

Funcionalidades Avanzadas en Redes Definidas por Software.

Proyecto de grado



Santiago Vidal

Instituto de Computación
Facultad de Ingeniería, Universidad de la República

September 2014

Índice general

Índice de figuras	v
Índice de cuadros	vii
1. Introducción	1
2. Estado del arte	3
2.1. Software Defined Networking	3
2.2. OpenFlow	4
2.3. OpenVSwitch	4
2.4. Virtualización de SDN	4
2.4.1. NS-3	4
2.4.2. Estinet	4
2.4.3. Mininet	4
2.5. Aplicaciones de SDN	4
2.6. Red Privada Virtual (VPN)	4
2.7. Multiprotocol Label Switching (MPLS)	4
3. Entorno virtual	7
3.1. Requerimientos del entorno virtual	7
3.2. ¿Por qué Mininet?	8
3.3. Diseño e implementación del entorno	8
3.3.1. RAUController	9
3.3.2. RAUSwitch	9
3.3.3. QuaggaRouter	11
3.3.4. RAUHost	11
3.4. Modo de uso del entorno	11
3.4.1. GraphML Loader	11

4. Pruebas de escala	13
4.1. Topologías de escala	14
4.1.1. Escenario 1	14
4.1.2. Escenario 2	16
4.2. Escala de servicios y flujos	18
4.2.1. Descripción del escenario	18
4.2.2. Resultados y observaciones	19
5. Conclusiones	25
5.1. Trabajo futuro	25
Bibliografía	27

Índice de figuras

3.1. Diagrama de clases del entorno.	9
3.2. Arquitectura de OpenVSwitch.	10
4.1. Topología chica	14
4.2. Topología mediana	15
4.3. Estadísticas de cache de flujos del nodo 'alice'.	21

Índice de cuadros

4.1. Pasos que cumple cada caso en la creación y uso exitoso de un servicio. . .	15
4.2. Tiempo de demora en crear VPN en la topología chica. El tiempo se mide en ms.	17
4.3. Tiempo de demora en crear una VPN de capa 2. El tiempo se mide en ms. .	17
4.4. Tiempo de demora en crear una VPN de capa 3. El tiempo se mide en ms. .	18
4.5. Throughput en Kbits/s para cada caso.	19
4.6. Evolución del consumo de memoria del controlador.	20

Capítulo 1

Introducción

Capítulo 2

Estado del arte

2.1. Software Defined Networking

Arquitectura

Modelo reactivo vs proactivo

2.2. OpenFlow

2.3. OpenVSwitch

2.4. Virtualización de SDN

2.4.1. NS-3

2.4.2. Estinet

2.4.3. Mininet

2.5. Aplicaciones de SDN

2.6. Red Privada Virtual (VPN)

2.7. Multiprotocol Label Switching (MPLS)

Se estudió el estado del arte en lo que respecta a opciones de emulación o simulación para SDN. A continuación se detallan las principales herramientas analizadas al momento de 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 investigación.

Estinet

Estinet requiere licencias pagas, y se optó 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 único 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.

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 topologías (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 esto se puede utilizar dos parámetros: topología y servicios. Es importante poder aplicar topologías 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. En este capítulo se estudian los requerimientos que debe cumplir este entorno, las herramientas estudiadas para lograrlo, y los detalles de diseño e implementación de la solución construida.

3.1. Requerimientos del entorno virtual

Los requerimientos de este entorno se pueden dividir en dos grupos. En primer lugar, la idea es que el entorno virtual se comporte de una forma lo más fiel posible al prototipo físico. Esto no quiere decir que deba usar las mismas herramientas, pero es deseable que así sea. En segundo lugar, hay que considerar los requerimientos inherentes de un entorno de simulación como el que se pretende. El primer grupo 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 ejecutar el protocolo de enrutamiento OSPF. Es deseable que lo hagan mediante la suite de ruteo Quagga.

- Deben soportar el protocolo OpenFlow 1.3. Esto se debe a que la aplicación que implementa VPNs depende de que los switches tengan soporte para MPLS, y OpenFlow ofrece esta funcionalidad a partir de la versión 1.3 (???). Es muy deseable que lo hagan mediante OpenVSwitch, ya que es lo que utilizan los RAUSwitch físicos.
- Se debe poder simular múltiples hosts, ya que son los agentes que se conectan a la red y se envían tráfico entre sí, para corroborar que los flujos de datos son correctos.
- La aplicación RAUFlow debe ejecutarse y comunicarse correctamente con los RAUSwitch. Esto implica que el controlador Ryu debe ser soportado por el entorno.

Cabe remarcar que los módulos SNMP y LSDB Sync quedan por fuera de los requerimientos principales, por ser no esenciales.

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

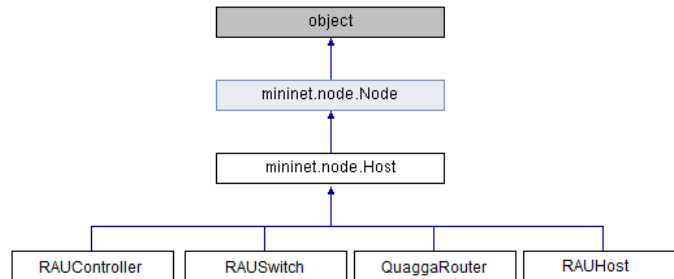
- Facilidad de configuración. Es importante que el entorno pueda generar distintas topologías y escenarios sin demasiado esfuerzo de configuración.
- Escalabilidad. Dado que uno de los objetivos es realizar pruebas de escala, el entorno debería ofrecer buena escalabilidad en la cantidad de nodos que puede simular. Esto se traduce a que una computadora promedio de uso personal pueda levantar algunas decenas de nodos virtuales como mínimo.

3.2. ¿Por qué Mininet?

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

Figura 3.1 Diagrama de clases del entorno.



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.

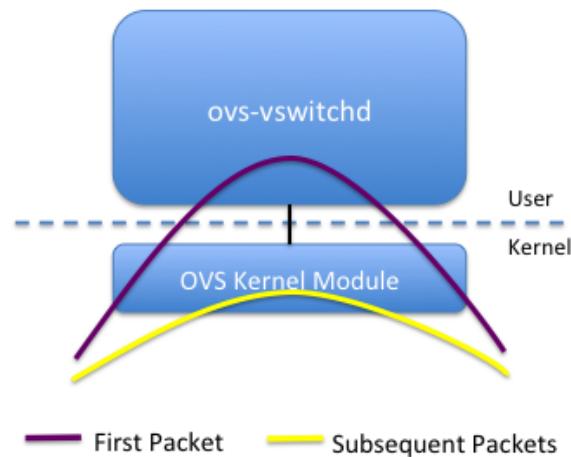
3.3.1. RAUController

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.

3.3.2. RAUSwitch

La clase RAUSwitch es el núcleo del entorno de simulación. 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:

Figura 3.2 Arquitectura de OpenVSwitch.



/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 (ovs-vswitchd y ovssdb-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 podemos ejecutar tantas instancias de OpenVSwitch como queramos, 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.

3.3.3. QuaggaRouter

Es una clase similar al RAUSwitch pero sin OpenVSwitch, 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*.

3.4. Modo de uso del entorno

En Mininet estándar, las topologías se crean mediante la API en Python. Se crea un objeto de tipo Topology, se le agregan los nodos que se desee, y se establecen los enlaces virtuales entre esos nodos. Como el entorno es en esencia una extensión de Mininet, hereda su facilidad de uso. La única diferencia radica en que las entidades de este entorno requieren parámetros adicionales para su creación, que serán detallados en el Anexo.

3.4.1. GraphML Loader

Mencionar: Cambiar SNMP por OVS

Capítulo 4

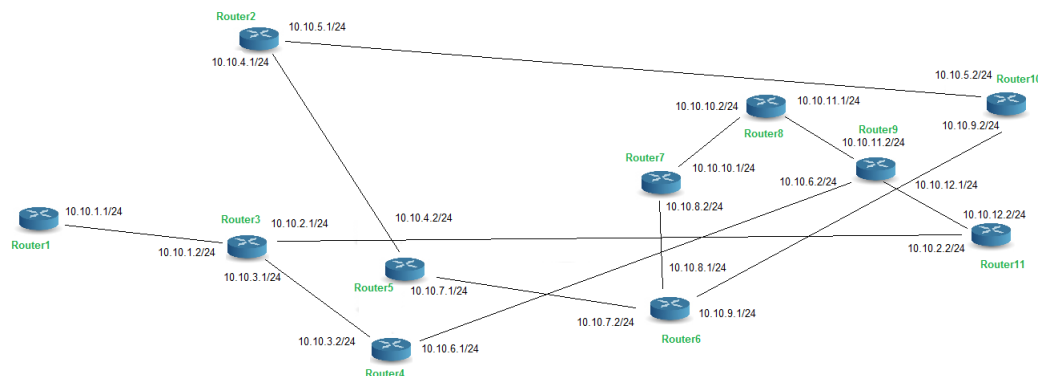
Pruebas de escala

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 topologías 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 (topologías, tipos de tráfico, etc) y por último, los resultados que arrojan. Todas las pruebas fueron realizadas en una máquina virtual con 3590 MB de RAM, procesador Intel Core i5-5200u a 2,2 GHz y Lubuntu 14.04 como sistema operativo.

Como se explicará mas adelante en el capítulo, en estas pruebas se utilizarán ciertas topologías creadas manualmente. Más allá de los detalles de cada topología, los siguientes puntos serán comunes a todas ellas.

- Habrá un conjunto de RAUSwitch, conectados de acuerdo a lo que dicte cada topología.
- Existirán dos subredes cliente. Serán implementadas por un QuaggaRouter y un RAUHost cada una (recordar las clases del entorno virtual). 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 futuras imágenes de las topologías.
- El controlador se conectará 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 omitirá en las futuras imágenes.

Figura 4.1 Topología chica



4.1. Topologías 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 topologías de prueba creadas manualmente, que se listan a continuación:

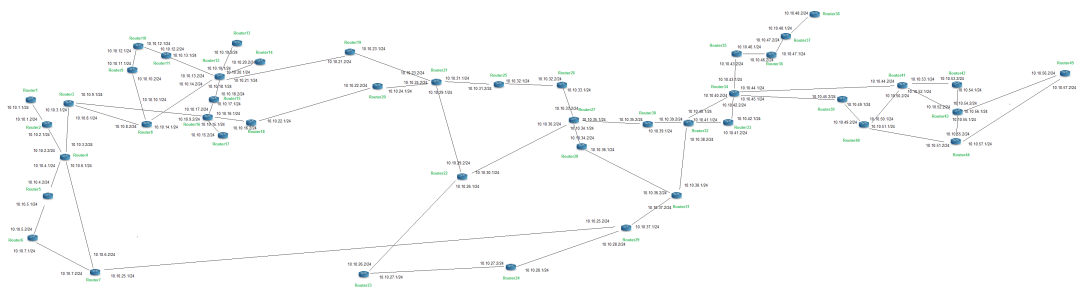
- **Básica:** 4 nodos en topología de full mesh. Es la utilizada en el prototipo físico.
- **Chica:** topología arbitraria de 11 nodos (fuente: Topology Zoo). Figura 4.1.
- **Mediana:** topología arbitraria de 45 nodos (fuente: Topology Zoo). Figura 4.2.
- **Grande:** topología de tipo arborescente compuesta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

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 busca verificar se detallan a continuación:

Algoritmo de ruteo

Figura 4.2 Topología mediana



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. 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 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.

Cuadro 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 14 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 detallan a continuación.

Error de código 1 Explicación.

Error de código 2 Explicación

Problema del entorno virtual para levantar la topología grande Explicación.

Observación acerca del MTU Explicación.

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 mediciones representativas y reducir el margen de error, en lugar de crear la VPN y medir su tiempo solamente, se crearán cuatro y se calculará el promedio de sus tiempos. 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 que hacerlo de forma manual a través de la interfaz web.

Cuadro 4.2 Tiempo de demora en crear VPN en la topología chica. El tiempo se mide en ms.

Largo del camino	1	2	4	6	8
Ejecución					
1	155 / 29	/	/	/	/
2	208 / 22	/	/	/	/
3	351 / 19	/	/	/	/
4	220 / 18	/	/	/	/
Media	234 / 22	/	/	/	/
Mediana	214 / 21	/	/	/	/
Desv. Estandar	83 / 5	/	/	/	/
CV					

Cuadro 4.3 Tiempo de demora en crear una VPN de capa 2. El tiempo se mide en ms.

Largo del camino	Básica	Chica	Mediana	Grande
1	58.3	65.0	83.1	
2	N/C	100.8	124.0	
4	N/C	105.2	124.1	
6	N/C	107.8	131.4	
8	N/C	109.6	128.6	
10	N/C	N/C	139.8	
12	N/C	N/C	133.4	

Las tablas 4.3 y 4.4 muestran para cada topología, los distintos largos de camino probados y su respectivo tiempo. La tabla 4.3 muestra los resultados cuando se trata de una VPN de capa 2, y la tabla 4.4 lo hace cuando es VPN de capa 3. Las celdas con *N/C* indican que no es posible crear un camino de ese largo para esa topología.

*** Algunas conclusiones de las pruebas de tiempo

Cuadro 4.4 Tiempo de demora en crear una VPN de capa 3. El tiempo se mide en ms.

Largo del camino	Básica	Chica	Mediana	Grande
1	6.3	6.8	19.3	
2	N/C	7.8	21.5	
4	N/C	9.4	22.3	
6	N/C	15.1	23.8	
8	N/C	12.0	23.7	
10	N/C	N/C	26.4	
12	N/C	N/C	27.5	

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 topologías que la prueba anterior, es decir: básica (4 nodos), chica (11 nodos) y mediana (45 nodos).

4.2.2. Resultados y observaciones

Cuadro 4.5 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 4.5 se pueden observar los throughput promedio medidos

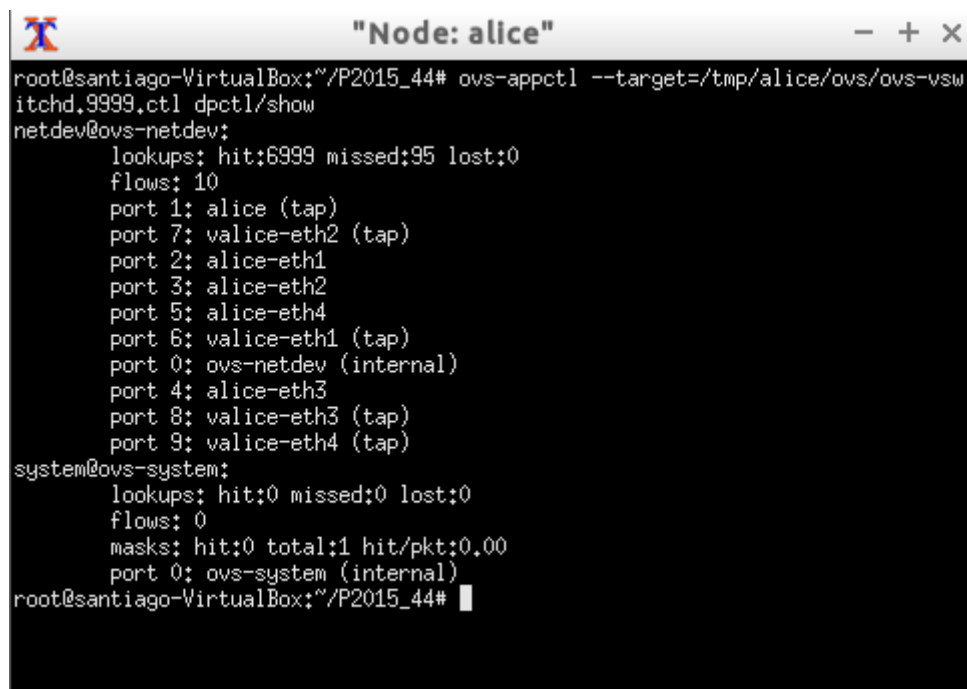
para un flujo de datos sobre la topología básica, con distintas cantidades de VPN existiendo en la red. La principal conclusión que se puede sacar de la tabla es que el throughput es constante para un camino y topología, sin importar la cantidad de VPN existentes en el momento (se asume que las pequeñas 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 4.3 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.

Cuadro 4.6 Evolución del consumo de memoria del controlador.

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

Figura 4.3 Estadísticas de cache de flujos del nodo 'alice'.

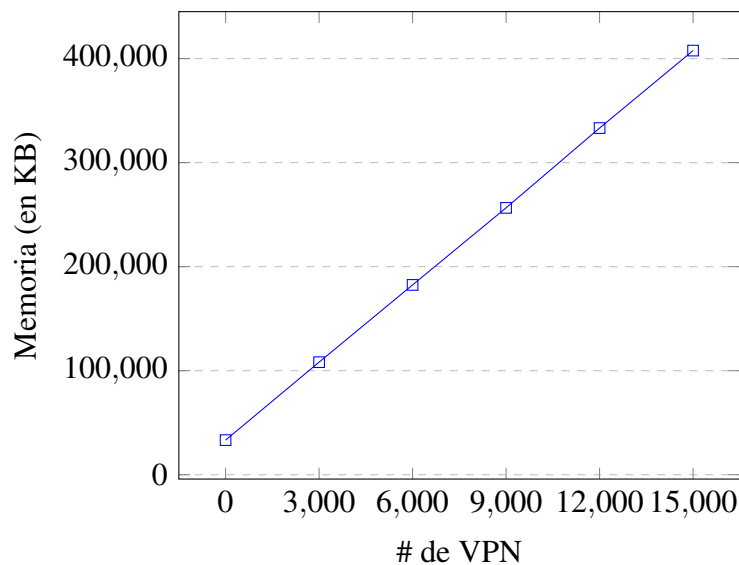


```

root@santiago-VirtualBox:~/P2015_44# ovs-appctl --target=/tmp/alice/ovs/ovs-vsw
itchd,9999,ctl dpctl/show
netdev@ovs-netdev:
  lookups: hit:6999 missed:95 lost:0
  flows: 10
  port 1: alice (tap)
  port 7: valice-eth2 (tap)
  port 2: alice-eth1
  port 3: alice-eth2
  port 5: alice-eth4
  port 6: valice-eth1 (tap)
  port 0: ovs-netdev (internal)
  port 4: alice-eth3
  port 8: valice-eth3 (tap)
  port 9: valice-eth4 (tap)
system@ovs-system:
  lookups: hit:0 missed:0 lost:0
  flows: 0
  masks: hit:0 total:1 hit/pkt:0.00
  port 0: ovs-system (internal)
root@santiago-VirtualBox:~/P2015_44#

```

Efecto de la cantidad de VPN sobre la memoria consumida



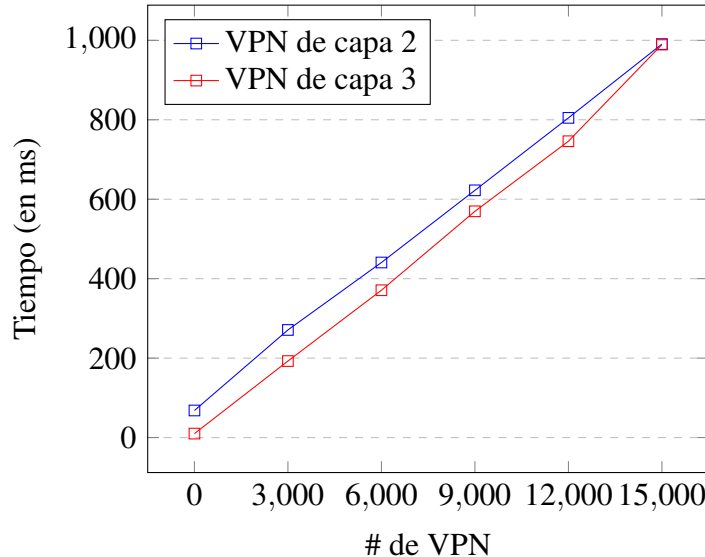
Otro objetivo de la prueba es determinar si la arquitectura, y en particular el controlador, tienen algún problema para manejar muchos servicios. En la prueba de servicios ya mencionada no se detectó ningún problema de esa índole. Sin embargo, es importante recordar que el controlador mantiene toda su 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 4.6 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 líneas 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 línea 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

Agregar un modulo de carga de topologias y numeracion IP automaatica.
Sanity checks en la topología (IP por ej.)

Bibliografía

