

75.43 Introducción a los sistemas distribuidos

Entrega Trabajo Práctico 3: Enlace

Integrantes:

Alumno	padron
Azcona, Gabriela Mariel	95363
Avigliano, Patricio Andres	98861
Blanco, Sebastian Ezequiel	98539

Fecha de Entrega: 23/10/2018

GitHub: <https://github.com/BlancoSebastianEzequiel/Datacenter>

Índice

1. Introducción teórica	1
2. Objetivo	3
3. Desarrollo	4
3.1. Preguntas	4
3.2. Maquina virtual	4
3.3. Topologia	4
3.4. Controlador	5
3.5. Firewall	7
3.6. Launch	7
4. Pruebas realizadas	8
4.1. Configuracion	8
4.2. Pingall	8
4.3. Balanceo de cargas con conexion TCP	9
4.4. Balanceo de cargas con ping	11
4.5. Denegacion de servicio	13
4.6. Pruebas automaticas	15
5. Conclusiones	16
6. Anexos (Código)	17
7. Referencias	29

1. Introducción teórica

- **SDN:**

Software Defined Networking es un paradigma que puede considerarse reciente en el cual los dispositivos intermediarios encargados de conmutar paquetes son configurados por una entidad controladora por medio de software. Decimos dispositivos intermediarios porque este nuevo paradigma permite una configuración tan flexible que se pierde la distinción entre switches, routers, NATs; ahora cada dispositivo se configura según las necesidades particulares de la red en la que habita.

- **OpenFlow:**

Es la herramienta que se utiliza para implementar esta nueva tecnología, mejor dicho es el protocolo por el cual se configuran los dispositivos intermediarios. La idea principal es reemplazar las tablas de ruteo de los routers y las tablas de direcciones Mac en los switches por tablas de flujo. Entonces un dispositivo OpenFlow decide qué hacer con los paquetes que le llegan en base a la tabla de flujo, por otro lado se configuran las políticas y el comportamiento que debe adoptar mediante el protocolo OpenFlow.

- **Control y Forwarding path:**

Un dispositivo de internet por definición debe funcionar con la mayor velocidad posible, por ello su funcionamiento está implementado por hardware y el costo de implementarlo exclusivamente por software en cuanto a velocidad sería muy elevado. Es por eso que los dispositivos OpenFlow se dividen en 2 planos: el plano de datos o forwarding (hardware) y el plano de control (software), este último es el que se comunica por medio de OpenFlow con la entidad controladora que indicará cómo deberá ser administrado el dispositivo. El plano de datos hará lo que sea necesario con cada paquete según la tabla de flujos mientras que el plano de control gestionará las decisiones a tomar sobre la construcción de la tabla, modificación de algún parámetro de la cabecera (por ejemplo al implementar Network Address Translation) y políticas de seguridad, entre otras funcionalidades.

- **Concepto de flujo:**

No existe una definición per se de lo que es un flujo pero lo entendemos como el conjunto de paquetes que esperamos que llegue de un mismo origen a un mismo destino (por destino y origen nos referimos a nivel enlace/red/-transporte) con similar latencia y por el mismo camino. Un ejemplo podría ser la respuesta de un http get, todos los paquetes de la respuesta provienen del mismo origen, van hacia el mismo destino y se espera que lleguen medianamente uno detrás del otro (suponiendo no haya pérdidas). Para lo que es un dispositivo OpenFlow un flujo se define como la 10-tupla formada por (PortIn, VLANID, srcEth, dstEth, typeEth, srcIP, dstIP, protoIP, srcport, dstport) y es sobre estos campos que se definen las entradas en la tabla

de flujos, luego podrán utilizarse los campos que sean necesarios según las políticas adoptadas por el plano de control.

- **IP blackholing:**

Es la decisión que se toma de descartar paquetes provenientes de una determinada dirección IP al detectar un ataque, los dispositivos OpenFlow permiten introducir políticas sobre lo que debe ser considerado como un ataque y en qué casos hacer IP blackholing de manera flexible y que se adapte a la sensibilidad de la red en la que está funcionando.

- **Firewall:**

Es un mecanismo de seguridad cuya función es proteger la red interna frente a amenazas de redes no confiables, para ello puede filtrar o redireccionar los paquetes que se consideran como no permitidos según ciertas reglas de seguridad. Por ejemplo si no queremos contestar paquetes ICMP podemos configurar para que todos los paquetes de ese protocolo sean descartados. Devuelta lo que permite OpenFlow es adaptar el firewall del dispositivo según las necesidades particulares de la red.

2. Objetivo

La idea del trabajo es familiarizarse con la tecnología de las SDNs y el protocolo OpenFlow, junto con las diversas problemáticas que permiten enfrentar; como objetivo secundario veremos una arquitectura de datacenters. Para ello simularemos mediante Mininet la estructura de un datacenter pequeño conectado bajo la topología Fat-Tree y configuraremos los switches mediante OpenFlow

3. Desarrollo

3.1. Preguntas

1. ¿Cuál es la diferencia entre un Switch y un router? ¿Qué tienen en común?
2. ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow?
3. ¿Se pueden reemplazar todos los routers de la Internet por Switches OpenFlow?

Piense en el escenario interASes para elaborar su respuesta

3.2. Maquina virtual

Necesitamos de la maquina virtual (VM) para poder ejecutar el controlador, el firewall y la topologia. Por lo tanto debemos usar distintas terminales de la misma para ello. Hay dos formas de hacer esto, una es usar la interfaz que provee la VM y abrir terminales desde alli. La otra es abrir terminales desde nuestra pc, y conectarnos via ssh a la maquina virtual. Parece que la primera opcion es mas comoda, pero en mi experiencia la VM se traba mucho mas. Para conectarse via ssh a la VM, primero clonamos nuestro repositorio y solo necesitamos ejecutar un script que hace el trabajo.

```
1 sh scripts/conect_to_VM.sh
2
```

Nos pedira la contraseña de nuestra pc, y luego la contraseña de la VM que es: *frenetic*.

Luego, si queremos tener nuestro repositorio en la VM, y ademas el controlador en la carpeta *pox/ext* del repositorio de pox en la VM, debemos ejecutar el siguiente comando:

```
1 sh scripts/resync.sh
2
```

3.3. Topologia

La topologia que se desarrollo se llama Fat tree. En la figura 5 vemos nuestra topologia por defecto que es un árbol de altura 3. La raiz tiene conectada 3 hosts que funcionan como clientes del datacenter, cada una de las raices a su vez, tendrán conectadas un host cada una, que funcionará como proveedor de contenido.

Los tres hosts clientes se comunican con los hosts del datacenter usando los protocolos ICMP, TCP y UDP.

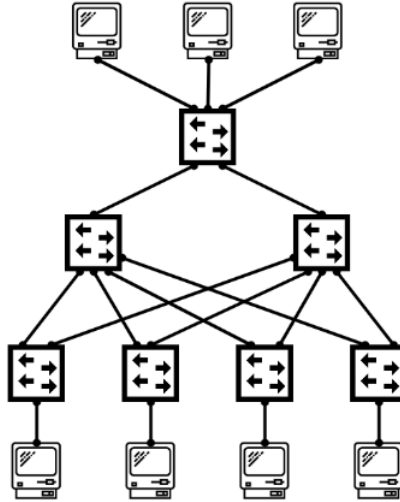


Figura 1: Ejemplo de como seria la topologia con un arbol de altura 3.

Para poder levantar la topologia sin pasarle los argumentos de configuracion, lo cual hace que por defecto tenga tres clientes y una altura de tres, abrimos una terminal que este en la raiz del sistema de archivos de la maquina virtual y podemos escribir lo siguiente:

```
1 sudo mm --custom ~/Datacenter/src/topology.py --topo mytopo
2 ↪ --mac --switch ovsk --controller remote
```

En caso de querer configurar la altura o cantidad de clientes, podemos escribir lo siguiente:

```
1 sudo mm --custom ~/Datacenter/src/topology.py --topo mytopo,
2 ↪ levels=4, clients=4 --mac --switch ovsk --controller remote
```

3.4. Controlador

El controlador se encarga de la logica del balanceo de cargas y tambien se encarga de llenar las tablas de los switch cuando estos les llega un mensaje que no saben responder. De esta manera nuestro controlador consta de *handlers* los cuales se ejecutan cuando un switch acude al controlador.

El controlador hace un estudio de la topologia para poder obtener un diccionario

de adyacencias, el cual sera util para poder calcular el camino minimo para el balanceo de cargas.

handle_LinkEvent:

Este handler es invocado cada vez que se detecta un link entre switches. El modulo *Discovery* es el que se encarga de esto. Se envias mensajes LLDP entre los switches y estos mismos responden, y de esta manera se puede aprender la topologia. En nuestro handler, cuando es invocado, llenamos un diccionario con la informacion pertinente.

host_tracker:

Es una clase de python que se encuantra en el repositorio de pox. En nuestro caso la utilizamos para poder descubrir el identificador del switch destino a partir de la direccion mac destino. Esto lo necesitamos porque las adyacencias se manejan con los identificadores de los switchs (dpid). Por eso, en el constructor del controlador se instancia el mismo, y por cada vez que se entra al handler que se encarga de llenar la tabla de los switches (*_handle_PacketIn*), se llama al handler del *host_tracker*, que se encarga de aprender los dpid. Luego, podremos obtener el identificador utilizando un metodo del *host_tracker* el cual accede a un diccionario de entradas de direcciones mac, y retorna el dpid. Ese metodo es *host_tracker.getMacEntry(addr)*

spanning tree:

Este modulo se encarga de eliminar los ciclos de la topologia. Su función es la de gestionar la presencia de bucles en topologías de red debido a la existencia de enlaces redundantes. El protocolo permite a los dispositivos de interconexión activar o desactivar automáticamente los enlaces de conexión, de forma que se garantice la eliminación de bucles.

handle_PacketIn:

Este handler se invoca frente a la llegada de un paquete, cuando un switch no sabe como responder frente a el, es decir, la tabla del switch no tiene un *match* para despacharlo por un puerto, por lo tanto, le pide ayuda al controlador. El algoritmo es bastante simple y consta de los siguientes pasos:

- Se llama al handler del *host_tracker* para que aprenda los dpid de cada mac.
- Si la topologia aun no fue aprendida, se retorna.
- Si el protocolo no es ni ICMP, o TCP o UDP se retorna
- Si el paquete es IPv6 se retorna
- Si el paquete ethernet no se encuentra, se retorna
- Si el paquete no es ni IP ni ARP, se retorna.
- Si la direccion mac destino, no se encuentra en nuestra tabla de matcheo de mac con dpid (generada a partir de la respuesta del *host_tracker* al invocar *host_tracker.getMacEntry(addr)*) se hace flooding y retorna

- Si el dpid origen coincide con el destino, no hacemos nada y guardamos esto en la tabla.
- si son distintos:
 - Se buscan todos los caminos minimos desde el dpid origen al destino en el diccionario de adyacencias.
 - Si no hay caminos, se retorna
 - si, existen caminos, se extrae de ellos los puertos de los cuales se sale del switch (seria el puerto de salida)
 - Se actualiza la tabla del switch
 - se envia el mensaje.

Tecnica ECMP - Balanceo de cargas:

Para resolverlo se creo una clase en python llamada *ECMPTable* la cual encapsula un diccionario que tiene como clave el identificador del switch correspondiente (*dpid*). Como valor de este identificador tiene otro diccionario en el cual su clave se una tupla de valores los cuales son, el identificador del switch destino (*dst_dpid*), la direccion mac origen (*src_addr*), la direccion mac destino (*dst_addr*), y un string que nos dice el protocolo (*protocol*: [ICMP, TCP, UDP]). Como valor a esta clave se encuentra el puerto de salida. De esta manera, frente a distintos caminos de igual peso, dependiendo el flujo, los puertos de salida son distintos.

3.5. Firewall

Se creo una clase llamada Firewall la cual, frente a la llegada de paquetes UDP, en caso de superar cierto maximo (en nuestro caso 100), se los bloquea. Luego de un tiempo, se los desbloquea. Para poder llevarlo a cabo, se creo un metodo llamado *request_for_switch_statistics*, el cual se llama cada un tiempo determinado mediante un timer. Este se encarga de pedirle las estadisticas a los switches, para poder calcular la frecuencia de paquetes UDP. Luego, tenemos otro handler llamado *_handle_flowstats_received* el cual se llama cada vez que los witches nos proveen sus estadisticas. El algoritmos se pregunta que los paquetes tengan el protocolo UDP, y almacena su cantidad en un diccioanrio. En caso de que la diferencia de cantidades entre la vez actual y la anterior acumulada sea mayor que nuestro maximo propuesto, se procede a bloquear el paquete y se setea un timer para este. En caso de no superar est maximo, se chequea si ya se esta bloqueado, y en ese caso se chequea el paso del tiempo con los timers.

3.6. Launch

Para poder levantar el controlador y el firewall, se creo un archivo llamado *launch.py* que configura el spanning tree, el discovery, el controlador y el firewall.

4. Pruebas realizadas

4.1. Configuración

Para poder correr el controlador y la topología debemos abrir la máquina virtual. Luego abrir dos terminales como se explico en la sección de máquina virtual en desarrollo. Para resumirlo en pasos, debemos:

1. Pararse en la raíz del repositorio
2. abrir una terminal y ejecutar: `sh scripts/resync.sh`
3. Luego, en la misma terminal ejecutar: `sh scripts/conect_to_VM.sh`. Nos pedirá nuestra contraseña y la contraseña de la VM que es: *frenetic*
4. Correr el controlador ejecutando: `pox/pox.py launch`
5. Repetir el paso 3 en otra terminal.
6. Levantar la topología ejecutando: `sh Datacenter/scripts/lift_topology.sh`

4.2. Pingall

Una vez que levantamos primero el controlador y luego la topología, los sitches tienen su tabla vacía, por lo que el primer pingall será consultado completamente al controlador para que resuelva las salidas. Por lo tanto el primer pingall siempre tardará un poco más, y en algunos casos, algún ping puede llegar a perderse por un timeout. A continuación mostramos un pingall apenas se levanta la topología y el controlador y otro pingall justo después así contrastaremos el tiempo que tarda cada uno:

```
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: 1.276723 secs
```

Figura 2: Pingall apenas se levanta el controlador y la topología

```

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: 0.075665 secs

```

Figura 3: Segundo Pingall

Podemos observar que no hubo perdidas, y ademas se ve claramente la diferencia de tiempo en que se resolvió los pings en cada caso donde en el primer caso se tardó unos 1,276723 *segundos* y en el segundo caso el tiempo disminuyó drásticamente a unos 0,075665 *segundos*.

4.3. Balanceo de cargas con conexión TCP

Para realizar esta prueba vamos a utilizar la herramienta *iperf* para hacer una conexión cliente-servidor entre dos hosts. Primero, vamos a ir a la terminal donde está corriendo mininet (donde levantamos la topología) y en el prompt de mininet escribiremos *xtermh1h4* para abrir dos terminales en cada host.

Luego vamos a abrir otra terminal (conectada a la VM como se explicó antes) y escribiremos: *sudo wireshark&*. Y abriremos la interfaz *s2 – eth1*. Y abriremos otro wireshark de la misma manera, pero en la interfaz *s3 – eth1*.

Luego en la terminal del host 4 levantaremos el servidor escribiendo lo siguiente: *iperf –s –p 80*. En la terminal del host 1 escribiremos lo siguiente para hacer la conexión: *iperf –c 10,0,0,4 –p 80*. y lo que haremos es ver ambas ventanas de wireshark y comprobar que los paquetes TCP solo pasan por algunas de las dos interfaces, pero no en ambas.

A continuación mostramos la captura de las terminales en cada host. y la comprobación del balanceo de cargas en wireshark:

```
"Node: h1"
root@ubuntu-1404:~# iperf -c 10.0.0.4 -p 80
-----
Client connecting to 10.0.0.4, TCP port 80
TCP window size: 85.3 KByte (default)
-----
[ 35] local 10.0.0.1 port 44832 connected with 10.0.0.4 port 80
[ ID] Interval      Transfer    Bandwidth
[ 35] 0.0-10.0 sec  43.7 GBytes  37.5 Gbits/sec
root@ubuntu-1404:~#
```

Figura 4: Conexión TCP cliente host 1

```
"Node: h4"
root@ubuntu-1404:~# iperf -s -p 80
-----
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
-----
[ 36] local 10.0.0.4 port 80 connected with 10.0.0.1 port 44832
[ ID] Interval      Transfer    Bandwidth
[ 36] 0.0-10.0 sec  43.7 GBytes  37.5 Gbits/sec
[]
```

Figura 5: Conexión TCP servidor host 4

Podemos comprobar que cumple con el balanceo de cargas ya que los paquetes solo viajan por el switch 2. Esto es así, porque ir por cualquiera de los switch es lo mismo respecto a llegar a destino y a cuanto pesa el mismo, es decir, tenemos un empate y vemos que el controlador decidió que de acuerdo a este flujo se elija solo el switch 2:

No.	Time	Source	Destination	Protocol	Length	Info
4	9.4248716...	10.0.0.1	10.0.0.4	TCP	74	37066 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=7327 TSecr=0 WS=512
6	9.5035791...	10.0.0.1	10.0.0.4	TCP	66	37066 → 80 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=7364 TSecr=7354
7	9.5036656...	10.0.0.1	10.0.0.4	TCP	90	37066 → 80 [PSH, ACK] Seq=1 Ack=1 Win=29696 Len=24 TSval=7364 TSecr=7354
8	9.5036712...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=25 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7354
9	9.5036749...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=1473 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7354 [TCP segment ...]
10	9.5036811...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=2921 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7354 [TCP segment ...]
11	9.5036849...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=4369 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7354 [TCP segment ...]
12	9.5036901...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=5817 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7354 [TCP segment ...]
13	9.5036938...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=7265 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7354 [TCP segment ...]
14	9.5036990...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=8713 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7354 [TCP segment ...]
15	9.5037027...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=10161 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7354 [TCP segment ...]
16	9.5037077...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=11609 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7354 [TCP segment ...]
27	9.5043192...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=13057 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7364 [TCP segment ...]
28	9.5043253...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=14505 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7364 [TCP segment ...]
29	9.5043309...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=15953 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7364 [TCP segment ...]
30	9.5043344...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=17401 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7364 [TCP segment ...]
31	9.5043393...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=18849 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7364 [TCP segment ...]
32	9.5043429...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=20297 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7364 [TCP segment ...]
33	9.5043479...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=21745 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7364 [TCP segment ...]
34	9.5043516...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=23193 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7364 [TCP segment ...]
35	9.5043569...	10.0.0.1	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=24641 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7364 [TCP segment ...]

Figura 6: Captura de wireshark monitoreando la interfaz eth1 del switch s2

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

Figura 7: Captura de wireshark monitoreando la interfaz eth1 del switch s3

4.4. Balanceo de cargas con ping

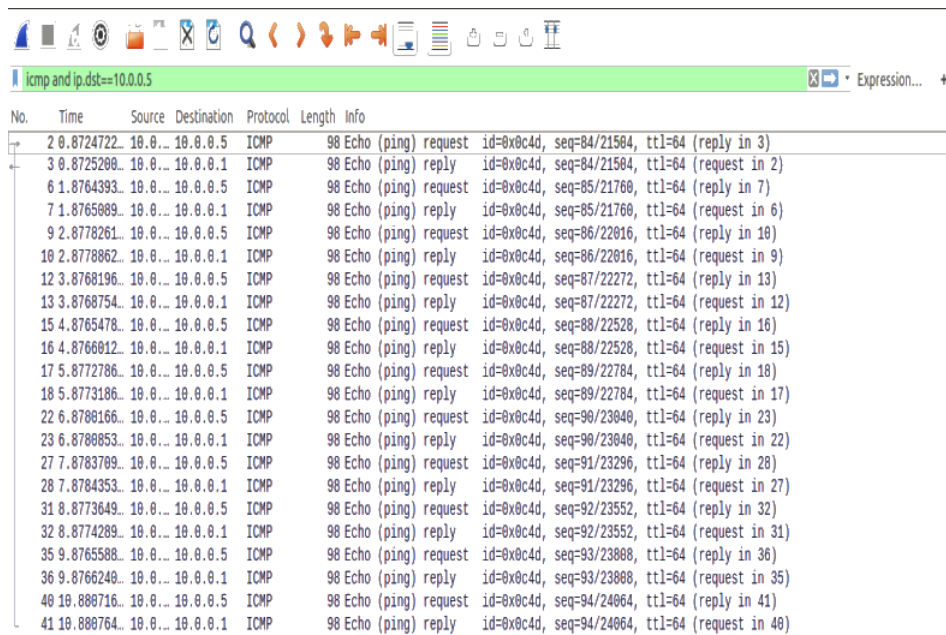
Para realizar esta prueba vamos a usar miniet haciendo un ping entre h1 y h5, y otro ping entre h2 y h5. De esta manera podremos ver que lo que debería pasar es que un flujo debería elegir el switch 2 o 3 y el otro flujo también sin que ambos elijan el mismo flujo. Los paquetes que se envían contienen un protocolo ICMP, por lo tanto haremos un monitoreo en las interfaces s3-eth1 y s2-eth1 para verificar que cuando hacemos el ping entre h1 y h5 una de ellas está vacía y la otra recibe los paquetes, pero al hacer el ping entre h2 y h5 debemos poder ver lo mismo pero en los switches invertidos.

Primero mostramos el monitoreo en wireshark cuando hacemos el ping entre h1 y h5:



No.	Time	Source	Destination	Protocol	Length	Info
2	0.8724722	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=84/21504, ttl=64 (reply in 3)

Figura 8: Captura de wireshark monitoreando la interfaz eth1 del switch s2



No.	Time	Source	Destination	Protocol	Length	Info
2	0.8724722	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=84/21504, ttl=64 (reply in 3)
3	0.8725200	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=84/21504, ttl=64 (request in 2)
6	1.8764393	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=85/21760, ttl=64 (reply in 7)
7	1.8765089	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=85/21760, ttl=64 (request in 6)
9	2.8778261	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=86/22016, ttl=64 (reply in 10)
10	2.8778862	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=86/22016, ttl=64 (request in 9)
12	3.8768196	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=87/22272, ttl=64 (reply in 13)
13	3.8768754	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=87/22272, ttl=64 (request in 12)
15	4.8765478	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=88/22528, ttl=64 (reply in 16)
16	4.8766012	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=88/22528, ttl=64 (request in 15)
17	5.8772786	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=89/22784, ttl=64 (reply in 18)
18	5.8773186	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=89/22784, ttl=64 (request in 17)
22	6.8780166	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=90/23040, ttl=64 (reply in 23)
23	6.8780853	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=90/23040, ttl=64 (request in 22)
27	7.8783709	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=91/23296, ttl=64 (reply in 28)
28	7.8784353	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=91/23296, ttl=64 (request in 27)
31	8.8773649	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=92/23552, ttl=64 (reply in 32)
32	8.8774289	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=92/23552, ttl=64 (request in 31)
35	9.8765588	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=93/23808, ttl=64 (reply in 36)
36	9.8766240	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=93/23808, ttl=64 (request in 35)
40	10.880716	10.0.0.5	10.0.0.1	ICMP	98	Echo (ping) request id=0x0c4d, seq=94/24064, ttl=64 (reply in 41)
41	10.880764	10.0.0.1	10.0.0.5	ICMP	98	Echo (ping) reply id=0x0c4d, seq=94/24064, ttl=64 (request in 40)

Figura 9: Captura de wireshark monitoreando la interfaz eth1 del switch s3

Luego mostramos el monitoreo en wireshark cuando hacemos el ping entre h2 y h5:

No.	Time	Source	Destination	Protocol	Length	Info
→ 10	0.2014358...	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0e05, seq=12/3072, ttl=64 (reply in 11)
28	1.2004480...	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0e05, seq=13/3328, ttl=64 (reply in 29)
49	2.2001658...	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0e05, seq=14/3584, ttl=64 (reply in 41)
60	3.2045812...	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0e05, seq=15/3840, ttl=64 (reply in 61)
66	4.2064456...	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0e05, seq=16/4096, ttl=64 (reply in 67)
73	5.2054203...	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0e05, seq=17/4352, ttl=64 (reply in 74)
78	6.2044588...	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0e05, seq=18/4608, ttl=64 (reply in 79)
81	7.2047255...	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0e05, seq=19/4864, ttl=64 (reply in 82)
86	8.2054091...	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0e05, seq=20/5120, ttl=64 (reply in 87)

Figura 10: Captura de wireshark monitoreando la interfaz eth1 del switch s2

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

Figura 11: Captura de wireshark monitoreando la interfaz eth1 del switch s3

Como se puede observar, el ping entre h1 y h5 solo pasa por el switch 3 y el ping entre h2 y h5 solo pasa por el switch 2.

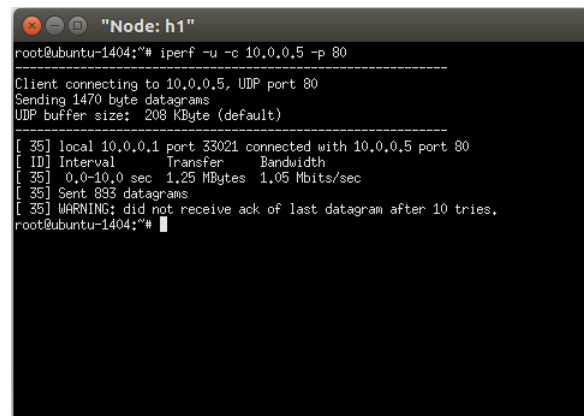
4.5. Denegacion de servicio

Para realizar esta prueba vamos a utilizar la herramienta *iperf* para hacer una conexcion cliente-servidor entre dos hosts. Primero, vamos a ir a la terminal donde esta corriendo mininet (donde levatamos la topologia) y en el prompt de mininet escribiremos *xtermh1h5* para abrir dos terminales en cada host.

Luego vamos a abrir otra terminal (conectada a la VM como se exlico antes) y escribiremos: *sudo wireshark &*. Y abriremos la interfaz *s5 - eth1*. Luego en la terminal del host 5 levantaremos el servidor escribiendo lo siguiente: *iperf -u -s -p 80*. En la terminal del host 1 escribiremos lo siguiente para hacer la conexcion: *iperf -u -c 10.0.0.4 -p 80*. y lo que haremos es ver ambas ventanas de wireshark y comprobar que los paquetes TCP solo pasan por algunas de las dos interfaz, pero no en ambas.

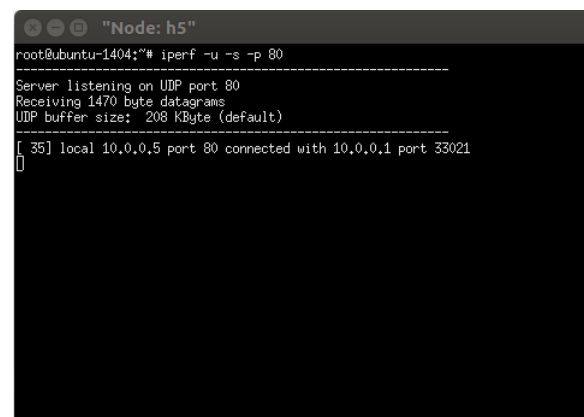
A continuacion mostramos la captura de las terminales en cada host y la captura de wireshark y debemos comprobar que el host 1 devuelve un warning el cual dice que no se pudieron enviar todos los datagramas y por lo tanto en wireshar debemos

poder recibir una cantidad menor a la que se queria enviar.



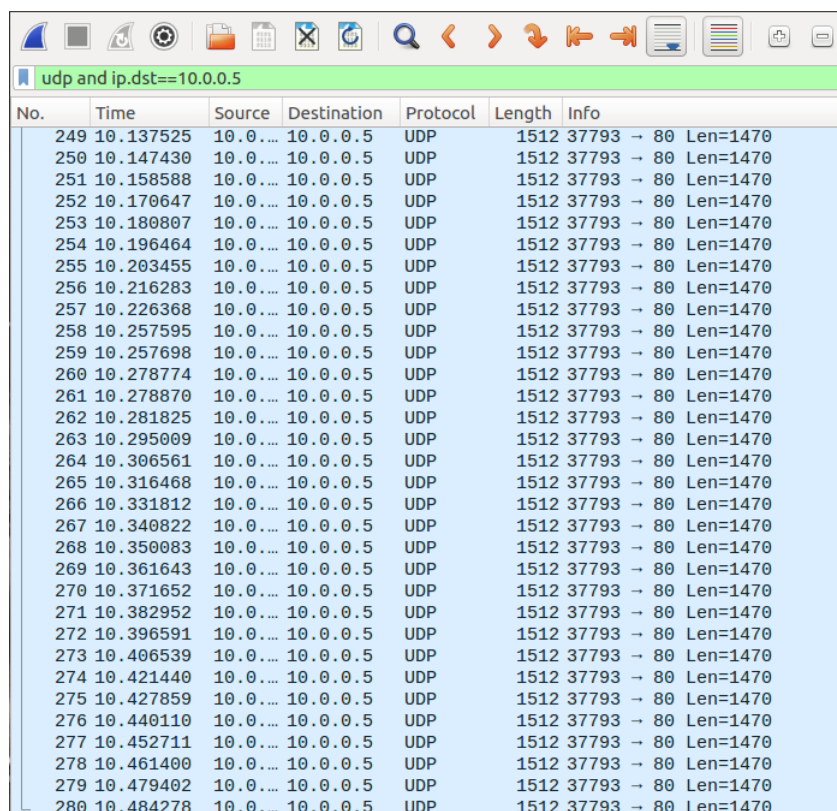
```
root@ubuntu-1404:~# iperf -u -c 10.0.0.5 -p 80
-----
Client connecting to 10.0.0.5, UDP port 80
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 35] local 10.0.0.1 port 33021 connected with 10.0.0.5 port 80
[ ID] Interval      Transfer    Bandwidth
[ 35] 0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 35] Sent 893 datagrams
[ 35] WARNING: did not receive ack of last datagram after 10 tries.
root@ubuntu-1404:~#
```

Figura 12: Conexion UDP servidor host 1



```
root@ubuntu-1404:~# iperf -u -s -p 80
-----
Server listening on UDP port 80
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 35] local 10.0.0.5 port 80 connected with 10.0.0.1 port 33021
[]
```

Figura 13: Conexion UDP servidor host 5



No.	Time	Source	Destination	Protocol	Length	Info
249	10.137525	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
250	10.147430	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
251	10.158588	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
252	10.170647	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
253	10.180807	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
254	10.196464	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
255	10.203455	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
256	10.216283	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
257	10.226368	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
258	10.257595	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
259	10.257698	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
260	10.278774	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
261	10.278870	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
262	10.281825	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
263	10.295009	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
264	10.306561	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
265	10.316468	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
266	10.331812	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
267	10.340822	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
268	10.350083	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
269	10.361643	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
270	10.371652	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
271	10.382952	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
272	10.396591	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
273	10.406539	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
274	10.421440	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
275	10.427859	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
276	10.440110	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
277	10.452711	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
278	10.461400	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
279	10.479402	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470
280	10.484278	10.0.0.1	10.0.0.5	UDP	1512	37793 → 80 Len=1470

Figura 14: Captura de wireshark monitoreando la interfaz eth1 del switch s5

Como se puede observar, vemos que recibimos menos datagramas porque como dice en la terminal del host 1, se quisieron enviar 893 y recibimos 280.

4.6. Pruebas automaticas

Esta seccion se realizo a medias ya que en principio se realizaron test unitarios usando *pytest* donde mediante wireshark, se lee la captura que se hizo con wireshark y se comprueba lo que se mostro anteriormente. Pero queda a modo de tareas a realizar poder ejecutar desde un solo script codigo que levante todas las terminales que hacen falta para poder hacer la conexcion y mediante *tcpdump*, generar la captura deseada. De esta manera, nuestros test no cambiarian, y siempre leerian un archivo **.pcap*, donde mediante *tshark* se lo transforma en un txt para poder leerlo usando una herramienta llamada *pandas*. En un futuro, como segundo release, se podria llevar a cabo dichos test y de esa manera de automatiza todo el proyecto.

5. Conclusiones

6. Anexos (Código)

```
1 from mininet.topo import Topo
2
3
4 class Topology(Topo):
5     def __init__(self, number_of_levels=3, number_of_clients=3):
6         """
7         :type number_of_levels: int
8         """
9         Topo.__init__(self)
10        self.level_links = {}
11        self.sw_num = 1
12        self.h_num = 1
13        self.number_of_levels = number_of_levels
14        self.number_of_clients = number_of_clients
15        self.add_clients()
16        self.add_switches_and_links()
17        self.add_content_providers()
18
19    def add_clients(self):
20        self.level_links[0] = []
21        for i in range(0, self.number_of_clients):
22            self.level_links[0].append(self.addHost('h%s' %
↪ self.h_num))
23            self.h_num += 1
24
25    def add_switches_and_links(self):
26        for level in range(0, self.number_of_levels):
27            next_level = level + 1
28            number_of_switches_in_level = 2 ** level
29            self.level_links[next_level] = []
30            for i in range(0, number_of_switches_in_level):
31                sw = self.addSwitch('s%s' % self.sw_num)
32                self.level_links[next_level].append(sw)
33                self.sw_num += 1
34                for device in self.level_links[level]:
35                    self.addLink(sw, device)
36
37    def add_content_providers(self):
38        for sw in self.level_links[self.number_of_levels]:
39            self.addLink(sw, self.addHost('h%s' % self.h_num))
40            self.h_num += 1
41
42
43 topos = {
44     'mytopo': (lambda levels=3, clients=3: Topology(levels, clients))
45 }
```

Listing 1: Topology

```
1 from pox.core import core
```

```

2 import pox.openflow.libopenflow_01 as of
3 from pox.lib.packet.ethernet import ethernet, ETHER_BROADCAST
4 from pox.lib.util import dpidToStr
5 from pox.lib.packet.packet_utils import _ethtype_to_str
6 from pox.host_tracker.host_tracker import host_tracker
7 import pox.lib.packet as pkt
8 from pox.lib.revent import *
9 from ecmp_table import ECMPTable
10
11 log = core.getLogger()
12
13
14 class Controller(object):
15
16     def __init__(self):
17         core.openflow.addListeners(self)
18
19     def startup():
20         core.openflow.addListeners(self, priority=0)
21         core.openflow_discovery.addListeners(self)
22
23     core.call_when_ready(startup, ('openflow',
    ↪ 'openflow_discovery'))
24
25     self.event = None
26     self.dpid = None
27     self.in_port = None
28     self.packet = None
29     self.dst_dpid = None
30     self.out_port = None
31     self.table = ECMPTable()
32     self.eth_packet = None
33     self.ip_packet = None
34     self.arp_packet = None
35     self.icmp_packet = None
36     self.tcp_packet = None
37     self.udp_packet = None
38     self.net_packet = None
39     self.protocol_packet = None
40     self.protocol = None
41     self.arp_table = {}
42     self.is_ip = True
43     self.adjacency = {}
44     self.host_tracker = host_tracker()
45     log.info("controller ready")
46
47     def add_adjacency(self, dpid1, port1, dpid2, port2):
48         if dpid1 not in self.adjacency:
49             self.adjacency[dpid1] = {}
50         self.adjacency[dpid1][port1] = {
51             "dpid": dpid2,
52             "port": port2

```

```

53         }
54
55     def remove_adjacency(self, dpid, port):
56         if dpid not in self.adjacency:
57             return
58         if port not in self.adjacency[dpid][port]:
59             return
60         del self.adjacency[dpid][port]
61
62     def _handle_LinkEvent(self, event):
63
64         ↪ log.info("_____")
65         link = event.link
66         if event.added:
67             ↪ self.add_adjacency(link.dpid1, link.port1, link.dpid2,
68             ↪ link.port2)
69             self.add_adjacency(link.dpid2, link.port2, link.dpid1,
70             ↪ link.port1)
71         elif event.removed:
72             self.remove_adjacency(link.dpid1, link.port1)
73             self.remove_adjacency(link.dpid2, link.port2)
74             log.info('link added is %s' % event.added)
75             log.info('link removed is %s' % event.removed)
76             log.info('switch1 %d' % link.dpid1)
77             log.info('port1 %d' % link.port1)
78             log.info('switch2 %d' % link.dpid2)
79             log.info('port2 %d' % link.port2)
80
81         ↪ log.info("_____")
82
83     def _handle_ConnectionUp(self, event):
84         log.debug("Connection %s" % (event.connection,))
85
86         msg = of.ofp_flow_mod()
87         msg.match.dl_dst = ETHER_BROADCAST
88         msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
89         event.connection.send(msg)
90
91         msg = of.ofp_flow_mod()
92         msg.match.dl_type = pkt.ethernet.IPV6_TYPE
93         event.connection.send(msg)
94
95         msg = of.ofp_flow_mod()
96         msg.match.dl_type = pkt.ethernet.ARP_TYPE
97         msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
98         event.connection.send(msg)
99
100     @staticmethod
101     def print_msg(msg):
102         print "+++++"
103         print msg
104         print "+++++"

```

```

101
102 def fill_arp_table(self):
103     entry = self.host_tracker.getMacEntry(self.addr_dst)
104     if entry is None:
105         log.info("HOST TRACKER COULD NOT FIND ENTRY DST")
106         return
107     self.arp_table[self.addr_dst] = {
108         "dpid": entry.dpid,
109         "port": entry.port
110     }
111
112 def print_adjacents(self):
113     msg = ""
114     for dpid in self.adjacency:
115         msg += "dpid: %s: [" % dpid
116         for port in self.adjacency[dpid]:
117             msg += "%s, " % self.adjacency[dpid][port]["dpid"]
118         msg += "]"
119         log.info(msg)
120         msg = ""
121
122 def has_discovered_the_entire_topology(self):
123     if len(self.adjacency.keys()) != 7:
124         return False
125     for dpid in self.adjacency:
126         size = len(self.adjacency[dpid])
127         if dpid in [4, 5, 6, 7] and size < 2:
128             return False
129         if dpid in [2, 3] and size < 4:
130             return False
131         if dpid == 1 and size < 1:
132             return False
133     return True
134
135 def _handle_PacketIn(self, event):
136     self.host_tracker._handle_PacketIn(event)
137     if not self.has_discovered_the_entire_topology():
138         log.info("Please wait... learning the topology")
139         return
140     self.event = event
141     self.dpid = event.connection.dpid
142
143     ↪ log.info("_____")
144     self.print_adjacents()
145     log.info("SWITCH %s" % self.dpid)
146     self.in_port = event.port
147     self.packet = event.parsed
148     log.info("ports: %s" % event.connection.ports)
149     log.info("ports: %s" % event.connection.ports)
150     log.info("in port: %s" % self.in_port)
151     if not self.packet.parsed:
152         log.warning("%s %s ignoring unparsed packet" %

```

```

152             (self.dpid, self.in_port))
153         return
154         log.info("HOST SRC %s" % self.packet.src)
155         log.info("HOST DST: %s" % self.packet.dst)
156         self.eth_packet = self.packet.find(pkt.ethernet)
157         self.addr_dst = self.packet.dst
158         self.fill_arp_table()
159         self.ip_packet = self.packet.find(pkt.ipv4)
160         self.arp_packet = self.packet.find(pkt.arp)
161         self.icmp_packet = self.packet.find(pkt.icmp)
162         self.tcp_packet = self.packet.find(pkt.tcp)
163         self.udp_packet = self.packet.find(pkt.udp)
164
165         if not self.validate_protocols():
166
167             return
168         if not self.validate_net_packets():
169             return
170
171         if self.addr_dst not in self.arp_table:
172             log.warning("Could not find dst")
173             return self.flood()
174
175         entry = self.arp_table[self.addr_dst]
176         self.dst_dpid = entry["dpid"]
177         if self.dpid == self.dst_dpid:
178             log.info("Current switch is destination")
179             self.out_port = entry["port"]
180         else:
181             if self.packet.dst.is_multicast:
182                 self.print_msg("MULTICAST")
183                 return self.flood()
184             log.info("Finding minimum paths from %s to %s"
185                     % (self.dpid, self.dst_dpid))
186             minimun_paths = self.get_minimun_paths()
187             log.info("finding out port")
188             self.out_port = self.get_out_port(minimun_paths)
189             if self.out_port is None:
190                 log.info("Could not find out port")
191                 return
192             log.info("out port: %s" % self.out_port)
193             log.info("Updating flow table")
194             self.update_table()
195             log.info("Sending packet")
196             self.send_packet()
197
198     def flood(self):
199         log.info("FLOODING PACKET")
200         msg = of.ofp_packet_out()
201         msg.buffer_id = self.event.ofp.buffer_id
202         msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
203         msg.data = self.event.ofp

```

```

204         msg.in_port = self.in_port
205         self.event.connection.send(msg)
206
207     def validate_protocols(self):
208         if self.udp_packet is not None:
209             log.info("UDP packet!")
210             self.protocol = "UDP"
211             self.protocol_packet = self.udp_packet
212             return True
213         elif self.tcp_packet is not None:
214             log.info("TCP packet!")
215             self.protocol = "TCP"
216             self.protocol_packet = self.tcp_packet
217             return True
218         elif self.icmp_packet is not None:
219             log.info("ICMP packet!")
220             self.protocol = "ICMP"
221             self.protocol_packet = self.icmp_packet
222             return True
223         else:
224             log.warning("icmp, tcp and udp packets are None!")
225             return False
226
227     def validate_net_packets(self):
228         if _ethtype_to_str[self.packet.type] == "IPV6":
229             log.warning("DROP IPV6 packet")
230             return False
231         if self.eth_packet is None:
232             log.warning("ETHERNET packet is None!")
233             return False
234         if self.ip_packet is not None:
235             log.info("IP packet!")
236             self.is_ip = True
237             self.net_packet = self.ip_packet
238         elif self.arp_packet is not None:
239             log.info("ARP packet!")
240             self.is_ip = False
241             self.net_packet = self.arp_packet
242         else:
243             log.warning("ARP and TCP packets are None!")
244             return False
245         return True
246
247     def match_protocol_packets(self, msg):
248         if self.is_ip:
249             msg.match.nw_src = self.net_packet.srcip
250             msg.match.nw_dst = self.net_packet.dstip
251             msg.match.nw_proto = self.net_packet.protocol
252             return msg
253         msg.match.nw_src = self.net_packet.protosrc
254         msg.match.nw_dst = self.net_packet.protodst
255         msg.match.nw_proto = self.net_packet.prototype

```



```

256         return msg
257
258     def match_packet(self, msg):
259         if not self.is_ip:
260             return msg
261         msg.match.nw_src = self.ip_packet.srcip
262         msg.match.nw_dst = self.ip_packet.dstip
263         msg.match.nw_proto = self.ip_packet.protocol
264         return msg
265
266     def update_table(self):
267         msg = of.ofp_flow_mod()
268         msg.match.dl_type = self.eth_packet.type
269         msg = self.match_packet(msg)
270         msg.buffer_id = self.event.ofp.buffer_id
271         if self.protocol != "ICMP":
272             msg.match.tp_src = self.protocol_packet.srcport
273             msg.match.tp_dst = self.protocol_packet.dstport
274         msg.actions.append(of.ofp_action_output(port=self.out_port))
275         self.event.connection.send(msg)
276         self.balance_of_charges()
277
278     def get_minimun_paths(self):
279         adjacents = self.get_adjacents(self.dpid)
280         if not adjacents:
281             log.warning("NO ADJACENTS FOUND")
282             return []
283         paths = [[neighbour] for neighbour in adjacents]
284         while not self.has_found_a_path(paths, self.dst_dpid):
285             last_paths = paths[:]
286             for path in last_paths:
287                 adjacents = self.get_adjacents(path[-1]["dpid"])
288                 for an_adjacent in adjacents:
289                     if an_adjacent["dpid"] != self.dpid:
290                         if not self.node_belongs_path(an_adjacent,
291 ↪ path):
292                             paths.append(path + [an_adjacent])
293             return self.filter_paths(paths, self.dst_dpid)
294
295     def node_belongs_path(self, node, path):
296         dpid = node["dpid"]
297         for a_node in path:
298             if a_node["dpid"] == dpid:
299                 return True
300         return False
301
302     def get_out_port(self, paths_to_dst):
303         if len(paths_to_dst) == 0:
304             return None
305         ports = self.get_all_ports(paths_to_dst)
306         data = (
307             ports,

```

```

307         self.dpid,
308         self.dst_dpid,
309         self.protocol,
310         self.packet.src,
311         self.packet.dst
312     )
313     return self.table.get_port_applying_ecmp(data)
314
315     def balance_of_charges(self):
316         log.info("saving (%s, %s, %s): %s" %
317             (self.dpid, self.dst_dpid, self.protocol,
↪ self.out_port))
318         data = (
319             self.dpid,
320             self.dst_dpid,
321             self.protocol,
322             self.packet.src,
323             self.packet.dst,
324             self.out_port
325         )
326         self.table.save_port(data)
327
328     @staticmethod
329     def get_all_ports(paths_to_dst):
330         return [a_path[0]["port"] for a_path in paths_to_dst]
331
332     @staticmethod
333     def filter_paths(paths, dpid):
334         paths_to_dst = []
335         for path in paths:
336             if path[-1]["dpid"] != dpid:
337                 continue
338             paths_to_dst.append(path)
339         return paths_to_dst
340
341     def has_found_a_path(self, paths, dpid):
342         for path in paths:
343             if path[-1]["dpid"] == dpid:
344                 log.info("FOUND A PATH!")
345                 return True
346         return False
347
348     def get_adjacents(self, dpid):
349         adjacents = []
350         if dpid not in self.adjacency:
351             return adjacents
352         for port in self.adjacency[dpid]:
353             adjacents.append({
354                 "dpid": self.adjacency[dpid][port]["dpid"],
355                 "port": port
356             })
357         return adjacents

```

```

358
359     def filter_repeated(self, adjacents):
360         filtered = []
361         belongs = False
362         for an_adjacent in adjacents:
363             for final_adjacent in filtered:
364                 dpid_1 = final_adjacent["dpid"]
365                 dpid_2 = an_adjacent["dpid"]
366                 port_1 = final_adjacent["port"]
367                 port_2 = an_adjacent["port"]
368                 if dpid_1 == dpid_2 and port_1 == port_2:
369                     belongs = True
370                     break
371             if not belongs:
372                 filtered.append(an_adjacent)
373                 belongs = False
374         return filtered
375
376     def send_packet(self):
377         msg = of.ofp_packet_out()
378         msg.actions.append(of.ofp_action_output(port=self.out_port))
379         msg.data = self.event.ofp
380         msg.buffer_id = self.event.ofp.buffer_id
381         msg.in_port = self.in_port
382         self.event.connection.send(msg)

```

Listing 2: Controller

```

1 import pox.lib.packet as pkt
2 from pox.core import core
3 import pox.openflow.libopenflow_01 as of
4 from time import time
5 from pox.lib.recoo import Timer
6 from pox.lib.revent import *
7
8 UDP_PROTOCOL = pkt.ipv4.UDP_PROTOCOL
9 IP_TYPE = pkt.ethernet.IP_TYPE
10 log = core.getLogger()
11
12
13 class Firewall(EventMixin):
14     def __init__(self):
15         self.MAX_UDP_PACKETS = 100
16         self.MAX_UDP_TIME = 100
17         self.last_udp_flow_packets = {}
18         self.total_udp_flow_packets = {}
19         self.blocked_udp_packets = {}
20         self.dst_ip = None
21         self.dpid = None
22         core.openflow.addListenerByName(
23             "FlowStatsReceived",
24             self._handle_flowstats_received

```

```

25     )
26     Timer(5, self.request_for_switch_statistics, recurring=True)
27     log.info("firewall ready")
28
29     @staticmethod
30     def print_msg(msg):
31         print "++++++++++++++++++++++++++++++++++++"
32         print msg
33         print "++++++++++++++++++++++++++++++++++++"
34
35     def request_for_switch_statistics(self):
36         for connection in core.openflow.connections:
37             body = of.ofp_flow_stats_request()
38             connection.send(of.ofp_stats_request(body=body))
39
40     def _handle_flowstats_received(self, event):
41         log.info("handle denial of service")
42         self.dpid = event.connection.dpid
43         self.total_udp_flow_packets = {}
44         for flow in event.stats:
45             self.dst_ip = flow.match.nw_dst
46             if self.dst_ip is None:
47                 log.info("DST IP IS NONE. COULD NOT HANDLE DoS")
48                 continue
49             if not self.get_udp_flow(flow):
50                 continue
51             self.evaluate_blocking()
52             current = self.total_udp_flow_packets[self.dst_ip]
53             self.last_udp_flow_packets[self.dpid] = {}
54             self.last_udp_flow_packets[self.dpid][self.dst_ip] =
↪ current
55
56     def get_udp_flow(self, flow):
57         if self.dst_ip is None or flow.match.nw_proto !=
↪ UDP_PROTOCOL:
58             return False
59         if self.dst_ip not in self.total_udp_flow_packets:
60             self.total_udp_flow_packets[self.dst_ip] =
↪ flow.packet_count
61         else:
62             self.total_udp_flow_packets[self.dst_ip] +=
↪ flow.packet_count
63         return True
64
65     def get_last_udp_flow_packets(self, dst_ip):
66         if self.dpid not in self.last_udp_flow_packets:
67             self.last_udp_flow_packets[self.dpid] = {}
68             self.last_udp_flow_packets[self.dpid][dst_ip] = 0
69         return 0
70         elif dst_ip not in self.last_udp_flow_packets[self.dpid]:
71             self.last_udp_flow_packets[self.dpid][dst_ip] = 0
72         return self.last_udp_flow_packets[self.dpid][dst_ip]

```

```

73
74     def evaluate_blocking(self):
75         for dst_ip in self.total_udp_flow_packets:
76             current = self.total_udp_flow_packets[dst_ip]
77             last = self.last_udp_flow_packets.get(self.dpid,
↪             {}).get(dst_ip, 0)
78             if (current - last) > self.MAX_UDP_PACKETS:
79                 self.block_udp_packet(dst_ip)
80             else:
81                 self.unblock_udp_packet(dst_ip)
82
83     def block_udp_packet(self, dst_ip):
84         log.info("BLOCKING UDP PACKET IN %s" % dst_ip)
85         if dst_ip not in self.blocked_udp_packets:
86             log.info("Blocking ip: %s" % dst_ip)
87             msg = of.ofp_flow_mod()
88             msg.match.nw_proto = UDP_PROTOCOL
89             msg.match.dl_type = IP_TYPE
90             msg.priority = of.OFP_DEFAULT_PRIORITY + 1
91             msg.match.nw_dst = dst_ip
92             self.send_message_to_all(msg)
93             self.blocked_udp_packets[dst_ip] = time()
94
95     def unblock_udp_packet(self, dst_ip):
96         if dst_ip not in self.blocked_udp_packets:
97             return
98         time_passed = time() - self.blocked_udp_packets[dst_ip]
99         if time_passed < self.MAX_UDP_TIME:
100             return
101         del self.blocked_udp_packets[dst_ip]
102         log.info("UNBLOCKING UDP PACKET IN %s" % dst_ip)
103         log.info("unblocking ip: %s" % dst_ip)
104         msg = of.ofp_flow_mod()
105         msg.match.nw_proto = UDP_PROTOCOL
106         msg.match.dl_type = IP_TYPE
107         msg.command = of.OFPFC_DELETE
108         msg.match.nw_dst = dst_ip
109         self.send_message_to_all(msg)
110
111     @staticmethod
112     def send_message_to_all(msg):
113         for a_connection in core.openflow.connections:
114             a_connection.send(msg)

```

Listing 3: Firewall

```

1 def launch():
2     import pox.log.color
3     pox.log.color.launch()
4     import pox.log
5     pox.log.launch(format="[ @@@bold@@@level %(name) -22s@@@reset] " +
6                      "@@@bold %(message)s@@@normal")

```

```

7  from pox.core import core
8  import pox.openflow.discovery
9  pox.openflow.discovery.launch()
10 core.getLogger("openflow.spanning_tree").setLevel("INFO")
11 import pox.openflow.spanning_tree
12 pox.openflow.spanning_tree.launch()
13 from controller import Controller
14 core.registerNew(Controller)
15 from firewall import Firewall
16 core.registerNew(Firewall)

```

Listing 4: file to launch controller and firewall

```

1  import random
2
3
4  class ECMPTable(object):
5
6      def __init__(self):
7          self.table = {}
8
9      def get_port_applying_ecmp(self, data):
10         (ports, dpid, dst_dpid, protocol, src_addr, dst_addr) = data
11         key = (dst_dpid, protocol, src_addr, dst_addr)
12         if dpid not in self.table:
13             self.table[dpid] = {}
14             random.shuffle(ports)
15             self.table[dpid][key] = ports[0]
16             return self.table[dpid][key]
17         elif key not in self.table[dpid]:
18             is_used = False
19             for a_port in ports:
20                 for a_key in self.table[dpid]:
21                     if a_port == self.table[dpid][a_key]:
22                         is_used = True
23                 if not is_used:
24                     self.table[dpid][key] = a_port
25                     return a_port
26             random.shuffle(ports)
27             self.table[dpid][key] = ports[0]
28             return self.table[dpid][key]
29         return self.table[dpid][key]
30
31     def save_port(self, data):
32         (dpid, dst_dpid, protocol, src_addr, dst_addr, port) = data
33         key = (dst_dpid, protocol, src_addr, dst_addr)
34         self.table[dpid] = {}
35         self.table[dpid][key] = port

```

Listing 5: ecmp table

7. Referencias

1. CREATE A LEARNING SWITCH
<https://github.com/mininet/openflow-tutorial/wiki/Create-a-Learning-Switch>
2. OPEN DATAPATH
<https://www.opennetworking.org/technical-communities/areas/specification/open-datapath/>