

# Contents

<b>1</b>	<b>Introducción</b>	<b>2</b>
<b>2</b>	<b>Estado del arte</b>	<b>3</b>
<b>3</b>	<b>Entorno virtual</b>	<b>5</b>
3.1	Requerimientos del entorno virtual . . . . .	5
3.2	¿Por qué Mininet? . . . . .	6
3.3	Diseño e implementación del entorno . . . . .	6
3.3.1	RAUController . . . . .	7
3.3.2	RAUSwitch . . . . .	8
3.3.3	QuaggaRouter . . . . .	8
3.3.4	RAUHost . . . . .	8
3.4	Modo de uso del entorno . . . . .	9
<b>4</b>	<b>Pruebas de escala</b>	<b>10</b>
4.1	Topologías de escala . . . . .	10
4.1.1	Descripción del escenario . . . . .	10
4.1.2	Resultados y observaciones . . . . .	12
4.2	Escala de servicios y flujos . . . . .	13
4.2.1	Descripción del escenario . . . . .	14
4.2.2	Resultados y observaciones . . . . .	15
<b>5</b>	<b>Conclusiones</b>	<b>19</b>
5.1	Trabajo futuro . . . . .	19
	Mencionar: Cambiar SNMP por OVS	

# Chapter 1

## Introducción

# Chapter 2

## Estado del arte

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 con-

struir 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.

# Chapter 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.

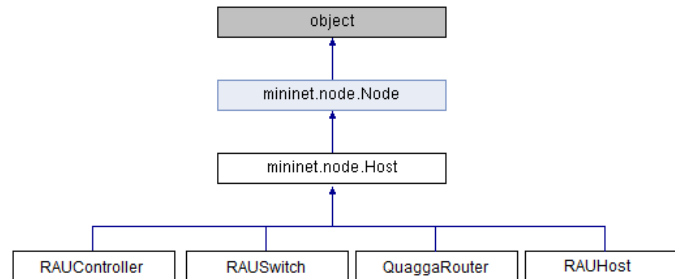
- Facilidad de configuración. Es importante que el entorno pueda generar distintas topologías y escenarios sin demasiado esfuerzo de configuración.
- Escalabilidad. Dado que uno de los objetivos es realizar pruebas de escala, el entorno debería ofrecer buena escalabilidad en la cantidad de nodos que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

Figure 3.1: Diagrama de clases del entorno.



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.

### 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: `/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 `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 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

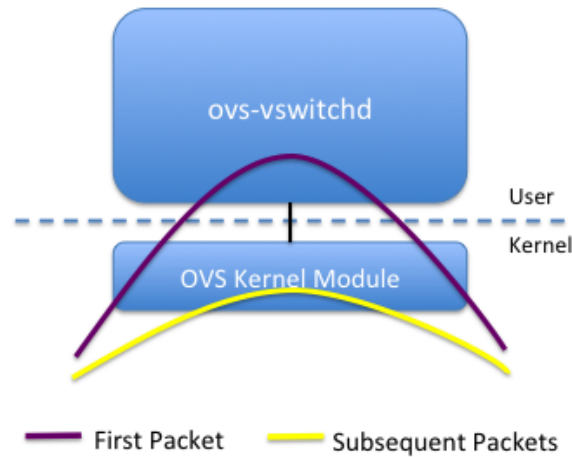
Es una clase similar al RAUSwitch pero sin OpenVSwitch, es decir, sólo usa Quagga. Apunta a representar el router CE que utilizaría una subred para conectarse a la red. Está conectado a un RAUSwitch de borde.

### 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*.



Figure 3.2: Arquitectura de OpenVSwitch.



### 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.

# Chapter 4

## Pruebas de escala

Con el entorno de simulación construido, el siguiente objetivo es realizar pruebas de escala sobre la arquitectura. Este trabajo consiste de dos etapas. La primera sección explica la primera de ellas, que es verificar que la arquitectura funciona para topologías de escala. Habiendo completado esta etapa, se pasa a 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 y Ubuntu 14.04 como sistema operativo.

### 4.1 Topologías de escala

Es importante poder asegurar que la arquitectura puede ser fácilmente migrada a redes reales. Para poder asegurar esto, es necesario comprobar que topologías con grandes cantidades de nodos no generan problemas inesperados. Dado que en el proyecto RRAP se contaba con cuatro dispositivos, este tipo de pruebas no han sido realizadas hasta el momento.

#### 4.1.1 Descripción del escenario

Este escenario consiste de una VPN punto a punto de capa 3. Dicha VPN permite tráfico de ethertype **0x0800**, es decir, tráfico del protocolo IPv4. Se utilizarán distintas topologías, pero la estructura de la prueba será similar siempre:

- 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 VPN. Esos datos se generarán con el comando *ping* y la herramienta *iperf*.

- El controlador, implementado por el RAUController. Éste se conectará con un switch 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.
- La red de RAUSwitch, conectados de acuerdo a lo que dicte cada topología.

Los aspectos que se buscan verificar con esta prueba son los siguientes:

### 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 OpenVSwitch. 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 configuran 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. Se genera tráfico utilizando el comando **ping** y la herramienta **iperf**. Con la herramienta tcpdump, se verifica el tráfico que pasa por cada nodo del camino.

Las topologías que se usarán son:

- **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.

Figure 4.1: Topología chica

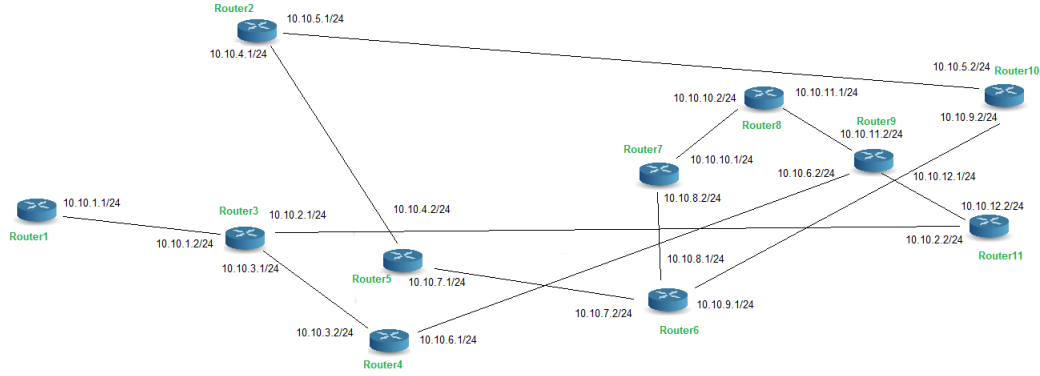
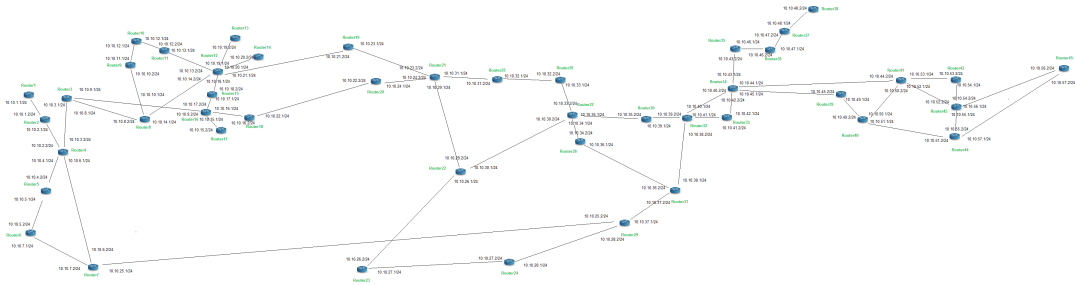


Figure 4.2: Topología mediana



- **Grande:** topología de tipo arborescente compuesta por 100 nodos.

## 4.1.2 Resultados y observaciones

En general, se observa que la aplicación no tiene problemas para manejar grandes cantidades de nodos, pero sí caminos largos. A continuación se desglosa el resultado de la prueba con cada topología. El mismo también se puede ver en la tabla 4.1. En ella se indica con una X los aspectos que funcionaron correctamente para cada caso. Los aspectos que se estudian son: se crea con éxito el servicio desde la interfaz web, el camino que se instala es correcto, los flujos de cada nodo del camino son correctos, y se clasifica correctamente el tráfico. Si se cumplen todos ellos, se puede concluir que la topología pasa la prueba.

Table 4.1: Resultados por topología.

Largo del camino	Servicio	Camino	Flujos	Clasificación de tráfico
1	X	X	X	X
7	X	X		
10				
X				

- **VPN con camino de 1 salto, en la topología básica.** La VPN se establece correctamente y el tráfico ICMP y TCP pasa sin problemas.
- **VPN con camino de 7 saltos, en la topología chica.** Los servicios se crean, y se instalan flujos en los nodos correctos, osea que el camino calculado es el más corto. Sin embargo, los flujos instalados en los nodos son incorrectos.
- **VPN con camino de 10 saltos, en la topología mediana.** La aplicación sufre una excepción de Python al crear los servicios.
- **VPN con camino de X saltos, en la topología grande.** La aplicación sufre una excepción de Python al crear los servicios, igual que el caso anterior (???).

#### Bug en código (ruta)

EXPLICAR: Error en el código que hacía que se instalaran mal los flujos en los nodos. Tenían incorrectos puertos de entrada y salida.

#### Bug en código (Dijkstra)

EXPLICAR: Error en el código del algoritmo de Dijkstra que hacía que se calcularan mal los costos, ya que se sumaba como strings (concatenación) en vez de sumar enteros.

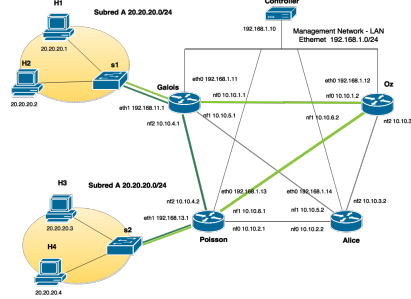
#### Problema del MTU al usar iperf

EXPLICAR: Hay que reducir 5 o 10 bytes (dependiendo de si el servicio usa 1 o 2 niveles de etiquetas MPLS) al MTU para que el tráfico pase.

## 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. Esto implica

Figure 4.3: Topología para prueba de escala de servicios.



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 podrán identificar posibles puntos de falla, o umbrales bajo los cuales debe mantenerse la red para funcionar con buen rendimiento. Dado que en esta prueba también se utilizarán topologías grandes, hay que recordar la misma es posible gracias a la corrección de los errores que se explicó en la sección anterior. También es importante recordar que aunque el entorno de simulación permite hacer un valioso estudio de escalabilidad, no generará resultados en lo que refiere a la 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

Este escenario es muy similar al anterior. 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 cabezales OpenFlow 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 y se encarga de crearlas.

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 ancho de banda debería ser afectado negativamente por la cantidad de flujos en sus tablas. El ancho de banda se medirá con la herramienta *iperf*.

### Escalabilidad en servicios

Se estudian posibles problemas que puedan tener la arquitectura de la red o la aplicación del controlador para manejar grandes cantidades de servicios e información.

Esta prueba se repite para las mismas topologías que la prueba anterior, es decir: básica (4 nodos), chica (11 nodos), mediana (45 nodos) y grande (100 nodos).

## 4.2.2 Resultados y observaciones

Table 4.2: Anchos de banda medidos para cada caso.

# de VPN	Básica	Chica	Mediana	Grande
1	X	Y	W	Z
1000	X	Y	W	Z
5000	X	Y	W	Z
10000	X	Y	W	Z
15000	X	Y	W	Z

Los resultados de la prueba con cada topología se puede ver en la tabla 4.2. Se pueden observar dos hechos interesantes. El primer resultado que se puede estudiar, es que el ancho de banda es más bajo a medida que se incrementa la cantidad de nodos y/o el largo del camino. Esto se explica desde dos ángulos. El primero es que al tener que pasar por un camino más largo, el paquete debe ser inspeccionado y reenviado más veces. Por lo tanto, este resultado debería ser visible también en un ambiente de prueba con dispositivos físicos. El segundo ángulo que explica este resultado radica en el entorno virtual mismo. Al tener que manejar más nodos virtuales, el sistema operativo anfitrión tiene menos recursos para dedicar a cada uno. Esto quiere decir que tanto las instancias de *iperf* que se encargan de recibir y enviar los paquetes, como las instancias de Open vSwitch que los procesan en cada

nodo del camino, tendrán menos tiempo de CPU para realizar su tarea.

El otro resultado que se puede observar en la tabla 4.2, es que el ancho de banda es constante para una topología, sin importar la cantidad de VPN existentes en el momento. Como se explica en el primer objetivo de esta prueba, se busca determinar si la existencia de muchos flujos en la tabla, implica que el switch OpenFlow demora más tiempo en encontrar el flujo que corresponde con un paquete entrante, y por lo tanto el mismo demora más en ser forwardado. Si esto fuera así, debería haber un impacto directo en el throughput. El máximo de VPN con el que se probó fueron 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 flujos en su tabla.

La explicación de porqué esa cantidad de flujos no afecta el ancho de banda 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 ese paquete. 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 OpenVSwitch, podemos examinar las estadísticas de la cache del datapath. Con el parámetro opcional *target* se apunta el comando a cada instancia de Open vSwitch, y por ende, a cada nodo. En las figuras 4.5 y 4.4 se observa, por un lado, la salida de `'iperf'` luego de hacer tres ejecuciones, y por otro, las estadísticas del nodo `'alice'` luego de dichas ejecuciones. 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.

Otro objetivo de la prueba es determinar si la arquitectura, y en particular la aplicación, tienen algún problema para manejar muchos servicios. No se detectó ninguna problema de esa índole. Sin ser una limitación, pero sí un factor importante, hay que recordar que los datos que maneja el controlador (entre ellos, los servicios) están en memoria. Por lo tanto se podrá agregar servicios mientras la computadora subyacente tenga suficiente memoria. La



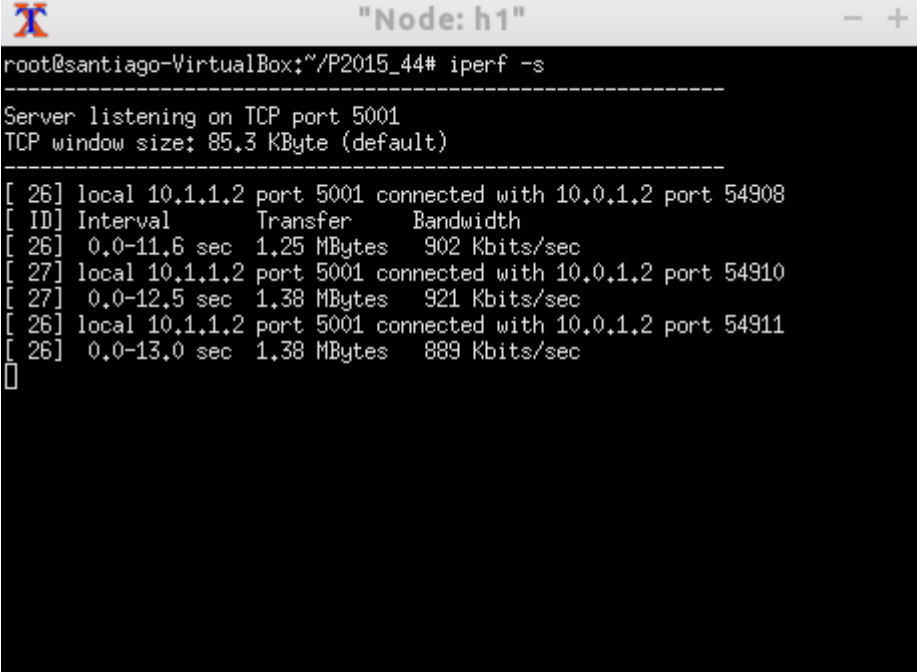
Figure 4.4: Estadísticas de cache de flujos del nodo 'alice'.



```
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#
```

creación de 15.000 servicios aumenta el consumo de memoria del controlador en 205 Mb (CONFIRMAR), por lo que un servicio ocupa alrededor de 14 Kb. A modo de ejemplo, si extrapolamos ese número a una computadora que puede dedicar 4 Gb de RAM al controlador, llegamos a que dicho controlador podrá mantener alrededor de 300.000 servicios.

Figure 4.5: Resultado de la ejecución de 3 pruebas iperf en el host h1.



```
root@santiago-VirtualBox:~/P2015_44# iperf -s
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 26] local 10.1.1.2 port 5001 connected with 10.0.1.2 port 54908
[ ID] Interval      Transfer    Bandwidth
[ 26] 0.0-11.6 sec  1.25 MBytes  902 Kbits/sec
[ 27] local 10.1.1.2 port 5001 connected with 10.0.1.2 port 54910
[ 27] 0.0-12.5 sec  1.38 MBytes  921 Kbits/sec
[ 26] local 10.1.1.2 port 5001 connected with 10.0.1.2 port 54911
[ 26] 0.0-13.0 sec  1.38 MBytes  889 Kbits/sec
□
```

# Chapter 5

## Conclusiones

### 5.1 Trabajo futuro

Agregar un modulo de carga de topologias y numeracion IP automatica.