, P. B. (2013). Fast, accurate simulation for sdn prototyping.

[51] Nick McKeown, Tom Anderson, H. B. G. P. L. P. J. R. S. S.-J. T. (2016). Openflow: Enabling innovation in campus networks.

[52] Niels L. M. van Adrichem, Christian Doerr, F. A. K. (2014). Opennetmon: Network monitoring in openflow software-defined networks.

[53] Nikhil Handigol, Brandon Heller, V. J. B. L. N. M. (2012). Reproducible network experiments using container-based emulation.

[54] OpenvSwitch (2016). Esquema de la base de datos de openvswitch. url: http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf, último acceso Mayo 2016.

[55] Philip Wette, Martin Dräxler, A. S. (2014). Maxinet: Distributed emulation of software-defined networks.

[56] Ronald van der Pol, Bart Gijsen, P. Z. D. F. C. R. M. K. (2016). Assessment of sdn technology for an easy-to-use vpn service.

[Santiago Vidal] Santiago Vidal, Jorge Rodrigo Amaro, E. V. M. G. E. G. Rauflow: building virtual private networks with mpls and openflow.

[58] Sharma, S. (2014). Implementing quality of service for the software defined networking enabled future internet.

[59] Suzuki Kazuya, K. H. (2014). An openflow controller for reducing operational cost of ip-vpns.

[60] Vitaly Antonenko, R. S. (2013). Global network modelling based on mininet approach.

[61] Wang, S.-Y. (2013). Estinet openflow network simulator and emulator.

[62] Wang, S.-Y. (2014). Comparison of sdn openflow network simulator and emulators: Estinet vs. mininet.

[63] Yustus Eko Oktian, SangGon Lee, H. L. (2014). Mitigating denial of service (dos) attacks in openflow networks.

Apendice A

Manual de usuario del emulador

A.1 Modo de uso

Para utilizar el emulador es necesario instalar:

- Mininet. Version 2.2.1 o mayor.
- Open vSwitch. Version 2.4.0 o mayor.
- Quagga. Probado con versión 0.99.22.4-3ubuntu1.

Posicionándose en el directorio raíz, el emulador se inicia con el siguiente comando:

```
sudo python start.py {RUTA_TOPOLOGIA}
```

El valor de {RUTA_TOPOLOGIA} debe ser el path hacia el script Python que configura la topología. Para conocer los detalles de lo que debe hacer ese script, leer la sección A.2.

El script **start.py** realiza las siguientes funciones:

- Carga la topología recibida por parámetro y la inicia.
- Borra el archivo utils/init_json.json, en caso de que una ejecución previa lo haya creado. El propósito de este archivo se verá mas adelante.
- Llama al método **start** de cada nodo virtual. Cada una de las cuatro clases de nodos (RAUSwitch, RAUHost, RAUController y QuaggaRouter) tiene este método, que se encarga de inicializar y configurar el nodo.

Luego de iniciar, Mininet ofrece una línea de comandos con la que el usuario puede interactuar. Ejecutando el siguiente comando se puede obtener una terminal Linux en cualquiera de los nodos.

```
xterm {NOMBRE_NODO}
```

De aquí en adelante en este manual de usuario, cuando se indique que hay que ejecutar un determinado comando en un nodo, se lo debe ejecutar en una consola xterm en dicho nodo.

Habiendo iniciado el entorno virtual, hay que llevar a cabo algunos pasos más para que sea totalmente funcional:

1. Esperar a que OSPF termine de distribuir las rutas y actualizar las bases de datos topológicas. Este proceso en general toma menos de un minuto, y varía de acuerdo al tamaño de la topología. Una manera de verificar esto es con el comando **route** y analizando la tabla de ruteo de los nodos.

2. Ejecutar el comando **python telnetRouters.py** en cualquiera de los RAUSwitch. Este script escrito en Python se encarga de consultar la base de datos topológica de OSPF mediante Telnet, parsear la información y enviarla al controlador. Es importante asegurarse que el paso 1 esta completo antes de ejecutarlo, ya que en caso contrario la base de datos de OSPF estará incompleta y se estarán enviando datos incorrectos. Luego de recibir la topología, el controlador todavía necesita la siguiente información de cada RAUSwitch: nombres de interfaces, direcciones IP y direcciones MAC. Para obtener esta información, se conecta automáticamente con el nodo que tiene levantado el Web Service que hace disponible la información de cada nodo. El nodo que levanta el Web Service es el controlador mismo (es decir, se conecta consigo mismo mediante localhost) pero puede ser cualquiera, siempre y cuando esté ejecutando el script **utils/wsOVS.py**. Este script es el sustituto que se creó para suplantar a **wsSNMP.py** (recordar sección 3.3.5).

3. Para poder crear servicios en RAUFlow, se debe indicar cuales RAUSwitch son de borde y cuales no. En el caso de los que son de borde, también se debe especificar la dirección IP y MAC del nodo CE (que típicamente será un RAUHost o QuaggaRouter) con el que el RAUSwitch está conectado. Tradicionalmente esto se hace en la interfaz web de RAUFlow, pero esto resulta tedioso y lento si se tienen muchos nodos. Para acelerar este proceso se creó un script llamado **nodeInits.py** que se puede ejecutar desde cualquier nodo, y se encarga de enviar toda esa información al controlador mediante pedidos HTTP. La ejecución de este script es opcional; si el usuario desea puede ingresar los datos mediante la interfaz web. El script envía los datos que se encuentren en el archivo **init_json.json**, y dicho archivo es creado automáticamente cuando se levanta el emulador. En caso de hacerse, la ejecución de este script debe ser posterior a la de telnetRouters.py, ya que en caso contrario se estarían mandando datos de nodos que el controlador todavía no conoce. En la sección A.2 se explicará como

indicarle al emulador que nodos son de borde, así como las direcciones de los nodos CE.

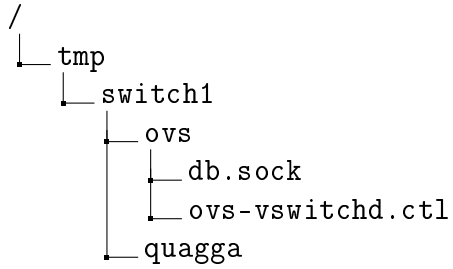
Luego de que el entorno está levantado y listo para usarse, se puede empezar a crear servicios. Para usar la interfaz web de RAUFlow se debe levantar un explorador desde el nodo controlador. Esto se puede lograr primero iniciando una consola xterm en dicho nodo, y luego ejecutando el comando que inicie el explorador. Una vez en la interfaz web de RAUFlow, se puede interactuar con ella de forma normal.

Al agregar VPN, es importante tener en cuenta que las rutas de los nodos clientes involucrados deben estar correctamente configuradas, ya sea tratándose de una VPN de capa 2 o 3. Esto se puede lograr con el comando *route* tradicional de Linux. En el caso de las VPN de capa 3, esto se puede evitar mediante el uso del parámetro **gw** al configurar la topología.

A.2 Cómo interactuar con cada instancia de Open vSwitch

Como se explica en el capítulo 3, cada RAUSwitch tiene su propia instancia de Open vSwitch ejecutándose en modo userspace. Esto modifica un poco la manera de usar sus comandos, ya que cada comando se debe 'apuntar' a la instancia con la que se desea interactuar.

Cada RAUSwitch tiene un directorio bajo /tmp donde se almacenan los archivos relacionados con su instancia de Open vSwitch y Quagga. El siguiente diagrama explica la estructura de archivos correspondiente a un nodo llamado 'switch1'.



El diagrama muestra dos archivos que son vitales para poder comunicarse con la instancia de Open vSwitch del nodo 'switch1'. Estos son: **db.sock** y **ovs-vswitchd.ctl**. El propósito de estos archivos se explicará más adelante. Los demás archivos que se mantienen en estos directorios son los relacionados con Quagga, y se omiten por simplicidad.

Open vSwitch tiene varias herramientas que permiten consultar datos y realizar configuraciones. Las de interés en este contexto son: ovs-appctl, ovs-vsctl, ovs-ofctl y ovs-dpctl. A continuación se explicará en que consiste cada una y como usarla apuntando a un nodo específico.

- **ovs-appctl** [9] es una herramienta que permite enviarle comandos al demonio ovs-vswitchd. Se le puede consultar cosas como flujos, logs, etc, así como realizar configu-

raciones en tiempo de ejecución. El entorno virtual tendrá múltiples instancias de este demonio ejecutando, así que es necesario indicar a qué instancia debe ser dirigido un comando. Esto se hace con la opción **-t** o **–target** seguido por el socket de Unix en el cual la instancia está escuchando por conexiones de control. Aquí entra en juego el archivo **ovs-vswitchd.ctl** mencionado anteriormente. Al iniciarse, cada instancia de Open vSwitch almacenará ese socket en el directorio privado de su nodo. Por lo tanto, para apuntar un comando 'ovs-appctl' hacia un nodo específico se debe hacer: *ovs-appctl –target=/tmp/nombre_nodo/ovs/ovs-vswitchd.ctl nombre_comando*.

- **ovs-vsctl** [12] permite conectarse con el proceso ovsdb-server, quien se encarga de mantener la base de datos de configuración de Open vSwitch. ovs-vsctl permite consultar y modificar dicha base de datos. Igual que en el caso de ovs-appctl, se debe especificar a que instancia se desea apuntar el comando. Esto se logra con la opción **–db**, que indica el modo de conexión que se utilizará. Este puede ser: un socket de Unix o la red. En caso de usar un socket de Unix, se debe indicar la ruta al archivo **db.sock** que le corresponde al nodo, de la siguiente manera: *ovs-vsctl –db=unix:/tmp/nombre_nodo/ovs/db.sock nombre_comando*. Por otro lado, si se desea enviar el comando a través de la red, se debe ejecutar: *ovs-vsctl –db=tcp:dirección_ip:6640 nombre_comando* usando la dirección IP del nodo de interés.

- **ovs-ofctl** [11] permite monitorear y administrar el switch OpenFlow. Por ejemplo, un uso frecuente es el de consultar el contenido de las tablas de flujos de un switch, que se hace con el siguiente comando: *ovs-ofctl -O OpenFlow13 dump-flows nombre_nodo*. A diferencia de las herramientas anteriores, no hace falta indicar de una forma especial a qué nodo apunta el comando. Alcanza con ejecutarlo en una consola xterm en el nodo que se busca administrar.

- **ovs-dpctl** [10] es un programa que permite consultar y administrar los flujos de los datapaths externos a ovs-vswitchd, como el datapath del kernel. El datapath del kernel no es utilizado en el entorno (debido a las múltiples instancias de Open vSwitch), por lo tanto no es posible usar esta herramienta. Para administrar el datapath en el userspace (también llamado netdev) se puede utilizar la familia de comandos *dpctl/** de ovs-appctl. Por ejemplo, *ovs-appctl –target=/tmp/nombre_nodo/ovs/ovs-vswitchd.ctl dpctl/show* muestra las estadísticas del cache de flujos para un determinado nodo.

A.3 API para configurar las topologias

La API que se debe usar para crear y personalizar las topologias es, en esencia, la misma que la de Mininet estándar. Cada topología debe ser configurada por un script Python, que debe definir una subclase de la clase *Topo* de Mininet. A dicha subclase se le debe agregar nodos y enlaces mediante los métodos *addHost*, *addSwitch*, y *addLink*. En el caso de los dos primeros, es necesario indicar la clase de nodo que se está agregando, y los parámetros necesarios para inicializar esa clase. Como se explica en el capítulo 3, se crearon cuatro nuevas clases de nodos, y a continuación se detallan los parámetros que se pueden usar en sus constructores. Los resaltados con * son obligatorios.

RAUSwitch

- **nombre** *. Nombre del switch.
- **ips** *. Lista con todas las direcciones IP (en formato CIDR, A.B.C.D/E) para el switch.
- **dpid** *. Datapath ID del switch. Debe ser un string hexadecimal de largo 16. En caso de no proveer este valor, el datapath ID se derivará del nombre del switch. Por ejemplo: si el nombre es *switch8*, su datapath ID será 8.
- **controller_ip** *. Dirección IP del controlador.
- **border**. Número entre 0 o 1 que indica si el switch es de borde o no. Si no se provee, se asume que border=0.
- **ce_ip_address**. Dirección IP del nodo CE con el que está conectado. Solo aplica si el switch es de borde.
- **ce_mac_address**. Dirección MAC del nodo CE con el que está conectado. Solo aplica si el switch es de borde.

RAUHost

- **nombre** *. Nombre del host.
- **ips** *. Lista con todas las direcciones IP (en formato CIDR, A.B.C.D/E) para el host. En general tendrá una única dirección IP, pero por si acaso se permite que tenga varias.
- **gw**. Dirección IP del default gateway. Es útil para el uso de VPN de capa 3.

- **ce_mac_address**. En caso de que el host se esté usando como nodo CE, este parámetro indica la dirección MAC que debe tener la interfaz que lo conecta con el RAUSwitch. En caso de existir, dicha interfaz debe ser la menor de todas.

RAUController

- **nombre** *. Nombre del controlador.
- **ips** *. Lista con todas las direcciones IP (en formato CIDR, A.B.C.D/E) para el controlador. En general tendrá una única dirección IP, pero por si acaso se permite que tenga varias.

QuaggaRouter

- **nombre** *. Nombre del router.
- **ips** *. Lista con todas las direcciones IP (en formato CIDR, A.B.C.D/E) para el router.
- **gw**. Dirección IP del default gateway. Es útil para el uso de VPN de capa 3.
- **ce_mac_address**. En caso de que el router se esté usando como nodo CE, este parámetro indica la dirección MAC que debe tener la interfaz que lo conecta con el RAUSwitch. En caso de existir, dicha interfaz debe ser la menor de todas.

Cada nodo tendrá un conjunto de interfaces de red, definidas de acuerdo a como se creen los enlaces de ese nodo. Por ejemplo, el siguiente fragmento de código crea enlaces entre tres nodos llamados *switch1*, *switch2* y *switch3*:

```
addLink(switch1, switch2, 0, 0)
addLink(switch1, switch3, 1, 0)
```

Los últimos dos parámetros indican qué número de interfaz debe tener cada nodo. La primera línea significa que tanto el switch1 como el switch2 tendrán eth0 conectadas a ese enlace. La segunda indica que el switch1 usará eth1 para el enlace, mientras que el switch3 usará eth0. Es importante prestar especial atención a esto, ya que el parámetro *ips* con el que se inicializa cada nodo debe cumplir ciertas reglas:

- La cantidad de direcciones IP debe coincidir con la cantidad de interfaces de red.
- Las direcciones IP serán asignadas según el orden en el que están en la lista, empezando desde la interfaz más baja.
- Si se trata de un RAUSwitch, la primera interfaz (y por ende la primera dirección IP de la lista) debe ser la de la red de gestión.

- Si se trata de un RAUSwitch y es de borde, la última interfaz debe ser la que lo conecte con el nodo CE, ya sea un RAUHost o un QuaggaRouter.
- Si se trata de un RAUHost o QuaggaRouter que actúa como nodo CE (es decir, está conectado a un RAUSwitch), la primera interfaz debe ser la que lo conecte con el RAUSwitch.

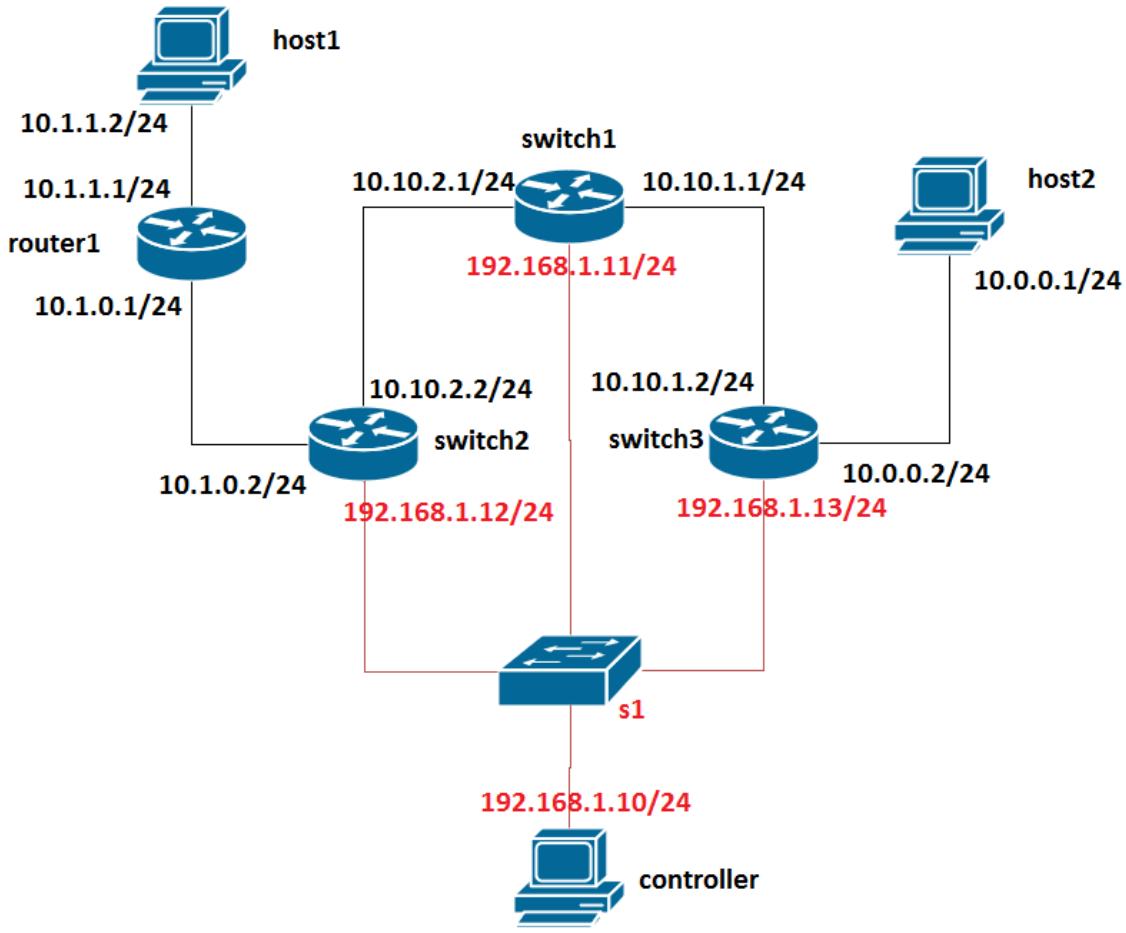
Estas restricciones son necesarias dado que el código que inicializa los nodos debe poder distinguir los distintos tipos de interfaces (red de gestión, red interna, customer edge). Una posible alternativa sería que el usuario cree las interfaces sin ningún tipo de restricción de orden, y con parámetros adicionales indique, para cada nodo, el tipo de cada interfaz. Se opta entonces por la solución que evita el uso de más parámetros, pero deja a cargo del usuario asegurarse que se cumplen las reglas establecidas.

A.3.1 Ejemplo

En la figura A.1 se puede ver una pequeña topología de ejemplo que consiste de 3 RAUSwitch, 1 QuaggaRouter actuando como CE, un RAUHost actuando como CE y otro RAUHost conectado al QuaggaRouter. También se puede ver la red de gestión marcada con rojo. A continuación se muestra el código que implementa esa topología:

```
from mininet.topo import Topo
from rau_nodes import RAUSwitch, QuaggaRouter, RAUController, RAUHost

class CustomTopology( Topo ):
  def __init__( self ):
        Topo.__init__( self )

        # Hosts
        host1 = self.addHost('host1',
                ips=['10.1.1.2/24'],
                cls=RAUHost)
        host2 = self.addHost('host2',
                ips=['10.0.0.1/24'],
                ce_mac_address='00:00:00:00:00:02',
                cls=RAUHost)

        # Router
        router1 = self.addHost('router1',
                ips=['10.1.0.1/24', '10.1.1.1/24'],
                ce_mac_address='00:00:00:00:00:01',
                cls=QuaggaRouter)

        # Switches
        # Los dpid se omiten ya que se pueden derivar de los nombres
        switch1 = self.addHost('switch1',
                ips=['192.168.1.11/24','10.10.2.1/24','10.10.1.1/24'],
```

Fig. A.1 Topología de ejemplo




```
            controller_ip="192.168.1.10",
            cls=RAUSwitch)

    switch2 = self.addHost('switch2',
            ips=['192.168.1.12/24','10.10.2.2/24','10.1.0.2/24'],
            controller_ip="192.168.1.10",
            border=1, ce_ip_address='10.1.0.1',
            ce_mac_address='00:00:00:00:00:01',
            cls=RAUSwitch)

    switch3 = self.addHost('switch3',
            ips=['192.168.1.13/24','10.10.1.2/24','10.0.0.2/24'],
            controller_ip="192.168.1.10",
            border=1, ce_ip_address='10.0.0.1',
            ce_mac_address='00:00:00:00:00:02',
            cls=RAUSwitch)
```

```
        # Controlador
        controller = self.addHost('controller',
                ips=['192.168.1.10/24'],
                cls=RAUController)

        # Switch de la red de gestion
        man_switch = self.addSwitch('s1',
                protocols='OpenFlow13',
                failMode='standalone')

        # Enlaces de la red de gestion
        # La primera interfaz de los RAUSwitch debe
        # contectarse con esta red
        self.addLink(man_switch, controller, 1, 0)
        self.addLink(man_switch, switch1, 2, 0)
        self.addLink(man_switch, switch2, 3, 0)
        self.addLink(man_switch, switch3, 4, 0)
        # Enlaces de la red interna
        self.addLink(switch1, switch2, 1, 1)
        self.addLink(switch1, switch3, 2, 1)
        # Enlaces de las redes cliente
        # La última interfaz de los nodos CE (router1 y host2) debe ser
        # la que lo conecte con la red SDN
        self.addLink(switch2, router1, 2, 0)
        self.addLink(router1, host1, 1, 0)
        self.addLink(switch3, host2, 2, 0)
```

A.4 GraphML Loader

Como se explica en el capítulo 3, GraphML Loader es un módulo que se desarrolló con el objetivo de asistir al usuario en el proceso de crear topologias. Recibe como entrada un archivo de tipo *graphml*, un formato que se puede encontrar, entre otros lados, en Topology Zoo [32], y produce el archivo Python que crea la topología dictada por el grafo. Se invoca de la siguiente manera:

```
python graphml_loader.py --file ruta_archivo_graphml
                         --output ruta_archivo_python
```

Un archivo graphml define un grafo mediante elementos que tienen el tag *node* o *edge*. La topología resultante de la ejecución del módulo será equivalente a dicho grafo, teniendo en cuenta que la red de gestión no debe estar incluida en el mismo. Naturalmente, las topologias disponibles en Topology Zoo (o cualquier otra fuente) no tienen todos los parámetros necesarios para instanciar una topología de forma completa. Como se verá a continuación, hay ciertos datos sobre los nodos que se pueden extraer a partir del archivo graphml, y otros que deben ser autogenerados por el módulo. La información que el módulo obtiene del archivo graphml es:

- **Tipo de nodo (type)**. Indica si el nodo es *rauhost*, *quaggarouter* o *rauswitch*. Naturalmente, este dato no estará presente en ninguna topología obtenida en Topology Zoo, o cualquier otra fuente. Por lo tanto, se requiere que el usuario lo ingrese manualmente. Si un nodo no especifica tipo, se asume que es *rauswitch*.

- **Identificador (id)**. Cada nodo tiene un identificador numérico, y dicho valor se utiliza para construir el nombre del nodo. Por ejemplo, si un nodo es de tipo *rauswitch* y tiene id=3, su nombre en la topología será *switch4*. Se suma uno al valor del identificador ya que es posible que haya un nodo con id=0, y *switch0* no es un nombre válido ya que 0 no es un datapath ID válido (como se explica anteriormente, se deriva el datapath ID a partir del nombre).

Los datos que GraphML Loader genera automáticamente son:

- Red de gestión. Tanto el controlador, como el switch genérico que lo conecta con los RAUSwitch son autogenerados, ya que no son parte de los grafos de entrada. La dirección IP del controlador toma el valor 192.168.1.10/24.

- Direcciones IP de los nodos. En el caso de los RAUSwitch, la dirección correspondiente a la interfaz de gestión tiene el formato **192.168.1.X/24**. Las otras interfaces de los RAUSwitch, y las interfaces de los demás nodos, llevan direcciones IP de tipo **10.10.X.Y/24**.

- Nombre de los nodos. Como se explica anteriormente, el nombre de los nodos es generado a partir del identificador que se encuentra en el archivo graphml.

- El módulo detecta automáticamente los enlaces de borde, es decir, entre un RAUSwitch y un RAUHost o QuaggaRouter, y agrega los parámetros necesarios para indicar que dicho switch es de borde.

- A los nodos CE (que están conectados con un RAUSwitch) les indica el default gateway como el switch con el que están conectados. Esto es útil para utilizar VPN de capa 3.

A.4.1 Ejemplo

A continuación se muestra un ejemplo de archivo de tipo *graphml* que describe una topología full mesh con 4 RAUSwitch y dos subredes cliente conectada a ella. Cada subred cliente está compuesta por un QuaggaRouter y un RAUHost.

```
<?xml version="1.0" encoding="utf-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
```

```
  <graph edgedefault="undirected">
    <node id="0">
      <data key="type">rauswitch</data>
    </node>
    <node id="1">
      <data key="type">rauswitch</data>
    </node>
    <node id="2">
      <data key="type">rauswitch</data>
    </node>
    <node id="3">
      <data key="type">rauswitch</data>
    </node>
    <node id="4">
      <data key="type">quaggarouter</data>
    </node>
    <node id="5">
      <data key="type">quaggarouter</data>
    </node>
    <node id="6">
      <data key="type">rauhost</data>
    </node>
    <node id="7">
      <data key="type">rauhost</data>
    </node>
    <edge source="0" target="1"></edge>
    <edge source="0" target="2"></edge>
    <edge source="0" target="3"></edge>
    <edge source="1" target="2"></edge>
    <edge source="1" target="3"></edge>
    <edge source="2" target="3"></edge>
    <edge source="2" target="4"></edge>
    <edge source="3" target="5"></edge>
    <edge source="4" target="6"></edge>
    <edge source="5" target="7"></edge>
  </graph>
</graphml>
```

Este archivo de ejemplo muestra sólo los datos que el módulo necesita, es decir, *nodes* y *edges*, y el *type* e *id* de cada nodo. Los archivos disponibles en Topology Zoo contienen mucha más información que no se usa, y se omite para simplificar el ejemplo. A continuación se muestra la topología de salida que genera el módulo GraphML Loader. Dicha topología está lista para ser cargada al emulador.

```
"""
Custom topology for Mininet, generated by GraphML Loader.
"""
from mininet.topo import Topo
from rau_nodes import RAUSwitch, QuaggaRouter, RAUController, RAUHost

class CustomTopology( Topo ):
  def __init__(self):
  "Create a topology."
```

```
# Initialize Topology
Topo.__init__(self)
# Add controller
root = self.addHost('controller',
                    cls=RAUController,
                    ips=['192.168.1.10/24'])

# Add management network switch
man_switch = self.addSwitch('s1',
                            protocols='OpenFlow13',
                            failMode='standalone')

# Add switches, hosts and routers
switch2 = self.addHost('switch2', cls=RAUSwitch,
                       controller_ip='192.168.1.10',
                       ips=['192.168.1.11/24', '10.10.1.2/24',
                            '10.10.4.1/24', '10.10.5.1/24'])
switch1 = self.addHost('switch1', cls=RAUSwitch,
                       controller_ip='192.168.1.10',
                       ips=['192.168.1.12/24', '10.10.1.1/24',
                            '10.10.2.1/24', '10.10.3.1/24'])
switch4 = self.addHost('switch4', cls=RAUSwitch,
                       controller_ip='192.168.1.10',
                       ips=['192.168.1.13/24', '10.10.3.2/24',
                            '10.10.5.2/24', '10.10.6.2/24',
                            '10.10.8.1/24'],
                       border=1, ce_ip_address='10.10.8.2',
                       ce_mac_address='00:00:00:00:00:2')
switch3 = self.addHost('switch3', cls=RAUSwitch,
                           controller_ip='192.168.1.10',
                           ips=['192.168.1.14/24', '10.10.2.2/24',
                                '10.10.4.2/24', '10.10.6.1/24',
                                '10.10.7.1/24'],
                           border=1, ce_ip_address='10.10.7.2',
                           ce_mac_address='00:00:00:00:00:1')
router6 = self.addHost('router6', cls=QuaggaRouter,
                       ips=['10.10.8.2/24', '10.10.10.1/24'],
                       ce_mac_address='00:00:00:00:00:2',
                       gw='10.10.8.1')
router5 = self.addHost('router5', cls=QuaggaRouter,
                       ips=['10.10.7.2/24', '10.10.9.1/24'],
                       ce_mac_address='00:00:00:00:00:1',
                       gw='10.10.7.1')
host8 = self.addHost('host8', cls=RAUHost,
                     ips=['10.10.10.2/24'])
host7 = self.addHost('host7', cls=RAUHost,
                     ips=['10.10.9.2/24'])
# Add links between nodes
self.addLink(man_switch, root, 1, 0)
self.addLink(man_switch, switch2, 2, 0)
self.addLink(man_switch, switch1, 3, 0)
self.addLink(man_switch, switch4, 4, 0)
self.addLink(man_switch, switch3, 5, 0)
self.addLink(switch2, switch1, 1, 1)
self.addLink(switch1, switch3, 2, 1)
```

```
        self.addLink(switch1, switch4, 3, 1)
        self.addLink(switch2, switch3, 2, 2)
        self.addLink(switch2, switch4, 3, 2)
        self.addLink(switch4, switch3, 3, 3)
        self.addLink(switch3, router5, 4, 0)
        self.addLink(switch4, router6, 4, 0)
        self.addLink(router5, host7, 1, 0)
        self.addLink(router6, host8, 1, 0)
```