Funcionalidades Avanzadas en Redes Definidas por Software.

Proyecto de grado

Santiago Vidal

Instituto de Computación

Facultad de Ingeniería, Universidad de la República

September 2014

Tabla de Contenidos

Tabla de Figuras

Tabla de Cuadros

Capítulo 1

Introducción

1.1 Contexto

Capítulo 2

Estado del arte

2.1 Software Defined Networking

Arquitectura
Modelo reactivo vs proactivo

2.2 OpenFlow

2.3 Open vSwitch

2.4 Red Privada Virtual (VPN)

2.5 Multiprotocol Label Switching (MPLS)

2.6 Simuladores para SDN

2.6.1 DOT

2.6.2 Estinet

2.6.3 fs-sdn

2.6.4 Flowsim

2.6.5 Mininet

2.6.6 NS-3

2.6.7 OFNet

2.6.8 Omnet++

2.6.9 Opencontrail

2.6.10 STS

2.7 Simuladores no SDN

2.7.1 CORE

2.7.2 Unified Networking Lab

2.7.3 VNX y VNUML

2.7.4 OpenStack

2.7.5 OPNET

2.7.6 Psimulator2

2.7.7 Shadow

hacer esta investigación.

NS-3

ns-3 fue descartado debido a que no ofrece soporte para Quagga ni OpenFlow 1.3 al momento de realizar esta investigacion.

Estinet

Estinet requiere licencias pagas, y se opto por elegir herramientas open source. Debido a la falta de documentación de libre acceso, no se sabe que tipo de capacidades ofrece.

Mininet

Mininet es un emulador de redes SDN que permite emular hosts, switches, controladores y enlaces. Utiliza virtualización basada en procesos para ejecutar múltiples instancias (hasta 4096) de hosts y switches en un unico kernel de sistema operativo. También utiliza una capacidad de Linux denominada *network namespace* que permite crear "interfaces de red virtuales", y de esta manera dotar a los nodos con sus propias interfaces, tablas de ruteo y tablas ARP. Lo que en realidad hace Mininet es utilizar la arquitectura *Linux container*, que tiene la capacidad de proveer virtualización completa, pero de un modo reducido ya que no requiere de todas sus capacidades. Mininet también utiliza *virtual ethernet (veth)* para crear los enlaces virtuales entre los nodos.

Mencionar MaxiNet (está en las referencias) Capaz mencionar Mininet-HiFi (buscar en el survey) Capaz mencionar Mininet CE (buscar en el survey) Capaz mencionar SDN Cloud DC (buscar en el survey)

LXC

La opción de crear nodos con Linux containers resuelve el problema de Quagga y OpenFlow 1.3, pero llevaría una gran cantidad de trabajo construir distintas topologias (sobre todo si son grandes), ya que casi todo debe ser configurado manualmente por el usuario. Es una opción similar a Mininet, solo que sin gozar de todas las facilidades que ofrece esta última.

Máquinas virtuales

Es una opción similar a LXC (Linux Containers), sólo que menos escalable.

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 la performance 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, las herramientas estudiadas, 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, porque puede que sea tecnológicamente imposible, 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

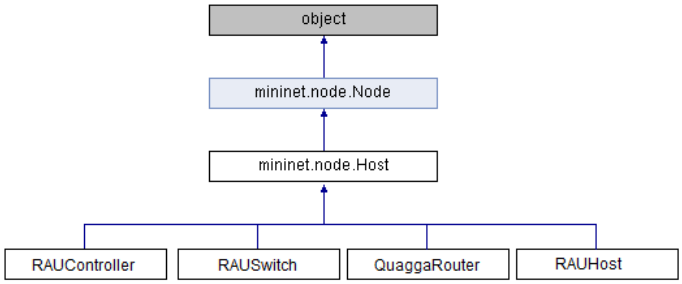
ejemplo, archivos de configuración. En el segundo grupo, se consideran los requerimientos inherentes a un entorno de simulación como el que se pretende.
El primer grupo de requerimientos se detalla a continuación.

- 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.
  - 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.
  - Resulta trivial que los RAUSwitch virtuales soporten OpenFlow, ya que es el cimiento de RAUFlow. Pero es importante que soporten 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 a partir de la versión 1.3 (???). Es muy deseable que lo hagan mediante Open vSwitch, ya que es lo que utilizan los RAUSwitch físicos.
  - 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.
- 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.
- La aplicación RAUFlow debe ejecutarse y comunicarse correctamente con los RAUSwitch. Esto también implica que el controlador Ryu debe ser soportado por el entorno.

El segundo grupo de requerimientos es más genérico, ya que son los que surgen para casi cualquier entorno de simulación de redes.

- Facilidad de configuración. Es importante que el entorno pueda generar distintas topologias y escenarios sin demasiado esfuerzo de configuración.
- Escalabilidad. Dado que uno de los objetivos es realizar pruebas de escala, el entorno debería ofrecer buena escalabilidad en la cantidad de nodos que puede simular. Esto

Fig. 3.1 Diagrama de clases del entorno.



se traduce a que una computadora promedio de uso personal pueda levantar algunas decenas de nodos virtuales como mínimo.

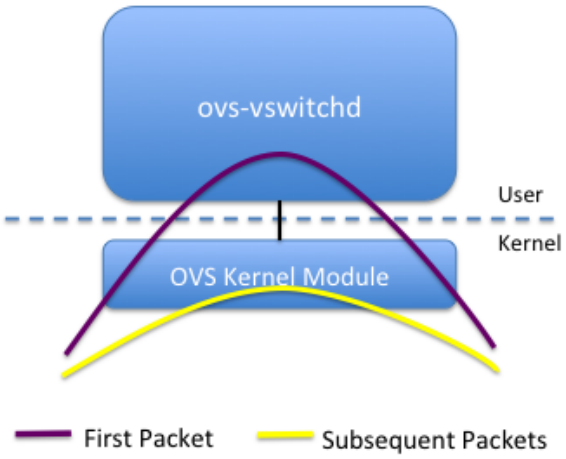
3.2 Elección de la herramienta

3.3 Diseño e implementación del entorno

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 OpenVSwitch, 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 OpenVSwitch 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, podemos ejecutar una instancia de Quagga y OpenVSwitch para cada host. De esta forma es posible crear un router 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.

Fig. 3.2 Arquitectura de OpenVSwitch.



3.3.1 RAUSwitch

La clase RAUSwitch es el núcleo del entorno virtual. Es un Host extendido de tal forma para que, gracias a la funcionalidad de directorios privados, ejecute su propia instancia de Quagga y OpenVSwitch. 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.

OpenVSwitch básicamente consiste de 2 demonios CONFIRMAR (appctl, ofctl, dpctl, vsctl) (ovs-vswitchd y ovsdb-server) que ejecutan en el user-space, y un módulo en el kernel que actúa como cache para los flujos recientes. Utiliza el protocolo 'netlink' para comunicar el user-space con el módulo en el kernel. Poder tomar decisiones sobre los paquetes a nivel del kernel, sin tener que pasar por el user-space, explica en gran medida el buen nivel de performance que ofrece OpenVSwitch. Sin embargo, tener múltiples módulos de kernel ejecutando en el mismo sistema operativo puede crear comportamientos impredecibles e incorrectos, ya que no está previsto para trabajar de esa forma.
Afortunadamente, OpenVSwitch puede ejecutarse completamente en modo user-space, es decir, sin soporte del módulo del kernel. Esto implica que se puede ejecutar tantas instancias de OpenVSwitch como se deseen, pero la performance va a ser significativamente peor. Esto no es una desventaja muy seria, ya que el objetivo del entorno no es ser performante al procesar paquetes. Cabe aclarar que en este modo OpenVSwitch continúa haciendo cacheo de flujos, pero ahora lo hace en el user-space.

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 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

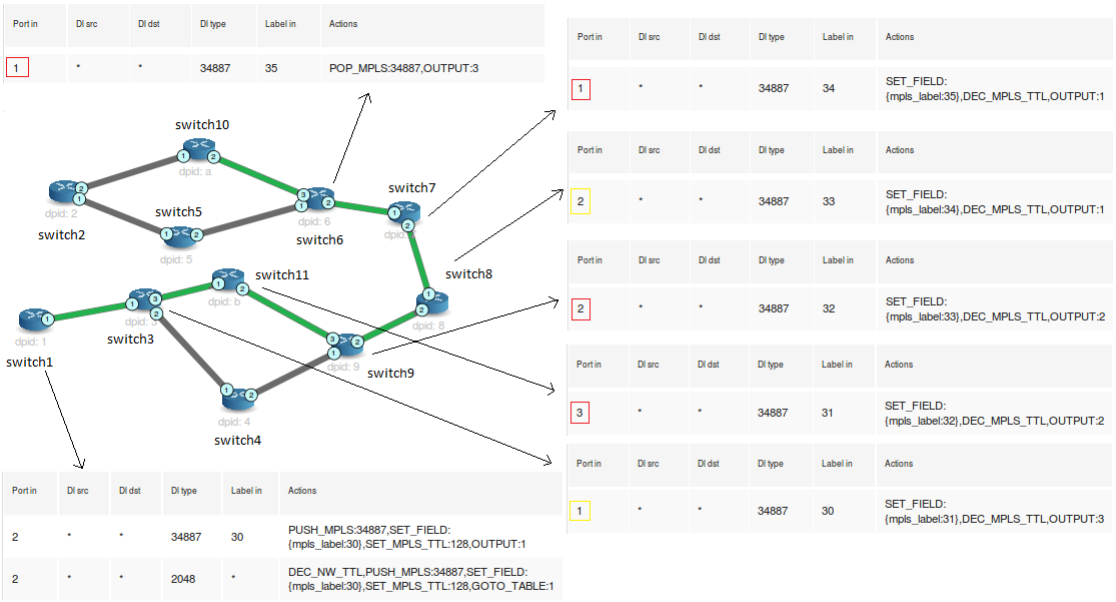
Como se explica en el Apéndice 1, cada topología se debe configurar en un script Python usando la API del entorno. Este proceso puede ser tedioso, especialmente si se trata de una topología grande. Todos los nodos se deben inicializar con los parámetros que requieren, también se deben crear los enlaces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

Tomando como base [20], 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 [14] 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 Problemas y errores encontrados

Como resultado de las pruebas funcionales básicas que se efectuaron sobre el entorno (que son detalladas en la sección 4.1.1) se detectaron determinados problemas que fueron necesarios estudiar. Esta sección explicará 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 otros son a causa del uso de un entorno virtual, y por lo tanto no se deberían tener en cuenta en un despliegue real.

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



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. Para solu-

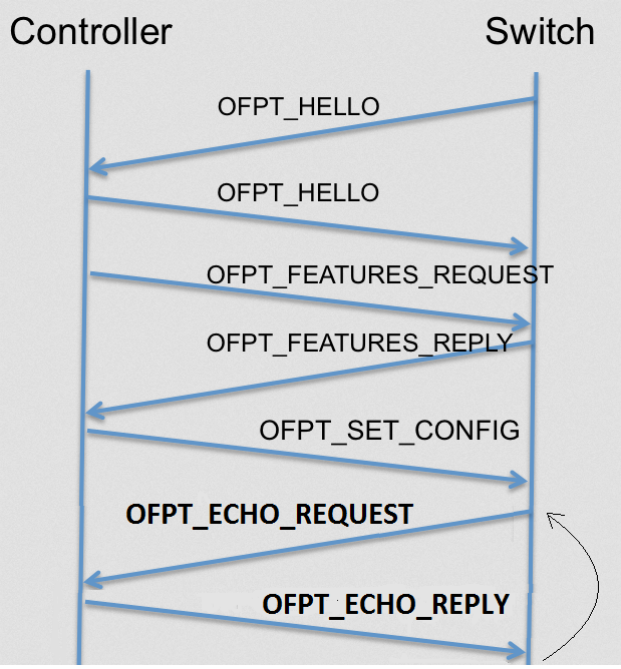
cionar esto se creó un "fork" del repositorio de RAUFlow para poder hacer las correcciones que correspondan. El arreglo de código de este error se puede ver en el siguiente commit: https://github.com/santiagovidal/LiveCode/commit/aeb575a10eb241dc3980a4c37846af7551bb7060.

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. Dicho arreglo se puede ver en el siguiente commit: https://github.com/santiagovidal/LiveCode/commit/4128923efcff38768aefd2864e10bd1adb63df52

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 la comunicación es correcta, se debería ver algo como lo 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

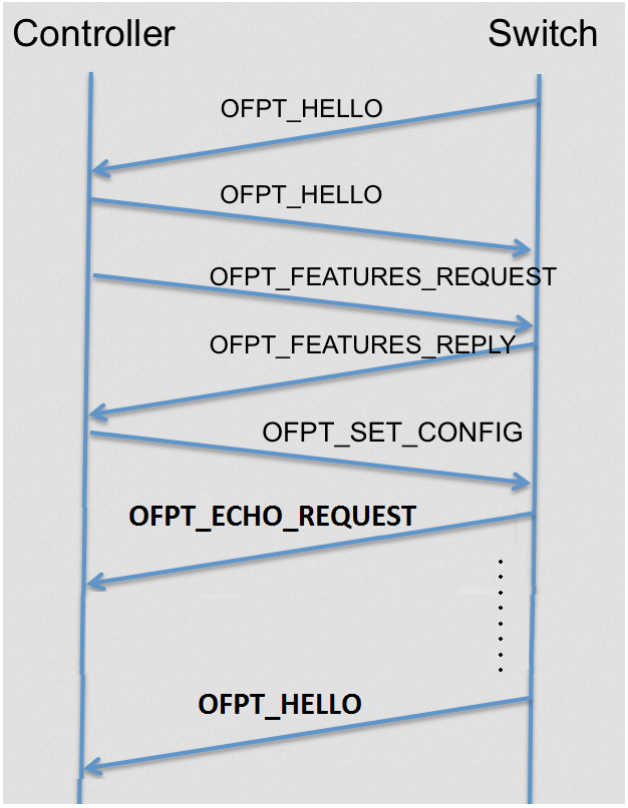
Fig. 3.4 Protocolo de control de OpenFlow.



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 [22], 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 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:

Fig. 3.5 Error de comunicación en protocolo de OpenFlow.



- 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 [19] 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 [19], 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 aproximadamente, 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 [2]. 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

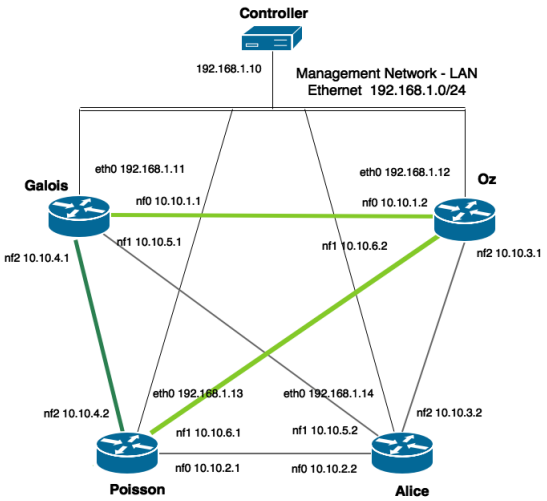
Con el entorno de simulación construido, el siguiente objetivo es realizar pruebas de escala sobre la arquitectura. Se realizan dos tipos de pruebas de escala: (a) verificar cómo funciona la arquitectura para topologias de escala, (b) realizar estudios de escala sobre la cantidad de servicios. En este 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. Todas las pruebas fueron realizadas en una máquina virtual con 32GB de RAM, procesador X y Lubuntu 14.04 como sistema operativo.

4.1 Topologias de escala

Es importante poder asegurar que la arquitectura puede ser fácilmente migrada a redes reales. Para poder asegurar esto, se diseñan los dos siguientes escenarios de prueba. El primero consiste en verificar los aspectos críticos de la arquitectura, como el algoritmo de ruteo y la clasificación de tráfico. En el segundo escenario se intenta estudiar como impacta el largo del camino y las características de la topología sobre el tiempo que demora el controlador en dar de alta una red privada virtual. Ambos escenarios utilizan un conjunto de topologias de prueba, que 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

Fig. 4.1 Topología básica



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 (gracias a la clase Switch de Mininet, en el modo *standalone*), que a su vez se conectará con los RAUSwitch. Esta será la red de gestión. Por simplicidad, dicha red se omite en las imágenes.

4.1.1 Escenario 1

En este escenario se crea una VPN punto a punto de capa 3 entre las dos subredes cliente, y se controla que tanto la creación de la VPN como el uso de la misma con tráfico, funciona correctamente. Esto se repite para cada topología de prueba. Los puntos específicos que se

Fig. 4.2 Topología chica

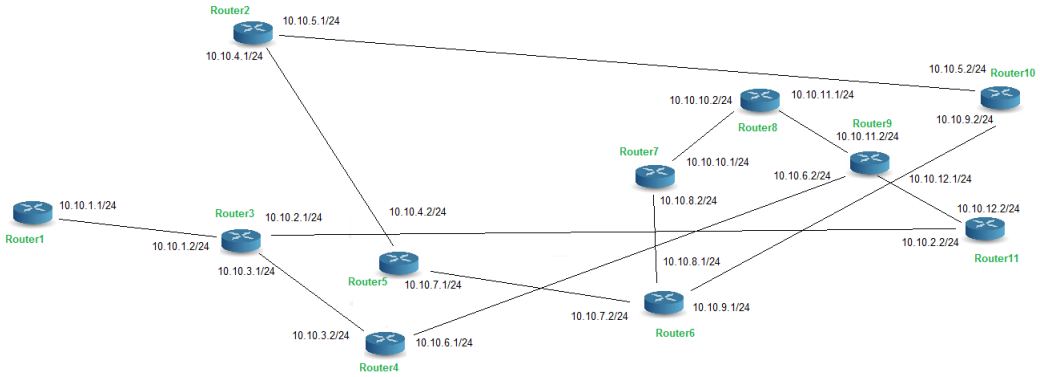
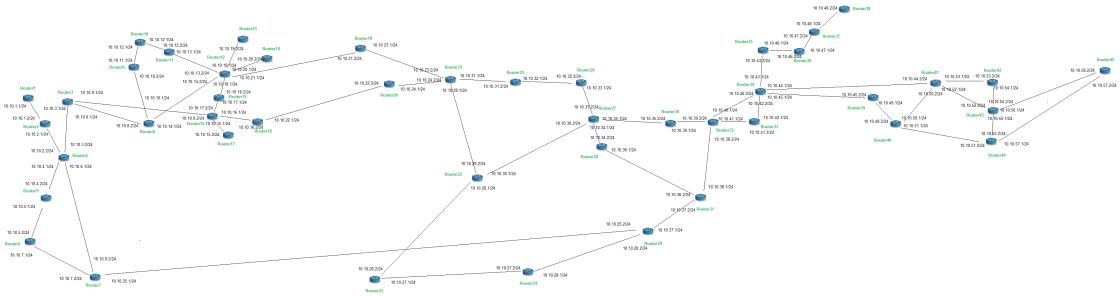


Fig. 4.3 Topología mediana

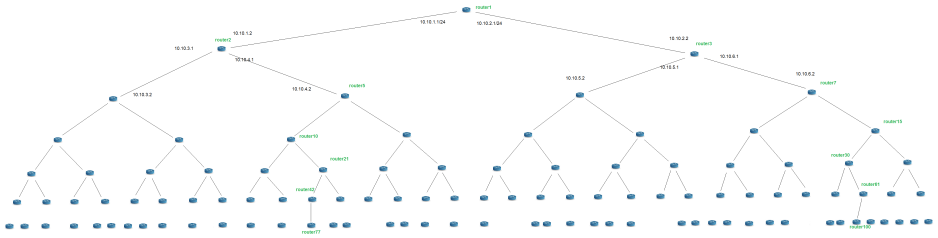


busca verificar se detallan a continuación:

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 (como pueden ser la mediana o grande, en este caso) 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.

Fig. 4.4 Topología grande



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 utilizará una VPN de capa 3, que permitirá tráfico con ethertype 0x0800, es decir, del protocolo IPv4. Se generará tráfico de este tipo utilizando el comando **ping** y la herramienta **iperf**. Con la herramienta tcpdump, se verificará que el tráfico pasa correctamente por cada nodo del camino.

Table 4.1 Pasos que cumple cada caso en la creación y uso exitoso de un servicio.

Largo del camino (topología)	Servicio	Camino	Flujos	Clasificación de tráfico
1 (básica)	X	X	X	X
7 (chica)	X	X		
10 (mediana)				
12 (grande)				

Con el propósito de hacer un diagnóstico más preciso sobre el proceso de creación y uso de una VPN, se lo descompone de 4 pasos conceptuales en los cuales podría haber fallas. Estos pasos son: (a) se crea con éxito el servicio, (b) el camino que se calcula es correcto, (c) los flujos de cada nodo del camino son correctos, (d) se clasifica correctamente el tráfico. Si se cumplen los 3 primeros pasos quiere decir que el servicio (y por ende la VPN que lo utilice) se establece correctamente y si se cumple el último paso entonces el tráfico pasa sin problemas por el servicio creado. En la tabla 4.1 se detallan los comportamientos observados para algunos de los casos estudiados. En ella se indica con una X los pasos que funcionaron correctamente para cada caso. Las celdas vacías indican qué paso falló en cada caso.

Analizando la tabla 4.1 se pueden observar como mínimo dos problemas. El primero es que en el caso de la topología chica y un camino de 7 saltos, el servicio se crea y los

flujos están en los nodos correctos (los del camino más corto entre las subredes cliente), pero los mismos no son correctos. El segundo comportamiento que se observa es que para el caso del camino de 10 saltos en la topología mediana, y el de 12 saltos en la topología grande, el servicio ni siquiera llega a crearse correctamente, es decir, la aplicación sufre una excepción de Python al intentar hacerlo. Las razones que explican esto, así como sus respectivas soluciones (si son posibles) se encuentran en la anterior sección 3.5, donde se explican los problemas encontrados y/o resueltos en el entorno. El resto de las pruebas que se mencionan en este capítulo fueron realizadas con dichas correcciones ya hechas.

4.1.2 Escenario 2

El objetivo de este escenario 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. 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. 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.

Las tablas 4.2, 4.3, 4.4 y 4.5 muestran los resultados obtenidos en el escenario. Cada tabla corresponde a una topología. En cada tabla se muestran los tiempos obtenidos para cada 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 y tiene sentido, 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 X). 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

Table 4.2 Tiempo de demora en crear VPN en la topología básica. El tiempo se mide en ms. Los valores de color marrón se corresponden con la VPN de capa 2, y los azules con la de capa 3.

Largo del camino	1
Ejecución	
1	188 / 14
2	188 / 23
3	199 / 15
4	188 / 15
Media	191 / 17
Mediana	188 / 15
Desv. Estandar	6 / 4
CV	0.03 / 0.24

Table 4.3 Tiempo de demora en crear VPN en la topología chica. El tiempo se mide en ms. Los valores de color marrón se corresponden con la VPN de capa 2, y los azules con la de capa 3.

Largo del camino	1	2	4	6	8
Ejecución					
1	155 / 29	241 / 25	269 / 52	257 / 42	285 / 51
2	208 / 22	309 / 27	298 / 42	320 / 55	333 / 64
3	351 / 19	317 / 31	327 / 43	348 / 56	337 / 64
4	220 / 18	317 / 29	301 / 39	326 / 57	342 / 60
Media	234 / 22	296 / 28	299 / 44	313 / 53	324 / 60
Mediana	214 / 21	313 / 28	300 / 43	323 / 56	335 / 62
Desv. Estandar	83 / 5	37 / 3	24 / 6	39 / 7	26 / 6
CV	0.35 / 0.23	0.13 / 0.11	0.08 / 0.14	0.12 / 0.13	0.08 / 0.10

camino, más flujos deben instalarse. Hay un caso particular en este comportamiento; el caso en que la VPN pasa por dos nodos (es decir, un camino de largo uno). En esa situación, la baja en el tiempo es más acentuada. Esto se debe a que cuando la VPN pasa por un camino de largo mayor que uno, el controlador debe configurar dos niveles de etiquetas. Esto requiere más tiempo de computo, y instalar un flujo (confirmar) 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 grande entonces se necesita más tiempo para crear la VPN. Ese comportamiento se explica con los siguientes puntos:

Table 4.4 Tiempo de demora en crear VPN en la topología mediana. El tiempo se mide en ms. Los valores de color marrón se corresponden con la VPN de capa 2, y los azules con la de capa 3.

Largo del camino	1	2	4	6	8	10	12
Ejecución							
1	254 / 101	567 / 104	372 / 129	364 / 126	436 / 130	414 / 181	441 / 289
2	318 / 92	746 / 134	504 / 105	397 / 187	515 / 271	548 / 197	519 / 152
3	355 / 110	520 / 112	498 / 135	675 / 164	529 / 202	485 / 214	613 / 154
4	318 / 110	611 / 127	438 / 145	529 / 153	787 / 179	477 / 191	571 / 178
Media	311 / 103	611 / 119	453 / 129	491 / 158	567 / 196	481 / 196	536 / 193
Mediana	318 / 106	589 / 120	468 / 132	463 / 159	522 / 191	481 / 194	545 / 166
Desv. Estandar	42 / 9	97 / 14	62 / 17	142 / 25	152 / 59	55 / 14	74 / 65
CV	0.14 / 0.09	0.16 / 0.12	0.14 / 0.13	0.29 / 0.16	0.27 / 0.30	0.11 / 0.07	0.14 / 0.34

Table 4.5 Tiempo de demora en crear VPN en la topología grande. El tiempo se mide en ms. Los valores de color marrón se corresponden con la VPN de capa 2, y los azules con la de capa 3.

Largo del camino	1	2	4	6	8	10	12
Ejecución							
1	425 / 483	642 / 810	1692 / 1000	1669 / 722	1516 / 2182	493 / 449	1843 / 594
2	940 / 325	457 / 223	2457 / 2209	1254 / 1016	2273 / 608	775 / 638	3438 / 1422
3	573 / 581	1044 / 319	1543 / 1070	1048 / 476	3482 / 856	784 / 563	1791 / 1018
4	428 / 282	2671 / 1346	1613 / 668	2678 / 569	3496 / 748	966 / 570	2848 / 849
Media	592 / 418	1204 / 675	1826 / 1237	1662 / 696	2692 / 1099	755 / 555	2480 / 971
Mediana	501 / 404	843 / 565	1653 / 1035	1462 / 646	2878 / 802	780 / 567	2346 / 934
Desv. Estandar	242 / 139	1009 / 516	425 / 672	725 / 236	972 / 729	195 / 78	803 / 348
CV	0.41 / 0.33	0.99 / 0.76	0.23 / 0.54	0.44 / 0.34	0.36 / 0.66	0.26 / 0.14	0.32 / 0.36

- 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 entorno virtual, 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 todo el entorno 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 para un camino de largo 2 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. La variabilidad que introduce la CPU no se comprueba en este trabajo, y se deja para trabajos futuros profundizar en el tema.

**Algunas conclusiones de estos resultados.

4.2 Escala de servicios y flujos

Entre los requerimientos de la RAU2 se encuentra el de la escalabilidad de usuarios. En particular, 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. He aquí la relevancia de las pruebas en la presente sección. Mediante el entorno virtual, se someterá la arquitectura a una cantidad de servicios relativamente grande y de esta forma se podrían identificar posibles puntos de falla, o umbrales bajo los cuales debe mantenerse la red para funcionar con buen rendimiento. Es importante recordar que aunque el entorno de simulación permite hacer un valioso estudio de escalabilidad, no generará resultados relevantes en lo que refiere al nivel de performance de la arquitectura. Recordar sección 3.3.2, 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.

4.2.1 Descripción del escenario

La idea principal del escenario es crear muchas VPN y analizar los comportamientos que esto genera. 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 RAUHost. Para cargar a la arquitectura 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.
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.

El objetivo es verificar los siguientes dos 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 crece su tabla 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. Se utilizará *iperf* para medir la velocidad de transferencia entre las subredes cliente.

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.

Esta prueba se repite para las mismas topologias que la prueba anterior, es decir: básica (4 nodos), chica (11 nodos) y mediana (45 nodos).

4.2.2 Resultados y observaciones

Table 4.6 Throughput en Kbits/s para cada caso.

# de VPN	Básica	Chica	Mediana	Grande
1	893	Y	W	Z
3000	887	Y	W	Z
6000	887	Y	W	Z
9000	890	Y	W	Z
12000	885	Y	W	Z
15000	886	Y	W	Z

Como se explica en el primer objetivo de esta prueba, se busca determinar si la existencia de muchos flujos en la tabla, implica que un switch OpenFlow demora más tiempo en

encontrar el flujo 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 4.6 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 es constante para un camino y topología, sin importar la cantidad de VPN existentes en el momento (se asume que las pequeñas diferencias numéricas entran en el margen de error). La máxima cantidad de VPN 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 de porqué esa cantidad de flujos no afecta se encuentra en la especificación de la herramienta Open vSwitch, que utiliza la arquitectura y el entorno virtual para implementar OpenFlow. Dicha herramienta 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 pasar por el pipeline.

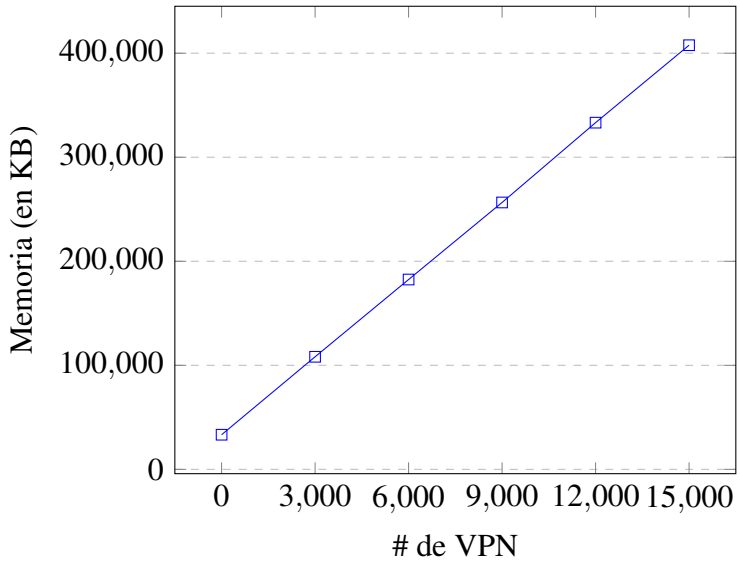
Mediante el comando 'ovs-appctl dpctl/show' de Open vSwitch, podemos examinar las estadísticas de la cache del datapath. Con el parámetro *target* se apunta el comando a cada instancia de Open vSwitch, y por ende, a cada nodo. En la figura 4.5 se observan las estadísticas del nodo 'alice'. En la sección '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. 4.5 Estadísticas de cache de flujos del nodo 'alice'.

Table 4.7 Evolución del consumo de memoria del controlador.

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

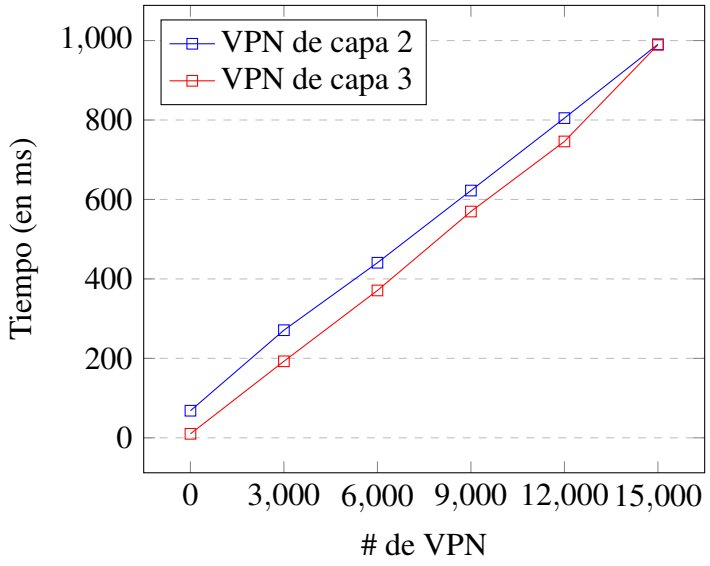
Efecto de la cantidad de VPN sobre la memoria consumida



Otro objetivo de la prueba es determinar si la arquitectura, y en particular el controlador, tienen algún problema para manejar muchos servicios. En la prueba de servicios ya mencionada no se detectó 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 podemos analizar la evolución del consumo de memoria del controlador a medida que se le agregan servicios. En la tabla 4.7 y la gráfica X se puede observar el resultado de estas mediciones.
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 Y KB cada 3000 servicios, por lo tanto se puede calcular que cada servicio ocupa Z KB (Y/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 P 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 VPN 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 siguiente gráfica.

Efecto de la cantidad de VPN 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 forma explicada en el capítulo 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 VPN 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 puede ser que la aplicación se vuelve más lenta a medida que sus estructuras de datos crecen. Sin embargo, no se observó ningún patrón en el código que indique esto.

***Algunas conclusiones de estas pruebas.

Capítulo 5

Conclusiones

5.1 Trabajo futuro

miniedit
: Sanity checks en la topología (IP por ej.)

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

[3] (2016). Flowsim. url: https://flowsim.flowgrammable.org/, último acceso Mayo 2016.

[4] (2016a). Manual de ovs-appctl. url: http://openvswitch.org/support/dist-docs/ovs-appctl.8.txt, último acceso Mayo 2016.

[5] (2016b). Manual de ovs-dpctl. url: http://openvswitch.org/support/dist-docs/ovs-dpctl.8.txt, último acceso Mayo 2016.

[6] (2016c). Manual de ovs-ofctl. url: http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt, último acceso Mayo 2016.

[7] (2016d). Manual de ovs-vsctl. url: http://openvswitch.org/support/dist-docs/ovs-vsctl.8.txt, último acceso Mayo 2016.

[8] (2016a). Mininet, sitio web oficial. url: http://mininet.org/, último acceso Mayo 2016.

[9] (2016b). miniNExT, extensión de MiniNet. url: https://github.com/USC-NSL/miniNExT, último acceso Agosto 2015.

[10] (2016). OFNet. url: http://www.sdninsights.org/, último acceso Mayo 2016.

[11] (2016). Open Source Network Simulators. url: http://www.brianlinkletter.com/open-source-network-simulators/, último acceso Mayo 2016.

[12] (2016). Repositorio con el código del entorno virtual. url: https://github.com/santiagovidal/P2015_44, último acceso Mayo 2016.

[13] (2016). STS - SDN Troubleshooting System. url: http://ucb-sts.github.io/sts/, último acceso Mayo 2016.

[14] (2016). The Internet Topology Zoo. url: http://topology-zoo.org/, último acceso Mayo 2016.

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

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

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

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

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

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

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

[22] OpenvSwitch (2016). Manual de OpenvSwitch. url: http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf, último acceso Mayo 2016.

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

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

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

Apendice A

Manual de usuario del emulador

A.1 Modo de uso

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,

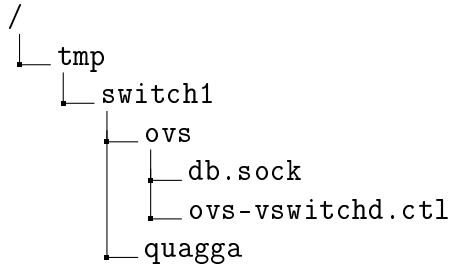
y luego ejecutando el comando que inicie el explorador. Una vez en la interfaz web de RAUFlow, se puede interactuar con ella de forma normal.

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** [4] es una herramienta que permite enviarle comandos al demonio ovs-vswitchd. Se le puede consultar cosas como flujos, logs, etc, así como realizar configuraciones 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** [7] 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** [6] 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** [5] 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.

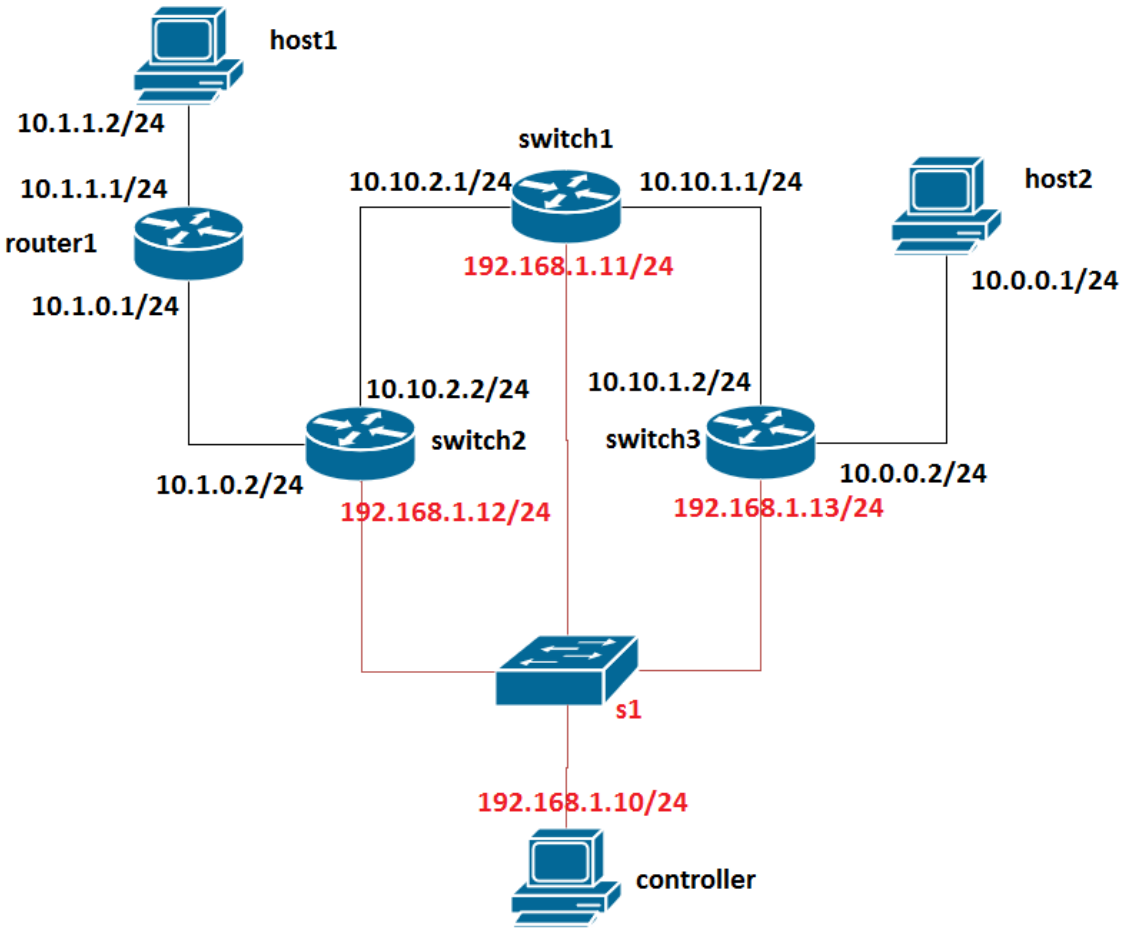
Cada nodo tendrá un conjunto de interfaces de red, definidas de acuerdo a como se creen los enlaces de ese nodo. Por ejemplo, el siguiente fragmento de código crea enlaces entre tres nodos llamados *switch1*, *switch2* y *switch3*:

```
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.

Fig. A.1 Topología de ejemplo



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'],
                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 [14], 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)
```