

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: 04/12/2018

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	5
3.3. Topologia	5
3.4. Controlador	7
3.5. Firewall	9
3.6. Launch	9
4. Pruebas realizadas	10
4.1. Configuracion	10
4.2. Pingall	10
4.3. Balanceo de cargas con conexion TCP	11
4.4. Balanceo de cargas con ping	13
4.5. Denegacion de servicio	15
4.6. Pruebas automaticas	17
5. Conclusiones	18
6. Anexos (Código)	19
6.1. topology.py	19
6.2. contoller.py	21
6.3. firewall.py	29
6.4. launch.py	32
6.5. ecmp_table.py	33
7. Referencias	34

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?

Diferencias:

- Direccionamiento dentro de la misma red
- Confección de la tabla de ruteo
 - Los switches aprenden a partir de la direccion origen
 - Los routers aprenden a partir de la direccion destino
- Metodología de ruteo (broadcast en SW)
- Configuración (plug-n-play vs DFGW)
- El router es el dispositivo que se encarga de reenviar los paquetes entre distintas redes
- El router es mas “inteligente” que el switch, ya que además de cumplir con la misma función, tiene además la capacidad de escoger la mejor ruta para que un determinado paquete de datos llegue a su destino
- Los routers son capaces de interconectar varias redes y generalmente trabajan en conjunto con hubs y switches.

Similitudes:

- Ambos envían información en elementos discretos (tramas vs datagramas)
 - Ambos tienen una tabla para decidir como enviar los paquetes
2. ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow?
Un switch normal funciona independientemente del resto de la red.

Un switch OpenFlow, cuando recibe un paquete, para el cual no tiene un flujo de salida, es decir, que en la tabla del switch no tiene un match respecto de la direccion de entrada, se pondrá en contacto con un controlador y le preguntará qué debe hacer con este paquete. Luego, el controlador puede actualizar la tabla del switch, posiblemente incluyendo alguna manipulación de paquetes. Una vez que el flujo se descarga al conmutador, cambiará los paquetes similares a velocidad de cable.

3. ¿Se pueden reemplazar todos los routers de la Internet por Switches OpenFlow?(Piense en el escenario interASes para elaborar su respuesta)
Si pensamos en router normales se podrian cambiar facilmente, pero habria que tener en cuenta que debe haber varios controladores, porque si todos acceden al mismo, este mismo colapsaria y no podria atender a todos los

pedidos al mismo tiempo (siedo que estamos remplazando muchos router de toda la internet).

Si pensamos en los sistemas autonomos, tenemos los routers de borde que tiene una caracteristica en particular, y esta es que manejan un protocolo llamado BGP en donde se puede decidir que prefijos se dan a conocer, se elijen los protocolos de ruteo, se determina la ingenieria de trafico y demas características. Por lo tanto habria que tener un controlador especial para estos donde se pueda manejar la logica del mismo y que ademas sean configurables ya que sabemos que la discrecionalidad de los prefijos se da por acuerdo comerciales, entonces, frente a cambios en estos acuerdos, se debe poder cambiar dicha discrecionalidad de una manera efectiva en el controlador de manera tal que sea flexible frente a cambios. Esto haria que sea mas facil la configuracion.

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.

```
$ sh scripts/conect_to_VM.sh
```

Nos pedira la cotraseña de nuetra 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:

```
$ sh scripts/resync.sh
```

3.3. Topologia

La topologia que se desarrollo se llama Fat tree. En la siguiente figura vemos nuetra 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.

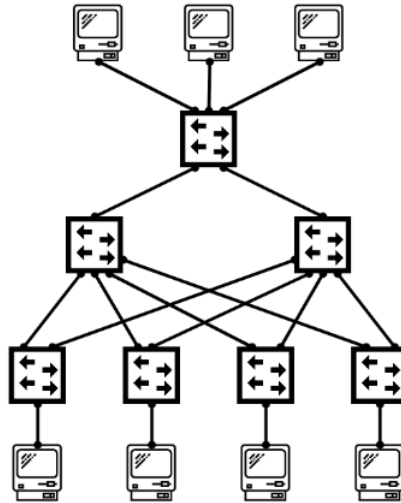


Figura 1: Ejemplo de como seria la topologia con un arbol de altura 3.

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:

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

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

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

Hay una herramienta para visualizar topologias a partir de nuestra salida en mininet. Si en mininet escribimos el comando `dump` y el comando `links` podemos usar cada salida en una pagina de internet, que el link esta en la referencias, la

cual se pega cada salida como se explica en dicha pagina y nos genera un arbol. De esta manera podemos comprobar que nuestra topologia se creo correctamente como se ve a continuacion:

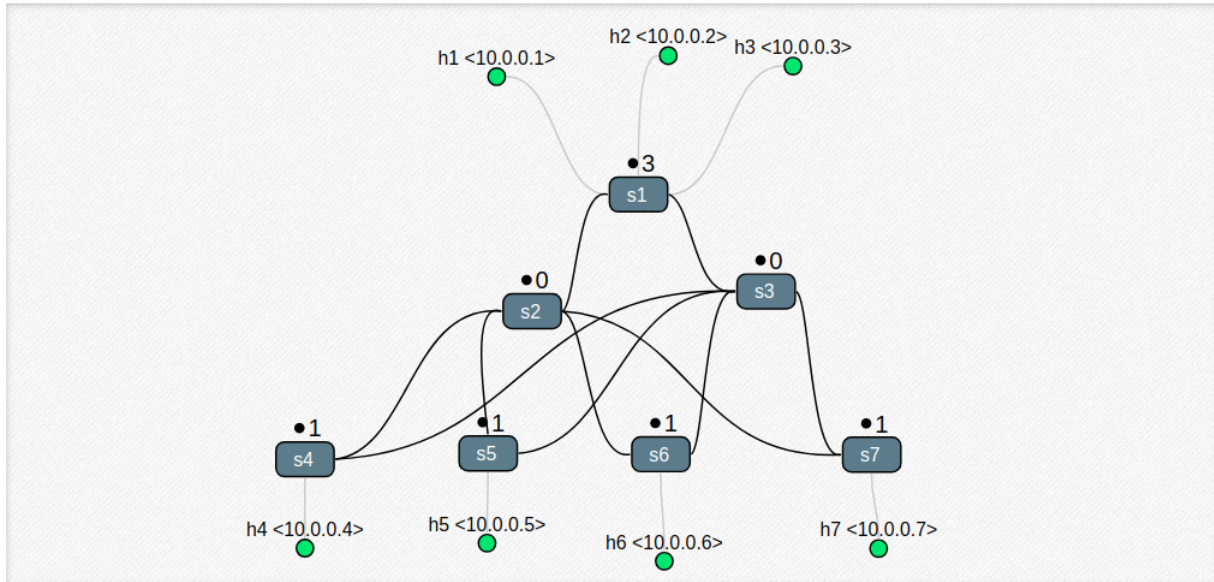


Figura 2: Topologia generada con los comandos dump y links de mininet.

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 envian mensajes LLDP entre los switches 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 encuentra 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 y 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 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 es 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` y cerrar terminal
3. Luego, abrir una 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 switches 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 3: 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 4: 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 `xterm h1 h4` 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 5: 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 6: Conexión TCP servidor host 4

No.	Time	Source	Destination	Protocol	Length	Info
4	9.4240716...	10.0.0.0	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...	10.0.0.0	10.0.0.4	TCP	66	37066 → 80 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=7364 TSecr=7354
7	9.5036656...	10.0.0.0	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...	10.0.0.0	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=25 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7354
9	9.5036749...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	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...	10.0.0.0	10.0.0.4	TCP	1514	37066 → 80 [ACK] Seq=24641 Ack=1 Win=29696 Len=1448 TSval=7364 TSecr=7364 [TCP segment ...]

Figura 7: Captura de wireshark monitoreando la interfaz eth1 del switch s2

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

Figura 8: Captura de wireshark monitoreando la interfaz eth1 del switch s3

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:

4.4. Balanceo de cargas con ping

Para realizar esta prueba vamos a usar mininet 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 switch. 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 esta vacia y la otra recibe los paquetes, pero al hacer el ping entre h2 y h5 debemos poder ver los mismo pero en los switches invertidos.

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



Figura 9: Captura de wireshark monitoreando la interfaz eth1 del switch s2

No.	Time	Source	Destination	Protocol	Length	Info
2	0.8724722	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=84/21584, ttl=64 (reply in 3)
3	0.8725208	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=84/21584, ttl=64 (request in 2)
6	1.8764393	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=85/21760, ttl=64 (reply in 7)
7	1.8765889	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=85/21760, ttl=64 (request in 6)
9	2.8778261	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=86/22016, ttl=64 (reply in 10)
10	2.8778862	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=86/22016, ttl=64 (request in 9)
12	3.8768196	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=87/22272, ttl=64 (reply in 13)
13	3.8768754	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=87/22272, ttl=64 (request in 12)
15	4.8765478	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=88/22528, ttl=64 (reply in 16)
16	4.8766012	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=88/22528, ttl=64 (request in 15)
17	5.8772786	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=89/22784, ttl=64 (reply in 18)
18	5.8773186	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=89/22784, ttl=64 (request in 17)
22	6.8788166	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=90/23040, ttl=64 (reply in 23)
23	6.8788853	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=90/23040, ttl=64 (request in 22)
27	7.8783709	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=91/23296, ttl=64 (reply in 28)
28	7.8784353	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=91/23296, ttl=64 (request in 27)
31	8.8773649	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=92/23552, ttl=64 (reply in 32)
32	8.8774289	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=92/23552, ttl=64 (request in 31)
35	9.8765588	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=93/23808, ttl=64 (reply in 36)
36	9.8766240	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=93/23808, ttl=64 (request in 35)
40	10.888716	10.0...	10.0.0.5	ICMP	98	Echo (ping) request id=0x0c4d, seq=94/24064, ttl=64 (reply in 41)
41	10.888764	10.0...	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0c4d, seq=94/24064, ttl=64 (request in 40)

Figura 10: 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	6.2044508	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 11: Captura de wireshark monitoreando la interfaz eth1 del switch s2

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

Figura 12: 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 levantamos la topologia) y en el prompt de mininet escribiremos `xterm h1 h5` para abrir dos terminales en cada host.

Luego vamos a abrir otra terminal (conectada a la VM como se explico antes) y escribiremos: `sudo wireshark &`. Y abriremos la interfaz `s5-eth1`. Luego en la terminal del host 5 levantaremos el servidor escribiendo lo siguiente:

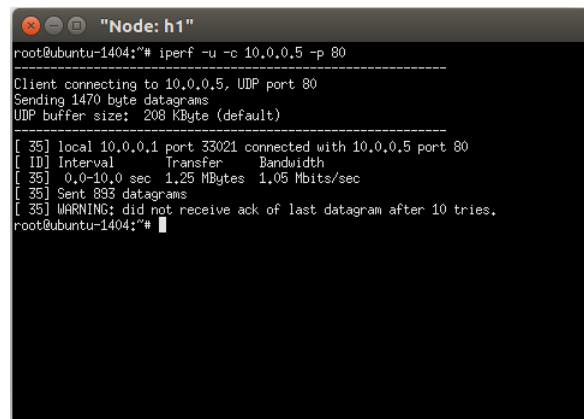
```
$ iperf -u -s -p 80
```

En la terminal del host 1 escribiremos lo siguiente para hacer la conexcion:

```
$ iperf -u -c 10.0.0.4 -p 80
```

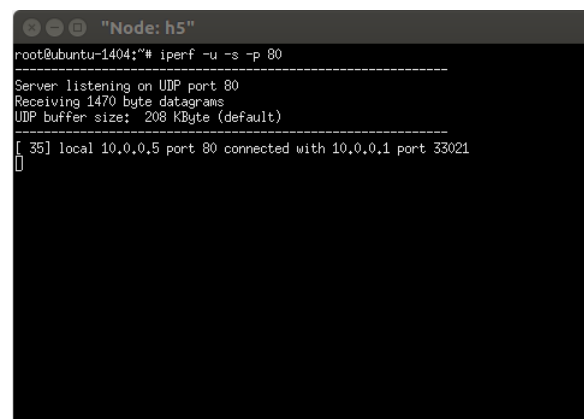
Y lo que haremos es ver ambas ventanas de wireshark y comprobar que los paquetes TCP solo pasan por algunas de las dos interfaz, pero no en ambas.

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



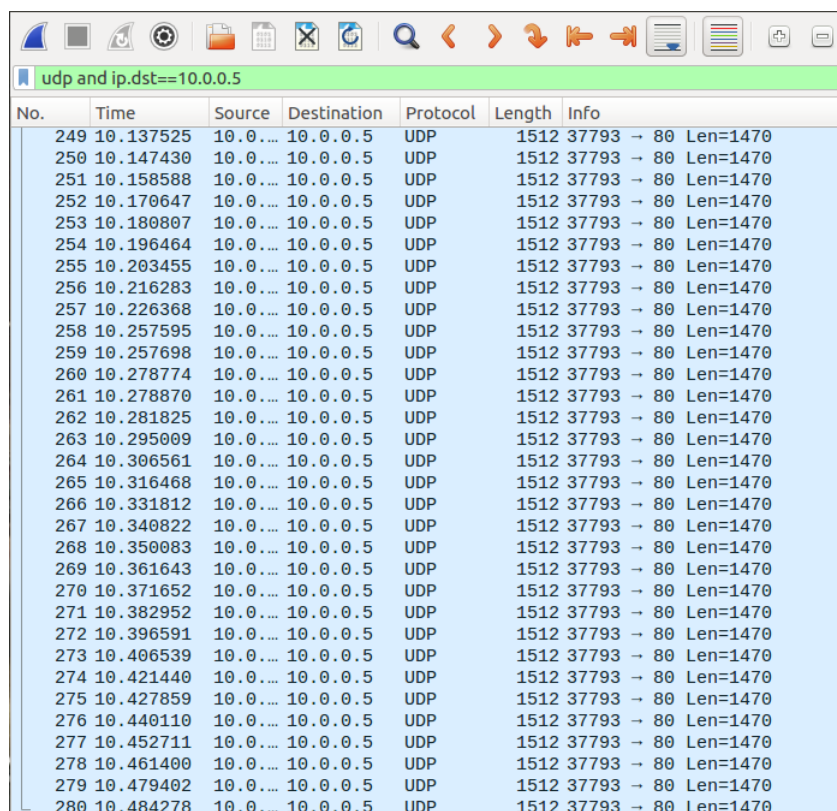
```
root@ubuntu-1404:~# iperf -u -c 10.0.0.5 -p 80
-----
Client connecting to 10.0.0.5, UDP port 80
Sending 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 35] local 10.0.0.1 port 33021 connected with 10.0.0.5 port 80
[ ID] Interval      Transfer    Bandwidth
[ 35] 0.0-10.0 sec  1.25 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 13: 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
0
```

Figura 14: 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 15: 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 `tshark`, se lee la captura que se hizo con `wireshark` y se comprueba lo que se mostro anteriormente. Pero queda como tarea a realizar, poder ejecutar desde un solo script un 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 se automatiza todo el proyecto. Para correr las pruebas, se tiene que ejecutar el siguiente comando: `sh scripts/test.sh` el cual ejecuta un script que descomprime un archivo en la carpeta `wireshark` que contiene los archivos de `wireshark` capturados y luego se ejecutan los test unitarios.

5. Conclusiones

Como conclusion podemos decir varias cosas. En primero lugar es muy interesante como se puede manipular el trafico de una cierta red, lo cual nos lleva a decir que la ingenieria de trafico es mas facil de llevar a cabo. Esto nos puede llegar a pensar que frente a como se encuentra hoy las redes, se podrian cambiar todos los routers por switches openflow.

En segundo lugar podemos decir que frente a ataques de denegacion de servicio, nos trae muchas ventajas, ya que mitigar dichos ataques se hace mucho mas facil. Pudimos ver el caso de los datagramas udp, pero tambien podemos recibir una inundacion de paquetes que consideramos validos, y frente a esa situacion, se puede tener un firewall que se encargue del problema y de esta manera se puede tener una reaccion mas efectiva y rapida frente a eso.

Finalmente se puede decir que respecto a las herramientas usadas, no son muy comodas, ya que para hacer las pruebas hay que levantar muchas terminales y se hace molesto. Ademas de que la maquina virtual suele trabarse y hacer lenta la ejecucion de los programas. De todas maneras se trato de llevar a cabo tests automaticos pero se hace dificil ya que no se encuentra una manera accesible de mockear la coneccion cliente-servidor y de esa manera poder generar desde un solo script la ejecucion del controlador, topologia, ventanas de wirechark o tcpdump y terminales de los host para las conecciones de iperf.

De todas maneras fue el tp mas interesante que hicimos en el cuatrimestre y sobre todo, desde el punto de vista de la programacion nos hace ver cosas distintas para aplicar nuestros conocimientos.

6. Anexos (Código)

6.1. topology.py

```
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

6.2. contoller.py

```
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',
24 ↪ 'openflow_discovery'))
25
26         self.event = None
27         self.dpid = None
28         self.in_port = None
29         self.packet = None
30         self.dst_dpid = None
31         self.out_port = None
32         self.table = ECMPTable()
33         self.eth_packet = None
34         self.ip_packet = None
35         self.arp_packet = None
36         self.icmp_packet = None
37         self.tcp_packet = None
38         self.udp_packet = None
39         self.net_packet = None
40         self.protocol_packet = None
41         self.protocol = None
42         self.arp_table = {}
43         self.is_ip = True
44         self.adjacency = {}
45         self.host_tracker = host_tracker()
46         log.info("controller ready")
47
48     def add_adjacency(self, dpid1, port1, dpid2, port2):
49         if dpid1 not in self.adjacency:
50             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         self.print_adjacents()
81
82         ↪ log.info("_____")
83
84     def _handle_ConnectionUp(self, event):
85         log.debug("Connection %s" % (event.connection,))
86
87         msg = of.ofp_flow_mod()
88         msg.match.dl_dst = ETHER_BROADCAST
89         msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
90         event.connection.send(msg)
91
92         msg = of.ofp_flow_mod()
93         msg.match.dl_type = pkt.ethernet.IPV6_TYPE
94         event.connection.send(msg)
95
96         msg = of.ofp_flow_mod()
97         msg.match.dl_type = pkt.ethernet.ARP_TYPE
98         msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
99         event.connection.send(msg)
100
101     def fill_arp_table(self):

```



```

98     entry = self.host_tracker.getMacEntry(self.addr_dst)
99     if entry is None:
100         log.info("HOST TRACKER COULD NOT FIND ENTRY DST")
101         return
102     self.arp_table[self.addr_dst] = {
103         "dpid": entry.dpid,
104         "port": entry.port
105     }
106
107     def print_adjacents(self):
108         msg = ""
109         for dpid in self.adjacency:
110             msg += "dpid: %s: [" % dpid
111             for port in self.adjacency[dpid]:
112                 msg += "%s, " % self.adjacency[dpid][port]["dpid"]
113             msg += "]"
114             log.info(msg)
115             msg = ""
116
117     def _handle_PacketIn(self, event):
118         self.host_tracker._handle_PacketIn(event)
119         self.event = event
120         self.dpid = event.connection.dpid
121
122         ↪ log.info("_____")
123         log.info("SWITCH %s" % self.dpid)
124         self.in_port = event.port
125         self.packet = event.parsed
126         log.info("ports: %s" % event.connection.ports)
127         log.info("ports: %s" % event.connection.ports)
128         log.info("in port: %s" % self.in_port)
129         if not self.packet.parsed:
130             log.warning("%s %s ignoring unparsed packet" %
131                         (self.dpid, self.in_port))
132             return
133         log.info("HOST SRC %s" % self.packet.src)
134         log.info("HOST DST: %s" % self.packet.dst)
135         self.eth_packet = self.packet.find(pkt.ethernet)
136         self.addr_dst = self.packet.dst
137         self.fill_arp_table()
138         self.ip_packet = self.packet.find(pkt.ipv4)
139         self.arp_packet = self.packet.find(pkt.arp)
140         self.icmp_packet = self.packet.find(pkt.icmp)
141         self.tcp_packet = self.packet.find(pkt.tcp)
142         self.udp_packet = self.packet.find(pkt.udp)
143
144         if not self.validate_protocols():
145             return
146         if not self.validate_net_packets():
147             return
148

```

```

149         if self.addr_dst not in self.arp_table:
150             log.warning("Could not find dst")
151             return self.flood()
152
153         entry = self.arp_table[self.addr_dst]
154         self.dst_dpid = entry["dpid"]
155         if self.dpid == self.dst_dpid:
156             log.info("Current switch is destination")
157             self.out_port = entry["port"]
158         else:
159             if self.packet.dst.is_multicast:
160                 return self.flood()
161             log.info("Finding minimum paths from %s to %s"
162                     %(self.dpid, self.dst_dpid))
163             minimum_paths = self.get_minimum_paths()
164             log.info("finding out port")
165             self.out_port = self.get_out_port(minimum_paths)
166             if self.out_port is None:
167                 log.info("Could not find out port")
168                 return
169             log.info("out port: %s" % self.out_port)
170             log.info("Updating flow table")
171             self.update_table()
172             log.info("Sending packet")
173             self.send_packet()
174
175     def flood(self):
176         log.info("FLOODING PACKET")
177         msg = of.ofp_packet_out()
178         msg.buffer_id = self.event.ofp.buffer_id
179         msg.actions.append(of.ofp_action_output(port=of.OFPP_FLOOD))
180         msg.data = self.event.ofp
181         msg.in_port = self.in_port
182         self.event.connection.send(msg)
183
184     def validate_protocols(self):
185         if self.udp_packet is not None:
186             log.info("UDP packet!")
187             self.protocol = "UDP"
188             self.protocol_packet = self.udp_packet
189             return True
190         elif self.tcp_packet is not None:
191             log.info("TCP packet!")
192             self.protocol = "TCP"
193             self.protocol_packet = self.tcp_packet
194             return True
195         elif self.icmp_packet is not None:
196             log.info("ICMP packet!")
197             self.protocol = "ICMP"
198             self.protocol_packet = self.icmp_packet
199             return True
200         else:

```

```

201         log.warning("icmp, tcp and udp packets are None!")
202         return False
203
204     def validate_net_packets(self):
205         if _ethtype_to_str[self.packet.type] == "IPv6":
206             log.warning("DROP IPv6 packet")
207             return False
208         if self.eth_packet is None:
209             log.warning("ETHERNET packet is None!")
210             return False
211         if self.ip_packet is not None:
212             log.info("IP packet!")
213             self.is_ip = True
214             self.net_packet = self.ip_packet
215         elif self.arp_packet is not None:
216             log.info("ARP packet!")
217             self.is_ip = False
218             self.net_packet = self.arp_packet
219         else:
220             log.warning("ARP and TCP packets are None!")
221             return False
222         return True
223
224     def match_protocol_packets(self, msg):
225         if self.is_ip:
226             msg.match.nw_src = self.net_packet.srcip
227             msg.match.nw_dst = self.net_packet.dstip
228             msg.match.nw_proto = self.net_packet.protocol
229             return msg
230         msg.match.nw_src = self.net_packet.protosrc
231         msg.match.nw_dst = self.net_packet.protodst
232         msg.match.nw_proto = self.net_packet.prototype
233         return msg
234
235     def match_packet(self, msg):
236         if not self.is_ip:
237             return msg
238         msg.match.nw_src = self.ip_packet.srcip
239         msg.match.nw_dst = self.ip_packet.dstip
240         msg.match.nw_proto = self.ip_packet.protocol
241         return msg
242
243     def update_table(self):
244         msg = of.ofp_flow_mod()
245         msg.match.dl_type = self.eth_packet.type
246         msg = self.match_packet(msg)
247         msg.buffer_id = self.event.ofp.buffer_id
248         if self.protocol != "ICMP":
249             msg.match.tp_src = self.protocol_packet.srcport
250             msg.match.tp_dst = self.protocol_packet.dstport
251         msg.actions.append(of.ofp_action_output(port=self.out_port))
252         self.event.connection.send(msg)

```

```

253         self.balance_of_charges()
254
255     def get_minimun_paths(self):
256         adjacents = self.get_adjacents(self.dpid)
257         if not adjacents:
258             log.warning("NO ADJACENTS FOUND")
259             return []
260         paths = [[neighbour] for neighbour in adjacents]
261         while not self.has_found_a_path(paths, self.dst_dpid):
262             last_paths = paths[:]
263             for path in last_paths:
264                 adjacents = self.get_adjacents(path[-1]["dpid"])
265                 for an_adjacent in adjacents:
266                     if an_adjacent["dpid"] != self.dpid:
267                         if not self.node_belongs_path(an_adjacent,
↪ path):
268                             paths.append(path + [an_adjacent])
269             return self.filter_paths(paths, self.dst_dpid)
270
271     def node_belongs_path(self, node, path):
272         dpid = node["dpid"]
273         for a_node in path:
274             if a_node["dpid"] == dpid:
275                 return True
276         return False
277
278     def get_out_port(self, paths_to_dst):
279         if len(paths_to_dst) == 0:
280             return None
281         ports = self.get_all_ports(paths_to_dst)
282         data = (
283             ports,
284             self.dpid,
285             self.dst_dpid,
286             self.protocol,
287             self.packet.src,
288             self.packet.dst
289         )
290         return self.table.get_port_applying_ecmp(data)
291
292     def balance_of_charges(self):
293         log.info("saving (%s, %, %): %s" %
294             (self.dpid, self.dst_dpid, self.protocol,
↪ self.out_port))
295         data = (
296             self.dpid,
297             self.dst_dpid,
298             self.protocol,
299             self.packet.src,
300             self.packet.dst,
301             self.out_port
302         )

```

```

303         self.table.save_port(data)
304
305     @staticmethod
306     def get_all_ports(paths_to_dst):
307         return [a_path[0]["port"] for a_path in paths_to_dst]
308
309     @staticmethod
310     def filter_paths(paths, dpid):
311         paths_to_dst = []
312         for path in paths:
313             if path[-1]["dpid"] != dpid:
314                 continue
315             paths_to_dst.append(path)
316         return paths_to_dst
317
318     def has_found_a_path(self, paths, dpid):
319         for path in paths:
320             if path[-1]["dpid"] == dpid:
321                 log.info("FOUND A PATH!")
322                 return True
323         return False
324
325     def get_adjacents(self, dpid):
326         adjacents = []
327         if dpid not in self.adjacency:
328             return adjacents
329         for port in self.adjacency[dpid]:
330             adjacents.append({
331                 "dpid": self.adjacency[dpid][port]["dpid"],
332                 "port": port
333             })
334         return adjacents
335
336     def filter_repeated(self, adjacents):
337         filtered = []
338         belongs = False
339         for an_adjacent in adjacents:
340             for final_adjacent in filtered:
341                 dpid_1 = final_adjacent["dpid"]
342                 dpid_2 = an_adjacent["dpid"]
343                 port_1 = final_adjacent["port"]
344                 port_2 = an_adjacent["port"]
345                 if dpid_1 == dpid_2 and port_1 == port_2:
346                     belongs = True
347                     break
348             if not belongs:
349                 filtered.append(an_adjacent)
350                 belongs = False
351         return filtered
352
353     def send_packet(self):
354         msg = of.ofp_packet_out()

```

```
355     msg.actions.append(of.ofp_action_output(port=self.out_port))
356     msg.data = self.event.ofp
357     msg.buffer_id = self.event.ofp.buffer_id
358     msg.in_port = self.in_port
359     self.event.connection.send(msg)
```

Listing 2: Controller

6.3. firewall.py

```
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.recoco 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     def request_for_switch_statistics(self):
30         for connection in core.openflow.connections:
31             body = of.ofp_flow_stats_request()
32             connection.send(of.ofp_stats_request(body=body))
33
34     def _handle_flowstats_received(self, event):
35         log.info("handle denial of service")
36         self.dpid = event.connection.dpid
37         self.total_udp_flow_packets = {}
38         for flow in event.stats:
39             self.dst_ip = flow.match.nw_dst
40             if self.dst_ip is None:
41                 log.info("DST IP IS NONE. COULD NOT HANDLE DoS")
42                 continue
43             if not self.get_udp_flow(flow):
44                 continue
45             self.evaluate_blocking()
46             current = self.total_udp_flow_packets[self.dst_ip]
47             self.last_udp_flow_packets[self.dpid] = {}
48             self.last_udp_flow_packets[self.dpid][self.dst_ip] =
49                 ↪ current
```

```

50     def get_udp_flow(self, flow):
51         if self.dst_ip is None or flow.match.nw_proto !=
↪ UDP_PROTOCOL:
52             return False
53         if self.dst_ip not in self.total_udp_flow_packets:
54             self.total_udp_flow_packets[self.dst_ip] =
↪ flow.packet_count
55         else:
56             self.total_udp_flow_packets[self.dst_ip] +=
↪ flow.packet_count
57         return True
58
59     def get_last_udp_flow_packets(self, dst_ip):
60         if self.dpid not in self.last_udp_flow_packets:
61             self.last_udp_flow_packets[self.dpid] = {}
62             self.last_udp_flow_packets[self.dpid][dst_ip] = 0
63         return 0
64         elif dst_ip not in self.last_udp_flow_packets[self.dpid]:
65             self.last_udp_flow_packets[self.dpid][dst_ip] = 0
66         return self.last_udp_flow_packets[self.dpid][dst_ip]
67
68     def evaluate_blocking(self):
69         for dst_ip in self.total_udp_flow_packets:
70             current = self.total_udp_flow_packets[dst_ip]
71             last = self.last_udp_flow_packets.get(self.dpid,
↪ {}).get(dst_ip, 0)
72             if (current - last) > self.MAX_UDP_PACKETS:
73                 self.block_udp_packet(dst_ip)
74             else:
75                 self.unblock_udp_packet(dst_ip)
76
77     def block_udp_packet(self, dst_ip):
78         log.info("BLOCKING UDP PACKET IN %s" % dst_ip)
79         if dst_ip not in self.blocked_udp_packets:
80             log.info("Blocking ip: %s" % dst_ip)
81             msg = of.ofp_flow_mod()
82             msg.match.nw_proto = UDP_PROTOCOL
83             msg.match.dl_type = IP_TYPE
84             msg.priority = of.OFP_DEFAULT_PRIORITY + 1
85             msg.match.nw_dst = dst_ip
86             self.send_message_to_all(msg)
87             self.blocked_udp_packets[dst_ip] = time()
88
89     def unblock_udp_packet(self, dst_ip):
90         if dst_ip not in self.blocked_udp_packets:
91             return
92         time_passed = time() - self.blocked_udp_packets[dst_ip]
93         if time_passed < self.MAX_UDP_TIME:
94             return
95         del self.blocked_udp_packets[dst_ip]
96         log.info("UNBLOCKING UDP PACKET IN %s" % dst_ip)
97         log.info("unblocking ip: %s" % dst_ip)

```



```
98     msg = of.ofp_flow_mod()
99     msg.match.nw_proto = UDP_PROTOCOL
100    msg.match.dl_type = IP_TYPE
101    msg.command = of.OFPFC_DELETE
102    msg.match.nw_dst = dst_ip
103    self.send_message_to_all(msg)
104
105    @staticmethod
106    def send_message_to_all(msg):
107        for a_connection in core.openflow.connections:
108            a_connection.send(msg)
```

Listing 3: Firewall

6.4. launch.py

```
1 def launch():
2     import pox.log.color
3     import pox.log
4     import pox.log.level
5     import logging
6     from pox.core import core
7     import pox.openflow.discovery
8     from controller import Controller
9     import pox.openflow.spanning_tree
10    from firewall import Firewall
11
12    pox.log.color.launch()
13    pox.log.launch(format=" [@@@bold@@@level %(name)-22s@@@reset] " +
14                    "@@@bold %(message)s@@@normal")
15    pox.log.level.launch(packet=logging.WARN,
16    ↪ host_tracker=logging.INFO)
17    pox.openflow.discovery.launch()
18    core.registerNew(Controller)
19    pox.openflow.spanning_tree.launch()
20    core.registerNew(Firewall)
```

Listing 4: file to launch controller and firewall

6.5. ecmp_table.py

```
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         if dpid not in self.table:
35             self.table[dpid] = {}
36         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/>
3. POX DOCUMENTATION
<https://noxrepo.github.io/pox-doc/html/>
4. VISUALIZADOR DE TOPOLOGIAS
<http://demo.spear.narmox.com/app/?apiurl=demo#!/mininet>
5. LEARNING SWITCH
https://github.com/att/pox/blob/master/pox/forwarding/l2_learning.py