

FACULTAD DE INGENIERÍA - U.B.A.

75.43 INTRODUCCIÓN A LOS SISTEMAS DISTRIBUIDOS
2DO. CUATRIMESTRE DE 2018

Trabajo Práctico 3: Enlace

MATIAS FELD, PADRÓN: 99170
AGUSTÍN ZORZANO, PADRÓN: 99224

Índice

1. Introducción teórica	2
2. Objetivo	2
3. Desarrollo	3
3.1. Preguntas a responder	3
3.2. Topología	4
3.3. Controlador	5
3.3.1. Configuración inicial del controlador	6
3.3.2. Direccionamiento de paquetes	6
3.3.3. Balance de carga	7
3.4. Firewall	7
3.4.1. Detectar inicio de ataques DOS	7
3.4.2. Detectar fin de ataques DOS	8
4. Pruebas realizadas	8
4.1. Ping	8
4.2. Uso de todos los enlaces	9
4.3. Wireshark	10
4.3.1. Camino del flujo de paquetes	10
4.3.2. Balance de carga	12
4.4. Iperf TCP	14
4.5. Iperf UDP	16
4.5.1. UDP sin bloqueo de firewall	16
4.5.2. UDP con bloqueo de firewall	16
5. Conclusiones	17
6. Anexos (Código)	18
6.1. Archivo topo.py	18
6.2. Archivo controller.py	19
7. Referencias	25

1. Introducción teórica

En la actualidad existe una gran cantidad de redes conformadas por switches y routers. Estas redes pueden ser de distintas dimensiones, topologías y complejidades. Además, las redes pueden cambiar a lo largo del tiempo, ya sea por cambios en los dispositivos de la red (por ejemplo el agregado de dispositivos) o por cambios manuales (por ejemplo decidir realizar Blackholing debido a un ataque).

Uno de los problemas que pueden tener las redes es la gran cantidad de dispositivos y la diversidad de fabricantes de éstos. Esto último trae como inconveniente que la configuración de cada dispositivo utiliza un protocolo diferente que depende del fabricante. Esto significa que para realizar alguna acción específica a la red, se deberá implementar para cada dispositivo un script distinto con su protocolo específico. Es debido a éste inconveniente, que aparecieron las SDN (Software-defined Networking) con la intención de facilitar e integrar el manejo administrativo de los dispositivos. Para ésto se generó una API (OpenFlow) que se maneja con un protocolo estándar y que luego lo traduce al protocolo específico de cada dispositivo.

Openflow propone la presencia de un controlador. El controlador es un elemento externo que interactúa a través de OpenFlow y se encarga de agregar o quitar entradas de la tabla del dispositivo. Este controlador puede optimizar el uso de la red, especialmente si se la compara con la utilización de Spanning Tree, ya que el controlador podrá utilizar todos los enlaces disponibles haciendo balance de cargas cuando hace el envío de paquetes (pues no tiene puertos bloqueados) y al mismo tiempo se evita el fenómeno de flooding por broadcast pues el controlador realizará spanning tree únicamente para el envío de estos mensajes. Además, permite realizar ingeniería de tráfico, ya que puede analizar cada paquete por separado y establecer distintos caminos para cada flujo.

2. Objetivo

El objetivo de este trabajo es la simulación y emulación de un pequeño datacenter, cuya red será una SDN. La topología utilizada para la construcción del datacenter será fat-tree. Se implementará un controlador que permita manejar el direccionamiento de paquetes dentro de la red, y un firewall que permita mitigar ataques DOS.

3. Desarrollo

3.1. Preguntas a responder

1. ¿Cuál es la diferencia entre un Switch y un router? ¿Qué tienen en común?

- Los switch son dispositivos de capa 2 mientras que los routers son dispositivos de capa 3.
- Los routers conectan dos redes distintas mientras que los switches son parte de una misma red.
- Los switches son transparentes para la red mientras que los routers no.
- Los routers tienen dirección IP, pero los switches no tienen dirección MAC.
- Como los switches son de capa 2, pueden realizar retransmisiones mientras que los routers no (pues IP no tiene retransmisión).
- Cuando un paquete llega a un router y éste no sabe qué hacer con él (la dirección IP destino no coincide con ninguna entrada de la tabla), el router lo descarta. En cambio, si un frame llega a un switch y éste no sabe qué hacer (la dirección MAC destino no se encuentra en la tabla del switch), envía un mensaje broadcast a todos los dispositivos para detectar a qué switch enviarlo.
- En ambos casos, los dispositivos tienen una tabla que utilizan para determinar la interfaz de salida.

2. ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow?

Los switches convencionales tienen distintos protocolos de configuración dependiendo del fabricante. Por otro lado, los switches OpenFlow son switches convencionales que entienden un único protocolo de configuración, que luego lo traducirá al protocolo propio del fabricante. Esto permite una comunicación unificada con todos los switches.

Los switches openflow se comunican con el controlador cuando no saben qué hacer con un paquete, y el controlador decide por él. En cambio, los switches normales deben decidir todo, y si no saben qué hacer envían el paquete por todas las interfaces.

3. ¿Se pueden reemplazar todos los routers de la Internet por Switches OpenFlow? Piense en el escenario interASes para elaborar su respuesta.

Para todos los routers internos de un AS no habría problema, ya que el controlador conoce toda la red y le indicará a cada uno cómo actuar frente a cada paquete.

Con respecto a los routers interASes, podrían reemplazarse por switches OpenFlow, siempre y cuando el controlador sea capaz de ejecutar el protocolo BGP.

Por un lado, el controlador debería analizar todos los mensajes BGP que se reciben, para poder informarle a toda la red las direcciones IP con las que pueden comunicarse. Por otro lado, el controlador también debe encargarse de, cada cierto tiempo, enviar los mensajes BGP a sus vecinos, como lo haría un router normal.

De esta forma, a través del controlador se podrían realizar todas las decisiones que permite el protocolo BGP, como por ejemplo la discrecionalidad en los anuncios de prefijos o elegir un camino en base al local preference.

3.2. Topología

La topología utilizada para el datacenter será fat-tree. Esta topología consiste en varios niveles, cada uno con el doble de switches que el anterior, donde todos los switches de un nivel están conectados con todos los switches del siguiente nivel. Además, cada switch que se encuentre en el último nivel estará conectado a un host proveedor de servicios, y el switch correspondiente al primer nivel estará conectado a todos los host clientes. En la figura 1 se muestra una topología con tres niveles:

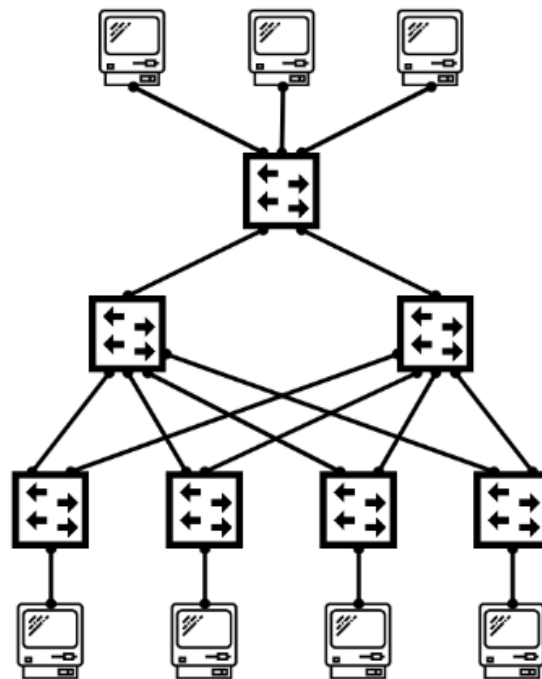


Figura 1: Topología Fat-Tree

Particularmente en la implementación realizada, se admite una cantidad variable y parametrizable tanto de niveles como de hosts clientes.

La cantidad de hosts clientes determina cuántos son los hosts conectados al switch que se encuentra en el nivel superior. Por defecto tiene un valor de 3.

La cantidad de niveles determina cuántos niveles de switches tendrá la topología, donde cada nivel tendrá 2^{n-1} switches (n es el número de nivel, comenzando en 1). Por defecto la topología tendrá 3 niveles.

Para ejecutar la topología con los valores por defecto en mininet, se puede utilizar el siguiente comando:

```
sudo mn --custom topo.py --topo topo --mac --switch ovsk --controller remote
```

Para cambiar los valores por defecto, por ejemplo 2 hosts clientes y 4 niveles, se puede utilizar el siguiente comando:

```
sudo mn --custom topo.py --topo topo,levels=4,hosts=2 --mac --switch ovsk --
      controller remote
```

En todo el desarrollo a partir de este punto se utilizarán los valores por defecto de la topología, salvo que se indique lo contrario.

Una vez inicializada la topología, utilizamos los comandos dump y links de mininet para obtener datos de la red generada, que luego se utilizaron en la herramienta online Narmox Spear[2] para graficar la topología de la red. De esta forma, se pudo verificar que la topología se creó correctamente. La figura 2 muestra la topología obtenida.

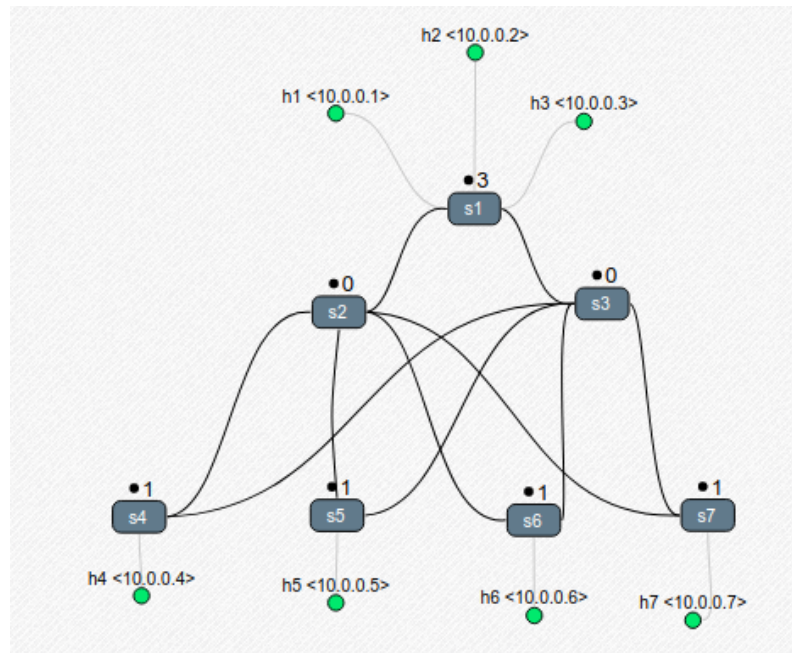


Figura 2: Topología Fat-Tree emulada

Se puede ver que la topología se creó correctamente. Se tiene una red fat-tree de tres niveles donde cada nivel se conecta con todos los switches del siguiente nivel. El switch s1 se encuentra conectado a 3 hosts y los switches s4, s5, s6 y s7 se encuentran conectados a un host cada uno.

3.3. Controlador

El controlador es el que permite que funcione el datacenter, ya que se encarga de indicarle a cada switch cómo debe actuar cuando recibe un paquete. Cuando un switch recibe un paquete, primero se fija si tiene alguna entrada en su tabla que se corresponda con el flujo del paquete. Si la hay, realiza la acción indicada en la tabla. Si no la hay, le envía el paquete al controlador, que analizará el paquete y decidirá por donde debe enviarlo el switch. Luego, ingresará una nueva entrada en la tabla del switch correspondiente al paquete, para que frente a futuros paquetes del mismo flujo el switch sepa cómo actuar.

La implementación del controlador se realizó utilizando la librería POX en Python.

Para ejecutar el controlador, se puede utilizar el siguiente comando (siempre que el archivo controller.py se encuentre en la carpeta pox/ext/):

```
./pox.py controller
```

3.3.1. Configuración inicial del controlador

Para que todo funcione correctamente, al ejecutar el controlador se inicializan algunos módulos provistos por POX para poder realizar las operaciones. Todo esto se realiza automáticamente en la función `launch()`, que se llama al ejecutar el controlador. A continuación enumeramos los módulos y explicaremos en qué se usan:

- Discovery: Este módulo se encarga de conocer toda la topología de la red, a través del intercambio de mensajes LLDP entre switches. Esto permitirá al controlador calcular todos los caminos posibles por los que un switch puede enviar un paquete para alcanzar su destino.
- Host tracker: Este módulo se encarga de conocer todos los hosts que están en la red y a qué switch están conectados. Dado que Discovery solo provee los switches de la red, era necesario contar con una forma de conocer los hosts para que al recibir un paquete se pueda conocer dónde se encuentra el host destino y así calcular el camino al mismo.
- Spanning tree: Este módulo se encarga de correr el algoritmo de spanning tree para generar un árbol sin ciclos entre los switches, y bloquear los puertos que no pertenecen al árbol al hacer un flooding de algún paquete. Aclaración: El flooding de paquetes, y por lo tanto el resultado del algoritmo spanning tree, solo se utilizará para los mensajes ARP, ya que tienen como dirección destino la correspondiente al multicast, o en algún caso particular en el que el host tracker no conozca el host destino. Para todos los demás casos, no se tendrá en cuenta el resultado del algoritmo.

Como se explicó en el último punto, en la simulación de mininet se generan mensajes ARP con dirección destino multicast, para que se puedan conocer las direcciones MAC de las IP. Para evitar que estos mensajes lleguen al controlador, en el momento en que un switch se conecta se le instala una entrada en la tabla para que haga un flooding automáticamente de estos paquetes.

3.3.2. Direccionamiento de paquetes

Como se explicó anteriormente, cuando un switch recibe un paquete y no tiene una entrada en su tabla que le indique como actuar, le delega la responsabilidad al controlador. Por lo tanto, el controlador debe decidir cómo realizar el direccionamiento de paquetes.

Cuando el controlador recibe un paquete, lo primero que hace es ver si conoce al destino. Si por alguna razón no se conoce al destino, no queda otra opción que hacer flooding del paquete. Si se conoce al destino, pueden ocurrir dos situaciones:

- Que el switch actual esté conectado al host destino, en este caso lo único que se hace es enviar el paquete al host por el puerto donde está conectado.
- Que el switch actual no esté conectado al host destino, en cuyo caso hay que calcular los caminos que puede tomar el paquete y elegir uno.

Para calcular los caminos posibles que puede tomar un paquete para llegar al destino, se realiza el siguiente algoritmo:

1. Inicializar i en 1.
2. Calcular todos los caminos que existen de longitud i , partiendo desde el switch origen.
3. Si no hay ningún camino de longitud i que finalice en el destino, volver al paso 2 con $i + 1$.
4. Filtrar todos los caminos para quedarse únicamente con los caminos que finalicen en el destino.

Una vez que se obtienen todos los caminos mínimos posibles, se elige uno (ver siguiente sección). Por último, se crea una entrada en la tabla del switch para el flujo del paquete, para que paquetes del mismo flujo sigan el mismo camino.

Se considera como flujo de un paquete al conjunto IP origen + IP destino + protocolo + puerto origen + puerto destino.

3.3.3. Balance de carga

Cuando hay múltiples caminos posibles para enviar un paquete, el controlador debe decidir cuál de ellos usar, asegurándose de que se vayan usando todos los enlaces disponibles para aprovechar al máximo la topología. Esto es lo que se denomina como balance de carga.

Para asegurarnos que el controlador vaya distribuyendo los flujos por los distintos enlaces, mantendremos para cada puerto del switch un contador, indicando cuántas veces se utilizó ese puerto. De esta forma, el controlador elegirá el camino cuyo puerto de salida sea el que menos veces se ha utilizado.

3.4. Firewall

El firewall es el encargado de mitigar los ataques de DOS. Se entiende por ataque de DOS cuando se recibe una cantidad considerable de paquetes en un determinado periodo de tiempo.

Cuando el firewall determina que hay un ataque de DOS hacia un host destino, lo que hará es descartar todos los paquetes UDP que se dirijan hacia ese destino. Para ello, insertará en la tabla de todos los switches una entrada que le indique al switch que debe descartar el paquete UDP.

Cuando se determina que el ataque terminó, se eliminara esa entrada de todas las tablas para que se permita nuevamente el flujo normal de paquetes UDP.

3.4.1. Detectar inicio de ataques DOS

Para detectar los ataques DOS, el firewall usará los datos estadísticos de las entradas en las tablas de los switches, en particular el dato de cuántos paquetes se enviaron con cada flujo. Para ello, cada cierto tiempo el controlador solicitará a los switches que le envíen las estadísticas y comparará la cantidad de paquetes UDP enviados en ese switch en ese periodo con la cantidad enviada en el periodo anterior. Si en algún switch esa diferencia es mayor al límite predefinido, entonces se determina que la red se encuentra bajo un ataque de DOS.

3.4.2. Detectar fin de ataques DOS

Mientras la red se encuentre bajo ataque, el firewall continuará chequeando los datos estadísticos de las tablas de los switches. Cuando en todos los switches la cantidad de paquetes UDP sea menor al límite, entonces se procederá a desbloquear los flujos UDP. Sin embargo, no se desbloqueará inmediatamente, deben pasar por lo menos 5 periodos seguidos donde la cantidad de paquetes sea baja para que se desbloqueen los flujos UDP. El contador de periodos se reiniciará cada vez que se detecte una alta tasa de paquetes UDP.

4. Pruebas realizadas

4.1. Ping

La primer prueba realizada consiste en realizar un ping entre todos los hosts. La idea es comprobar que el controlador direcciona correctamente los paquetes ICMP para que lleguen al destino. Cuando un ping entre dos hosts es exitoso, significa que los paquetes pudieron enviarse y recibirse en ambos sentidos. Si algún ping falla, la prueba se considera fallida.

En el primer ping se comprobó que todos los hosts pudieron comunicarse con todos y no hubo pérdida de paquetes.

```
mininet> time pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7
h2 -> h1 h3 h4 h5 h6 h7
h3 -> h1 h2 h4 h5 h6 h7
h4 -> h1 h2 h3 h5 h6 h7
h5 -> h1 h2 h3 h4 h6 h7
h6 -> h1 h2 h3 h4 h5 h7
h7 -> h1 h2 h3 h4 h5 h6
*** Results: 0% dropped (42/42 received)
*** Elapsed time: 3.972326 secs
```

Figura 3: Primer pingall

Se puede observar que el tiempo que llevó es bastante alto. Esto es así porque ningún switch tenía entradas en su tabla, por lo que todos los paquetes tuvieron que pasar por el controlador para que calcule el camino y llene las tablas.

Para comprobarlo, corrimos nuevamente el comando, y como ahora las tablas estaban llenas, se realizó mucho más rápido porque el controlador no intervino.

<pre>mininet> time pingall *** Ping: testing ping reachability h1 -> h2 h3 h4 h5 h6 h7 h2 -> h1 h3 h4 h5 h6 h7 h3 -> h1 h2 h4 h5 h6 h7 h4 -> h1 h2 h3 h5 h6 h7 h5 -> h1 h2 h3 h4 h6 h7 h6 -> h1 h2 h3 h4 h5 h7 h7 -> h1 h2 h3 h4 h5 h6 *** Results: 0% dropped (42/42 received) *** Elapsed time: 0.150791 secs</pre>	<pre>mininet> time pingall *** Ping: testing ping reachability h1 -> h2 h3 h4 h5 h6 h7 h2 -> h1 h3 h4 h5 h6 h7 h3 -> h1 h2 h4 h5 h6 h7 h4 -> h1 h2 h3 h5 h6 h7 h5 -> h1 h2 h3 h4 h6 h7 h6 -> h1 h2 h3 h4 h5 h7 h7 -> h1 h2 h3 h4 h5 h6 *** Results: 0% dropped (42/42 received) *** Elapsed time: 0.144178 secs</pre>
---	---

Figura 4: Segundo pingall

Para estar seguros de que funciona todo, probamos realizar el ping en una topología de 4 niveles, ya que tiene muchos más switches, y por lo tanto más caminos posibles y más hosts:

```

mininet> time pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10
*** Results: 0% dropped (110/110 received)
*** Elapsed time: 12.126626 secs

mininet> time pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10
*** Results: 0% dropped (110/110 received)
*** Elapsed time: 0.383906 secs

```

Figura 5: Pingall en topología de 4 niveles

En este caso es aún más notorio el hecho de que se llena la tabla de los switches, y se ve en la diferencia entre el tiempo del primer ping y el segundo.

4.2. Uso de todos los enlaces

Como se explicó en secciones anteriores, el controlador debe asegurarse de distribuir los flujos de paquetes en los distintos enlaces. Por lo tanto, debemos asegurarnos que se usen todos los puertos.

Para eso, aprovechando que el controlador guarda la cantidad de veces que se usa cada puerto del switch, utilizamos esa información para ir chequeando cada cierto tiempo la cantidad de puertos usados. La información del uso de los puertos se muestra a través de los logs.

En la figura 6 se observa que al iniciar la red ningún puerto se encuentra en uso debido a que no se ha mandado ningún mensaje.

```

[controller] Switch 00-00-00-00-00-01: Ports used: 0/5
[controller] Switch 00-00-00-00-00-02: Ports used: 0/5
[controller] Switch 00-00-00-00-00-03: Ports used: 0/5
[controller] Switch 00-00-00-00-00-04: Ports used: 0/3
[controller] Switch 00-00-00-00-00-05: Ports used: 0/3
[controller] Switch 00-00-00-00-00-06: Ports used: 0/3
[controller] Switch 00-00-00-00-00-07: Ports used: 0/3

```

Figura 6: Puertos usados al inicio de la red

A medida que vamos haciendo algunos pings, la cantidad de puertos usados va aumentando. Esto se puede observar en la figura 7.

```

[controller] Switch 00-00-00-00-00-01: Ports used: 4/5
[controller] Switch 00-00-00-00-00-02: Ports used: 3/5
[controller] Switch 00-00-00-00-00-03: Ports used: 2/5
[controller] Switch 00-00-00-00-00-04: Ports used: 2/3
[controller] Switch 00-00-00-00-00-05: Ports used: 0/3
[controller] Switch 00-00-00-00-00-06: Ports used: 2/3
[controller] Switch 00-00-00-00-00-07: Ports used: 2/3

```

Figura 7: Utilización de puertos

Por último, cuando se realizan todos los pings, todos los puertos de todos los switches están en uso:

```
[controller] Switch 00-00-00-00-00-01: All ports used
[controller] Switch 00-00-00-00-00-02: All ports used
[controller] Switch 00-00-00-00-00-03: All ports used
[controller] Switch 00-00-00-00-00-04: All ports used
[controller] Switch 00-00-00-00-00-05: All ports used
[controller] Switch 00-00-00-00-00-06: All ports used
[controller] Switch 00-00-00-00-00-07: All ports used
```

Figura 8: Utilización de todos los puertos

Por lo tanto, podemos confirmar que el controlador distribuye la carga en los distintos enlaces y no usa el árbol obtenido con el spanning tree.

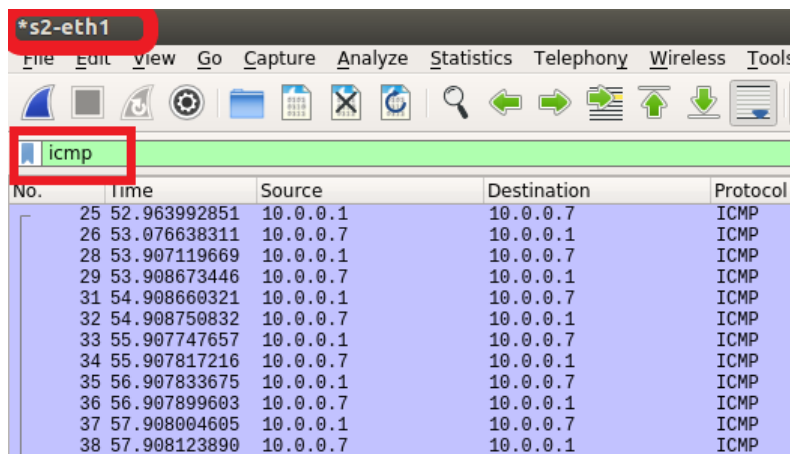
4.3. Wireshark

4.3.1. Camino del flujo de paquetes

Utilizaremos el programa wireshark para comprobar que todos los paquetes del mismo flujo sigan el mismo camino y pasen por los mismos switches. Observando la figura de la topología, se puede ver que cualquier paquete enviado desde un host cliente hacia un host servidor pasa por el switch numero 1, y desde ahí siempre tiene dos opciones, ir al switch 2 o ir al switch 3. En todos los casos ambos switches permiten al paquete llegar al destino.

Para comprobar que para un mismo flujo siempre se usa el mismo camino, realizaremos un ping entre un host cliente y un host servidor, utilizaremos wireshark en las interfaces de entrada tanto del switch 2 como del switch 3, y comprobaremos que en una de las dos interfaces no habrá ningún paquete icmp.

La figura 9 muestra la salida del wireshark con filtro en paquetes icmp para el switch 2:



No.	Time	Source	Destination	Protocol
25	52.963992851	10.0.0.1	10.0.0.7	ICMP
26	53.076638311	10.0.0.7	10.0.0.1	ICMP
28	53.907119669	10.0.0.1	10.0.0.7	ICMP
29	53.908673446	10.0.0.7	10.0.0.1	ICMP
31	54.908660321	10.0.0.1	10.0.0.7	ICMP
32	54.908750832	10.0.0.7	10.0.0.1	ICMP
33	55.907747657	10.0.0.1	10.0.0.7	ICMP
34	55.907817216	10.0.0.7	10.0.0.1	ICMP
35	56.907833675	10.0.0.1	10.0.0.7	ICMP
36	56.907899603	10.0.0.7	10.0.0.1	ICMP
37	57.908004605	10.0.0.1	10.0.0.7	ICMP
38	57.908123890	10.0.0.7	10.0.0.1	ICMP

Figura 9: Paquetes ICMP en S2

La figura 10 muestra la salida del wireshark con filtro en paquetes icmp para el switch 3:

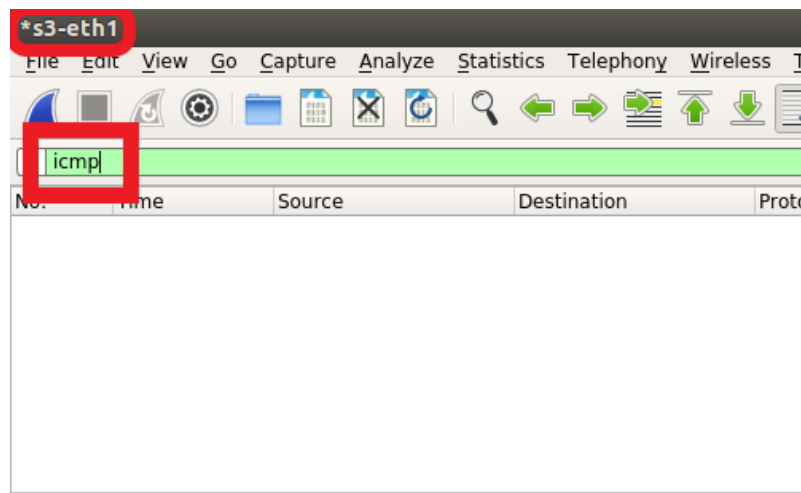


Figura 10: Paquetes ICMP en S3

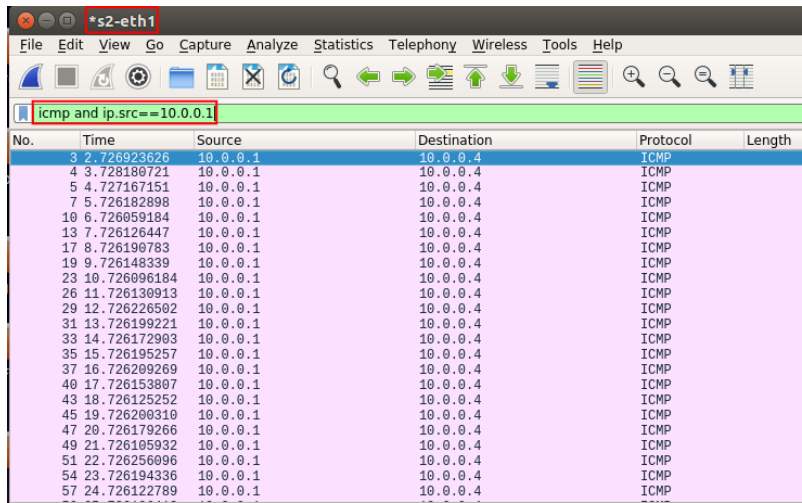
Se puede confirmar que los paquetes siempre toman el mismo camino, y en este caso el controlador decidió que sea a través del switch 2.

4.3.2. Balance de carga

Utilizaremos el programa wireshark para comprobar que los switches realicen balance de carga. Como se mencionó anteriormente, todos los paquetes provenientes de h1, h2 o h3 pasaran por el switch s1 y luego podrán ir por s2 o s3. Por lo tanto, el switch s1 deberá realizar balance de carga puesto que los pesos son iguales por ambos caminos.

Para comprobar que el switch s1 realiza balance de carga, realizaremos por un lado un ping entre el host h1 y el host h4, y por otro lado un ping entre h2 y h4. Utilizando wireshark en las interfaces de entrada de s2 y s3, comprobaremos si el switch s1 realizó balance de carga.

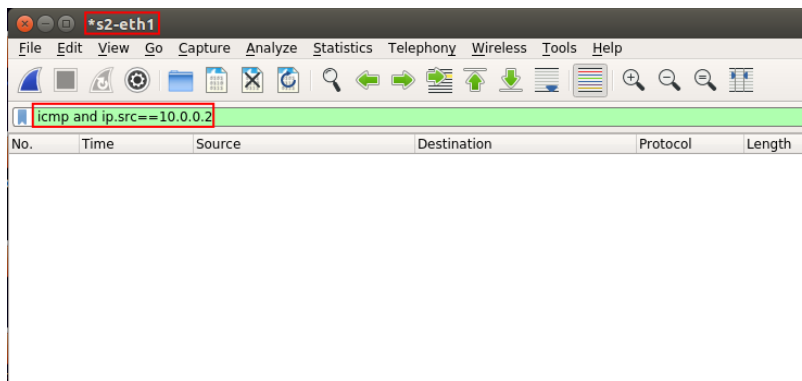
Las figura 11 y 12 muestra la salida del wireshark para el switch 2. Se puede observar que los paquetes provenientes del host h1 pasaron por el switch s2 mientras que los paquetes provenientes del host h2 no pasaron por el switch s2.



The image shows a Wireshark packet capture on interface s2-eth1. The filter is 'icmp and ip.src==10.0.0.1'. The packet list shows 24 ICMP Echo (ping) requests from source 10.0.0.1 to destination 10.0.0.4. The packets are numbered 3 through 27 in the list, with timestamps ranging from 2.726923626 to 24.726122789 seconds.

No.	Time	Source	Destination	Protocol	Length
3	2.726923626	10.0.0.1	10.0.0.4	ICMP	
4	3.728180721	10.0.0.1	10.0.0.4	ICMP	
5	4.727167151	10.0.0.1	10.0.0.4	ICMP	
7	5.726182898	10.0.0.1	10.0.0.4	ICMP	
10	6.726059184	10.0.0.1	10.0.0.4	ICMP	
13	7.726126447	10.0.0.1	10.0.0.4	ICMP	
17	8.726190783	10.0.0.1	10.0.0.4	ICMP	
19	9.726148339	10.0.0.1	10.0.0.4	ICMP	
23	10.726096184	10.0.0.1	10.0.0.4	ICMP	
26	11.726130913	10.0.0.1	10.0.0.4	ICMP	
29	12.726226502	10.0.0.1	10.0.0.4	ICMP	
31	13.726199221	10.0.0.1	10.0.0.4	ICMP	
33	14.726172903	10.0.0.1	10.0.0.4	ICMP	
35	15.726195257	10.0.0.1	10.0.0.4	ICMP	
37	16.726209269	10.0.0.1	10.0.0.4	ICMP	
40	17.726153807	10.0.0.1	10.0.0.4	ICMP	
43	18.726125252	10.0.0.1	10.0.0.4	ICMP	
45	19.726200310	10.0.0.1	10.0.0.4	ICMP	
47	20.726179266	10.0.0.1	10.0.0.4	ICMP	
49	21.726105932	10.0.0.1	10.0.0.4	ICMP	
51	22.726256096	10.0.0.1	10.0.0.4	ICMP	
54	23.726194336	10.0.0.1	10.0.0.4	ICMP	
57	24.726122789	10.0.0.1	10.0.0.4	ICMP	

Figura 11: Balance de carga-paquetes en S2



The image shows a Wireshark packet capture on interface s2-eth1. The filter is 'icmp and ip.src==10.0.0.2'. The packet list is empty, indicating that no packets were captured from source 10.0.0.2.

No.	Time	Source	Destination	Protocol	Length
-----	------	--------	-------------	----------	--------

Figura 12: Balance de carga-paquetes en S2

La figura 13 y 14 muestra la salida del wireshark para el switch 3. Se puede observar que, a diferencia del caso anterior, los paquetes provenientes del host h1 no pasaron por el switch s3 mientras que los paquetes provenientes del host h2 si pasaron por el switch s3.

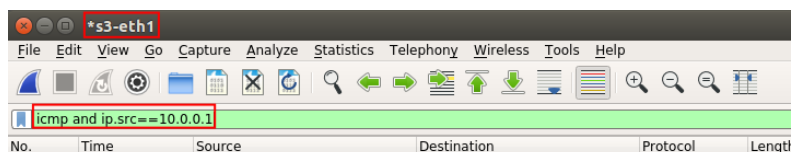


Figura 13: Balance de carga-paquetes en S3

The image shows the Wireshark interface with the filter 'icmp and ip.src==10.0.0.2' applied. The packet list shows 30 ICMP packets from 10.0.0.2 to 10.0.0.4, indicating traffic from host h2 was captured on this interface.

No.	Time	Source	Destination	Protocol	Length
7	8.191149940	10.0.0.2	10.0.0.4	ICMP	
9	9.192204845	10.0.0.2	10.0.0.4	ICMP	
12	10.191488774	10.0.0.2	10.0.0.4	ICMP	
16	11.191468262	10.0.0.2	10.0.0.4	ICMP	
19	12.191538848	10.0.0.2	10.0.0.4	ICMP	
21	13.191471882	10.0.0.2	10.0.0.4	ICMP	
25	14.191454579	10.0.0.2	10.0.0.4	ICMP	
27	15.191432525	10.0.0.2	10.0.0.4	ICMP	
30	16.191521243	10.0.0.2	10.0.0.4	ICMP	
32	17.191581816	10.0.0.2	10.0.0.4	ICMP	
35	18.191528028	10.0.0.2	10.0.0.4	ICMP	
37	19.191496793	10.0.0.2	10.0.0.4	ICMP	
39	20.191492092	10.0.0.2	10.0.0.4	ICMP	
41	21.191492710	10.0.0.2	10.0.0.4	ICMP	
44	22.191476746	10.0.0.2	10.0.0.4	ICMP	
46	23.191541529	10.0.0.2	10.0.0.4	ICMP	
49	24.191540987	10.0.0.2	10.0.0.4	ICMP	
51	25.191421717	10.0.0.2	10.0.0.4	ICMP	
53	26.191571769	10.0.0.2	10.0.0.4	ICMP	
55	27.191539843	10.0.0.2	10.0.0.4	ICMP	
58	28.191489302	10.0.0.2	10.0.0.4	ICMP	
61	29.191559761	10.0.0.2	10.0.0.4	ICMP	
65	30.191542924	10.0.0.2	10.0.0.4	ICMP	

Figura 14: Balance de carga-paquetes en S3

Por lo tanto, se puede confirmar que el switch s1 realizó balance de carga para flujos distintos.

4.4. Iperf TCP

Esta prueba consiste en realizar una conexión TCP entre un host cliente y un host servidor, y comprobar que la transferencia de mensajes entre ellos es exitosa. Para eso, se utilizará el programa Iperf para establecer una conexión TCP entre ambos hosts.

Cliente:

```
"Node: h1"
root@mati:~# iperf -c 10.0.0.7 -p 80
-----
Client connecting to 10.0.0.7, TCP port 80
TCP window size: 85.3 KByte (default)
-----
[ 35] local 10.0.0.1 port 56174 connected with 10.0.0.7 port 80
[ ID] Interval      Transfer    Bandwidth
[ 35] 0.0-10.0 sec  8.41 GBytes 7.22 Gbits/sec
root@mati:~#
```

Figura 15: Cliente

Servidor:

```
"Node: h7"
root@mati:~# iperf -s -p 80
-----
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
-----
[ 36] local 10.0.0.7 port 80 connected with 10.0.0.1 port 56174
[ ID] Interval      Transfer    Bandwidth
[ 36] 0.0-10.0 sec  8.41 GBytes 7.22 Gbits/sec

```

Figura 16: Servidor

Como se ve, la conexión fue exitosa y se pudieron comunicar ambos hosts.

Al igual que el caso anterior, utilizamos wireshark para comprobar que todos los mensajes de la conexión (que pertenecen al mismo flujo) sigan el mismo camino.

La figura 17 muestra la salida del wireshark con filtro en paquetes tcp para el switch 2:

***s2-eth1**

Filter: `ip.dst == 10.0.0.7 and tcp`

No.	Time	Source	Destination	Protocol
22667	88.236956867	10.0.0.1	10.0.0.7	TCP
22668	88.236961057	10.0.0.1	10.0.0.7	TCP
22669	88.236963292	10.0.0.1	10.0.0.7	TCP
22670	88.236969438	10.0.0.1	10.0.0.7	TCP
22671	88.236975304	10.0.0.1	10.0.0.7	TCP
22673	88.239356509	10.0.0.1	10.0.0.7	TCP
22674	88.239360420	10.0.0.1	10.0.0.7	TCP
22675	88.239364611	10.0.0.1	10.0.0.7	TCP
22676	88.239368522	10.0.0.1	10.0.0.7	TCP
22677	88.239372433	10.0.0.1	10.0.0.7	TCP
22678	88.239376343	10.0.0.1	10.0.0.7	TCP
22679	88.239380534	10.0.0.1	10.0.0.7	TCP
22680	88.239384445	10.0.0.1	10.0.0.7	TCP
22681	88.239388076	10.0.0.1	10.0.0.7	TCP
22682	88.239391987	10.0.0.1	10.0.0.7	TCP
22683	88.239395619	10.0.0.1	10.0.0.7	TCP
22685	88.240375031	10.0.0.1	10.0.0.7	TCP
22686	88.240379500	10.0.0.1	10.0.0.7	TCP
22687	88.240383411	10.0.0.1	10.0.0.7	TCP
22688	88.240387601	10.0.0.1	10.0.0.7	TCP
22689	88.240391512	10.0.0.1	10.0.0.7	TCP
22690	88.240395423	10.0.0.1	10.0.0.7	TCP
22691	88.240399334	10.0.0.1	10.0.0.7	TCP
22692	88.240403525	10.0.0.1	10.0.0.7	TCP

▶ Frame 213: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
 ▶ Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:
 ▶ Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.7
 ▶ Transmission Control Protocol, Src Port: 56174, Dst Port: 80, Seq: 0

Figura 17: Paquetes TCP en S2

La figura 18 muestra la salida del wireshark con filtro en paquetes tcp para el switch 3:

***s3-eth1**

Filter: `ip.dst == 10.0.0.7 and tcp`

No.	Time	Source	Destination	Protocol	Leng
-----	------	--------	-------------	----------	------

Figura 18: Paquetes TCP en S3

Otra vez se puede confirmar que todos los paquetes del flujo siguen el mismo camino.

4.5. Iperf UDP

Con las pruebas UDP comprobaremos por un lado que los datagramas puedan ser enviados entre dos hosts, y por otro lado que el firewall funcione correctamente y bloquee los paquetes UDP si la tasa de paquetes es muy alta.

4.5.1. UDP sin bloqueo de firewall

En este caso se enviará un datagrama desde un host cliente hacia un host servidor. El resultado debería ser que el servidor recibió correctamente el datagrama.

```
"Node: h1"
root@mati:~# iperf -u -c 10.0.0.7 -p 80 --stdin
-----
Client connecting to 10.0.0.7, UDP port 80
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 35] local 10.0.0.1 port 60488 connected with 10.0.0.7 port 80
hola
[ ID] Interval      Transfer    Bandwidth
[ 35] 0.0-35.5 sec  1.44 KBytes  332 bits/sec
[ 35] Sent 1 datagrams
[ 35] Server Report:
[ 35] 0.0- 0.2 sec  1.44 KBytes  60.7 Kbits/sec   0.000 ms   0/   1 (0%)
root@mati:~#
```

Figura 19: Envío de un datagrama UDP

En la figura 19 se puede ver el reporte del servidor en el cual se indica que el datagrama se recibió correctamente.

4.5.2. UDP con bloqueo de firewall

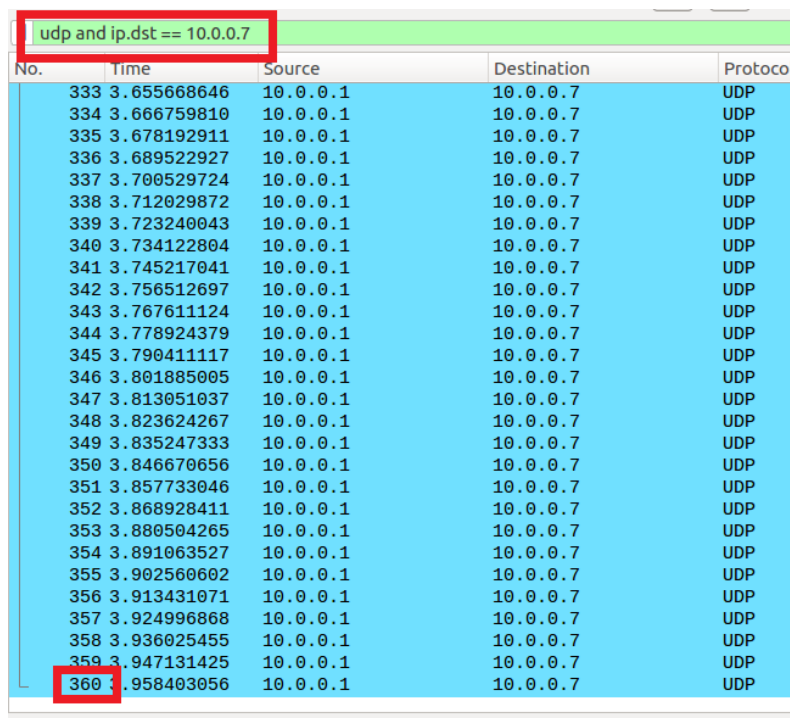
En este caso se enviará una cantidad alta de datagramas, por lo que el firewall debería detectar el ataque DOS y bloquear todos los paquetes UDP.

```
root@mati:~# iperf -u -c 10.0.0.7 -p 80
-----
Client connecting to 10.0.0.7, UDP port 80
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 35] local 10.0.0.1 port 40358 connected with 10.0.0.7 port 80
[ ID] Interval      Transfer    Bandwidth
[ 35] 0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 35] Sent 893 datagrams
[ 35] WARNING: did not receive ack of last datagram after 10 tries.
root@mati:~#
```

Figura 20: Envío de 893 datagramas UDP

Como se observa en la figura 20, no hay reporte del servidor. Esto es porque no recibió los 893 datagramas que el cliente envió. Por lo tanto, podemos confirmar que el firewall bloquea correctamente los paquetes UDP.

En la figura 21 se puede comprobar que efectivamente el servidor no recibe todos los datagramas, sino que recibe solamente los primeros 360.



No.	Time	Source	Destination	Protocol
333	3.655668646	10.0.0.1	10.0.0.7	UDP
334	3.666759810	10.0.0.1	10.0.0.7	UDP
335	3.678192911	10.0.0.1	10.0.0.7	UDP
336	3.689522927	10.0.0.1	10.0.0.7	UDP
337	3.700529724	10.0.0.1	10.0.0.7	UDP
338	3.712029872	10.0.0.1	10.0.0.7	UDP
339	3.723240043	10.0.0.1	10.0.0.7	UDP
340	3.734122804	10.0.0.1	10.0.0.7	UDP
341	3.745217041	10.0.0.1	10.0.0.7	UDP
342	3.756512697	10.0.0.1	10.0.0.7	UDP
343	3.767611124	10.0.0.1	10.0.0.7	UDP
344	3.778924379	10.0.0.1	10.0.0.7	UDP
345	3.790411117	10.0.0.1	10.0.0.7	UDP
346	3.801885005	10.0.0.1	10.0.0.7	UDP
347	3.813051037	10.0.0.1	10.0.0.7	UDP
348	3.823624267	10.0.0.1	10.0.0.7	UDP
349	3.835247333	10.0.0.1	10.0.0.7	UDP
350	3.846670656	10.0.0.1	10.0.0.7	UDP
351	3.857733046	10.0.0.1	10.0.0.7	UDP
352	3.868928411	10.0.0.1	10.0.0.7	UDP
353	3.880504265	10.0.0.1	10.0.0.7	UDP
354	3.891063527	10.0.0.1	10.0.0.7	UDP
355	3.902560602	10.0.0.1	10.0.0.7	UDP
356	3.913431071	10.0.0.1	10.0.0.7	UDP
357	3.924996868	10.0.0.1	10.0.0.7	UDP
358	3.936025455	10.0.0.1	10.0.0.7	UDP
359	3.947131425	10.0.0.1	10.0.0.7	UDP
360	3.958403056	10.0.0.1	10.0.0.7	UDP

Figura 21: Datagramas recibidos antes del bloqueo

5. Conclusiones

Con la realización de este trabajo pudimos observar las ventajas que trae la utilización de switches OpenFlow. Mediante el uso de un controlador, se pueden establecer reglas para aplicar en caso de ataques de forma rápida, sencilla y automática, pudiendo generar así un firewall inteligente y que además puede ser modificado o adaptado de forma sencilla a toda la red. Se pueden establecer reglas para mitigar las congestiones obteniendo así un mejor aprovechamiento de la red, realizando balance de carga entre los distintos enlaces disponibles. También, al no utilizar spanning tree de forma permanente (solo se usa para los mensajes multicast), se tienen todos los enlaces disponibles por lo que la red se puede aprovechar en su totalidad pudiendo disminuir la congestión y optimizar el tráfico.

Por otro lado, aunque no se realizó en este trabajo, se podría realizar ingeniería de tráfico muy fácilmente, de forma similar a lo realizado para el balance de carga, ya que el controlador puede analizar cada flujo de paquetes por separados e indicar distintas acciones para cada uno.

Por lo tanto, se puede concluir que el uso de SDN permite mejorar el rendimiento de las redes, aumentar su seguridad y aplicar cambios o nuevas políticas de forma rápida y sencilla a toda la red o parte de la red.

6. Anexos (Código)

6.1. Archivo topo.py

```

1 from mininet.topo import Topo
2 import sys
3
4 class MyTopo(Topo):
5     def __init__(self, levels=3, hosts=3):
6         # Initialize topology
7         Topo.__init__(self)
8
9         switches = {}
10
11         switch_number = 1
12         host_number = 1
13
14         #Add client hosts
15         switches[0] = []
16         for host in range(hosts):
17             name = 'h' + str(host_number)
18             switches[0].append(self.addHost(name))
19             host_number += 1
20
21         #Add switches
22         for level in range(levels):
23             switch_count = 2 ** level
24             switches[level+1] = []
25             for switch in range(switch_count):
26                 name = 's' + str(switch_number)
27                 addedSwitch = self.addSwitch(name)
28                 switches[level+1].append(addedSwitch)
29                 switch_number += 1
30                 for link in switches[level]:
31                     self.addLink(addedSwitch, link)
32
33         #Add server hosts
34         for switch in switches[levels]:
35             name = 'h' + str(host_number)
36             host = self.addHost(name)
37             host_number += 1
38             self.addLink(switch, host)
39
40
41
42
43
44 topos = {'topo': (lambda levels=3, hosts=3: MyTopo(levels=levels, hosts=hosts))
45           }

```

6.2. Archivo controller.py

```

1 from pox.core import core
2 import pox.openflow.libopenflow_01 as of
3 import pox.lib.packet as pkt
4 from pox.lib.addresses import EthAddr, IPAddr
5 import pox.lib.util as poxutil
6 import pox.lib.packet as pkt
7 from pox.lib.revent import *
8 import pox.lib.recoco as recoco
9 from pox.openflow.discovery import Discovery
10 from pox.lib.util import dpidToStr
11 from pox.lib.recoco import Timer
12 from pox.host_tracker import host_tracker
13
14
15 # Create a logger for this component
16 log = core.getLogger()
17 ports_used = {}
18
19 class MyController(EventMixin):
20
21     def __init__(self):
22         core.openflow.addListeners(self)
23
24     def _handle_ConnectionUp(self, event):
25         #Flood multicast packets => arp
26         log.info("Flooding multicast and arp packets in switch: " + dpidToStr(
27             event.connection.dpid))
28
29         #multicast
30         msg = of.ofp_flow_mod()
31         msg.match.dl_dst = pkt.ETHER_BROADCAST
32         msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
33         event.connection.send(msg)
34
35         #arp
36         msg = of.ofp_flow_mod()
37         msg.match.dl_type = pkt.ethernet.ARP_TYPE
38         msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
39         event.connection.send(msg)
40
41         #drop ipv6 packets because they are strange and broken
42         msg = of.ofp_flow_mod()
43         msg.match.dl_type = pkt.ethernet.IPV6_TYPE
44         event.connection.send(msg)
45
46         ports_used[event.dpid] = {}
47         for port in event.connection.ports:
48             ports_used[event.dpid][port] = 0
49
50     def _handle_PacketIn(self, event):
51         """ Packet processing """
52         packet = event.parsed
53         dpid = event.connection.dpid
54
55         eth_packet = packet.find(pkt.ethernet)
56         ip_packet = packet.find(pkt.ipv4)
57         icmp_packet = packet.find(pkt.icmp)

```

```

58     tcp_packet = packet.find(pkt.tcp)
59     udp_packet = packet.find(pkt.udp)
60
61     if icmp_packet is None and tcp_packet is None and udp_packet is None:
62         return
63
64     def flood():
65         #Flood incoming packet if dst is not known yet
66         #Do not update flow table
67         log.info("Flooding packet in switch: " + dpidToStr(event.connection
68             .dpid) + " --- dst=" + str(eth_packet.dst))
69         msg = of.ofp_packet_out()
70         msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
71         msg.data = event.ofp
72         msg.in_port = event.port
73         event.connection.send(msg)
74
75     dstEntry = core.host_tracker.getMacEntry(eth_packet.dst)
76     if dstEntry is None:
77         return flood()
78
79     log.info("Calculating packet path in switch: " + dpidToStr(event.
80         connection.dpid) + " --- dst=" + str(eth_packet.dst))
81
82     dst = dstEntry.dpid
83     port = None
84
85     if (dst == dpid):
86         #current switch is destination switch
87         port = dstEntry.port
88     else:
89         #calculate all possible minimum paths
90         paths = [[neighbour] for neighbour in self.getNeighbours(dpid)]
91         dsts = self.getPathsToDst(paths, dst)
92         while not dsts:
93             #for each iteration, calculates all paths from src which has
94             #length n
95             #if any of those paths end in dst, finish while
96             oldPaths = paths
97             paths = []
98             for path in oldPaths:
99                 neighbours = self.getNeighbours(path[-1].dpid2)
100                 for neighbour in neighbours:
101                     paths.append(path + [neighbour])
102                 dsts = self.getPathsToDst(paths, dst)
103
104         if len(dsts) == 0:
105             return
106
107         else:
108             #dsts has all possible minimum paths to dst
109             ports = [dstPath[0].port1 for dstPath in dsts]
110             port = self.getLeastUsedPort(ports, dpid)
111
112     text = "Making rule for sending packet in switch: " + dpidToStr(dpid) +
113         '\n'

```

```

114         text += "Ethernet:_" + str(eth_packet.src) + "_->_" + str(eth_packet.
115             dst) + '\n'
116
117         #update flow table
118         msg = of.ofp_flow_mod()
119         msg.match.dl_type = eth_packet.type
120         msg.match.nw_src = ip_packet.srcip
121         msg.match.nw_dst = ip_packet.dstip
122         msg.match.nw_proto = ip_packet.protocol
123         text += "IPv4:_" + str(ip_packet.srcip) + "_->_" + str(ip_packet.dstip)
124         if tcp_packet is not None:
125             msg.match.tp_src = tcp_packet.srcport
126             msg.match.tp_dst = tcp_packet.dstport
127             text += "\nTCP:_" + str(tcp_packet.srcport) + "_->_" + str(
128                 tcp_packet.dstport)
129         if udp_packet is not None:
130             msg.match.tp_src = udp_packet.srcport
131             msg.match.tp_dst = udp_packet.dstport
132             text += "\nUDP:_" + str(udp_packet.srcport) + "_->_" + str(
133                 udp_packet.dstport)
134
135         msg.actions.append(of.ofp_action_output(port = port))
136         event.connection.send(msg)
137         print text
138
139         ports_used[dpid][port] += 1
140
141         #send packet
142         msg = of.ofp_packet_out()
143         msg.actions.append(of.ofp_action_output(port = port))
144         msg.data = event.ofp
145         msg.in_port = event.port
146         event.connection.send(msg)
147
148
149     def getNeighbours(self, dpid):
150         """Returns all neighbours of switch with dpid"""
151         neighbours = []
152         for adjacency in core.openflow_discovery.adjacency:
153             if adjacency.dpid1 == dpid:
154                 neighbours.append(adjacency)
155         return neighbours
156
157     def getPathsToDst(self, paths, dst):
158         """Returns all paths from list which end with dst"""
159         dstPaths = []
160         for path in paths:
161             if path[-1].dpid2 == dst:
162                 dstPaths.append(path)
163         return dstPaths
164
165     def getLeastUsedPort(self, ports, dpid):
166         minUses = float('Inf')
167         minPort = 0
168         for port in ports:
169             uses = ports_used[dpid][port]
170             if uses < minUses:
171                 minUses = uses
172                 minPort = port
173         return minPort

```

```

171
172
173
174 class MyFirewall(EventMixin):
175
176     udp_max_packet = 100
177     udp_max_block_time = 5
178
179     def __init__(self):
180         core.openflow.addListeners(self)
181         core.openflow.addListenerByName("FlowStatsReceived", self.
            handle_flow_stats)
182         self.udp_packets = {}
183         self.udp_packet_count = {}
184         self.blocked = {}
185         self.unblockTried = set()
186
187         #Check stats every 5 seconds
188         Timer(5, self.requestStats, recurring = True)
189
190     def handle_flow_stats(self, event):
191         #Check udp packets sent based in flow table statistics
192         dpid = event.dpid
193         self.udp_packets[dpid] = {}
194         flow_packets = {}
195         for flow in event.stats:
196             ip = flow.match.nw_dst
197             if ip is not None and flow.match.nw_proto == pkt.ipv4.UDP_PROTOCOL:
198                 #Count packets sent for each udp flow table entry
199                 flow_packets[ip] = flow_packets.get(ip, 0) + flow.packet_count
200
201         for ip in flow_packets.keys():
202             #Packets sent in this period is calculated
203             #Total udp packets sent - last period udp packets sent
204             packets = flow_packets[ip] - self.udp_packet_count.get(dpid, {}).
                get(ip, 0)
205             if packets:
206                 self.udp_packets[dpid][ip] = packets
207                 if packets > self.udp_max_packet:
208                     self.blockUdp(ip)
209                 else:
210                     self.unblockUdp(ip)
211
212         for ip in self.blocked.keys():
213             #If ip is blocked and does not have new packets sent, try to
                unblock
214             if not ip in self.udp_packets[dpid].keys():
215                 self.unblockUdp(ip)
216
217         self.udp_packet_count[dpid] = flow_packets
218
219
220
221     def blockUdp(self, ip):
222         #Blocks Udp packets in all switches
223         blocks = self.blocked.get(ip, 0)
224         if not blocks:
225             msg = of.ofp_flow_mod()
226             msg.match.dl_type = pkt.ethernet.IP_TYPE
227             msg.match.nw_proto = pkt.ipv4.UDP_PROTOCOL

```

```

228         msg.match.nw_dst = ip
229         msg.priority = of.OFP_DEFAULT_PRIORITY + 1
230
231         for con in core.openflow.connections:
232             con.send(msg)
233
234
235
236         log.info("Blocking_UDP_flows_for_ip_dst_" + str(ip))
237         self.blocked[ip] = self.udp_max_block_time
238         self.unblockTried.add(ip)
239
240
241     def unblockUdp(self, ip):
242         if ip in self.unblockTried:
243             return
244
245         blocks = self.blocked.get(ip, 0)
246
247         if blocks:
248             self.unblockTried.add(ip)
249             log.info("Trying_to_unblock_UDP_flows_for_ip_dst_" + str(ip) + " :_"
250                     + str(blocks))
251
252             if blocks == 1:
253                 #It was blocked for self.udp_max_block_time periods
254                 msg = of.ofp_flow_mod()
255                 msg.match.dl_type = pkt.ethernet.IP_TYPE
256                 msg.match.nw_proto = pkt.ipv4.UDP_PROTOCOL
257                 msg.match.nw_dst = ip
258                 msg.command = of.OFPFC_DELETE
259
260                 log.info("Unblocking_UDP_flows_for_ip_dst_" + str(ip))
261
262                 for con in core.openflow.connections:
263                     con.send(msg)
264
265                 self.blocked[ip] = blocks - 1
266
267
268     def requestStats(self):
269         #Requests udp statistics from the switch that is connected to clients
270
271         self.unblockTried = set()
272
273         for connection in core.openflow.connections:
274             connection.send(of.ofp_stats_request(body=of.ofp_flow_stats_request(
275                 )))
276
277
278 class MyPortStats(EventMixin):
279
280     def __init__(self):
281         #Check port stats every 30 seconds
282         Timer(30, self.check_use_of_ports, recurring = True)
283
284     def check_use_of_ports(self):
285         for connection in core.openflow.connections:
286             switch_ports_used = 0

```



```

286         switch_ports_total = 0
287         dpid = connection.dpid
288         for port in connection.ports:
289             if port != of.OFPP_LOCAL:
290                 switch_ports_total += 1
291                 if ports_used[dpid][port] > 0:
292                     switch_ports_used += 1
293         if switch_ports_total == switch_ports_used:
294             text = "All_ports_used"
295         else:
296             text = "Ports_used: " + str(switch_ports_used) + "/" + str(
                switch_ports_total)
297         log.info("Switch " + dpidToStr(dpid) + ": " + text)
298
299
300
301 def launch ():
302     import pox.log.color
303     pox.log.color.launch()
304     import pox.log
305     pox.log.launch(format="[level%(name)-22sreset] " +
306                     "%(message)snormal")
307     import pox.log.level
308     import logging
309     pox.log.level.launch(packet=logging.WARN, host_tracker=logging.INFO)
310
311     from pox.core import core
312     import pox.openflow.discovery
313     pox.openflow.discovery.launch()
314
315     core.registerNew(MyController)
316
317     import pox.openflow.spanning_tree
318     pox.openflow.spanning_tree.launch()
319
320     import pox.host_tracker
321     pox.host_tracker.launch()
322
323     core.registerNew(MyFirewall)
324     core.registerNew(MyPortStats)

```

7. Referencias

1. <https://repositorio.unican.es/xmlui/bitstream/handle/10902/4484/Carlos%20Arteche%20Gonzalez.pdf;sequence=1>
2. <http://demo.spear.narmox.com/app/?apiurl=demo#!/mininet>
3. <https://datapath.io/resources/blog/how-we-run-bgp-on-top-of-openflow/>
4. <https://noxrepo.github.io/pox-doc/html/>