

TP2: Software-Defined Networks

[75.43/75.33/95.60]

Introducción a los Sistemas Distribuidos

2C 2022

Grupo 11

Alumno	Padrón	Email
Gomez, Joaquin	103735	joagomez@fi.uba.ar
Grassano, Bruno	103855	bgrassano@fi.uba.ar
Opizzi, Juan Cruz	99807	jopizzi@fi.uba.ar
Stancanelli, Guillermo	104244	gstancanelli@fi.uba.ar
Valdez, Santiago	103785	svaldez@fi.uba.ar

Índice

1. Introducción	2
2. Hipótesis y suposiciones realizadas	2
3. Implementación	2
3.1. Topología	2
3.2. Firewall	3
4. Ejecución	4
5. Pruebas	5
5.1. Bloqueo a cualquier host con puerto destino 80	5
5.2. Bloqueo de paquetes provenientes del host1, con protocolo UDP y puerto destino 5001	8
5.3. Bloqueo de paquetes entre dos hosts elegidos arbitrariamente	10
6. Pruebas Adicionales	13
6.1. Bloqueo a cualquier host con puerto destino 80	13
6.2. Bloqueo de paquetes UDP provenientes del host1 con puerto destino 5001	14
7. Preguntas a responder	15
7.1. ¿Cuál es la diferencia entre un Switch y un router? ¿Qué tienen en común?	15
7.2. ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow?	15
7.3. ¿Se pueden reemplazar todos los routers de la Internet por Switches OpenFlow? . .	15
8. Dificultades encontradas	16
9. Conclusiones	16
10. Referencias	16

1. Introducción

El presente trabajo práctico tiene como objetivo familiarizarse con los desafíos por los cuales surgen las SDNs y el protocolo OpenFlow, a través del cual se programan los dispositivos de red. Dado que ahora los dispositivos son programables, también se buscará aprender a controlar el funcionamiento de los switches a través de una API.

2. Hipótesis y suposiciones realizadas

Para la realización del trabajo practico se tomaron las siguientes hipótesis y supuestos.

- Es valido que la topología tenga un solo *switch*. Este sera el mínimo posible.
- Al elegir dos *hosts* cualquiera a bloquear, estos tienen que estar en los lados opuestos de la cadena de *switches* si se tiene más de un *switch* y el *firewall* no esta en el borde mas cercano.

3. Implementación

3.1. Topología

Para el trabajo se pidió la elaboración de una topología parametrizable sobre la cual probar diferentes funcionalidades.

La parametrización a través de `number_of_switches` permite variar la cantidad de *switches* que presenta la red formando una cadena con dos *hosts* en los extremos.

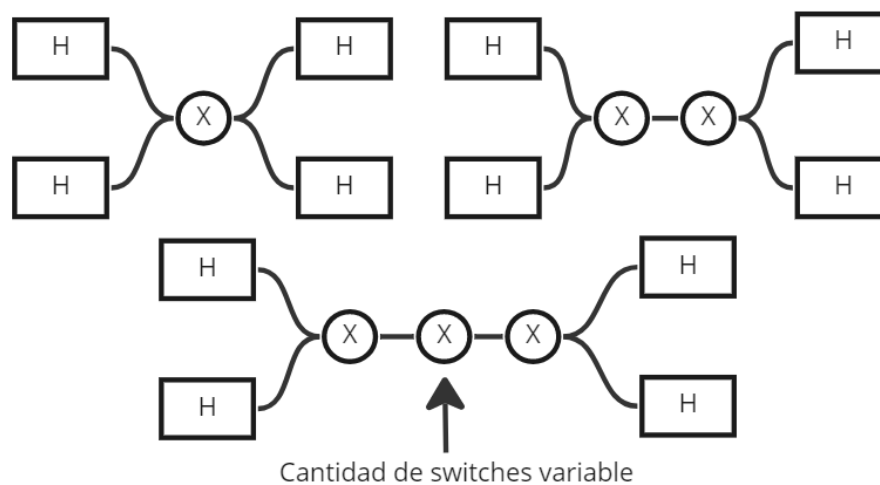


Figura 1: Ejemplos de topologías posibles, cambia la cantidad de switches.

Se realizaron algunas pruebas unitarias para verificar el armado de la topología. Estas pueden ser ejecutadas mediante:

```
python3 -m unittest topology_tests.py
```

3.2. Firewall

La implementación del Firewall consistió en el armado de las siguientes reglas. Para esto nos apoyamos en la biblioteca de *pox*.

- Se deben descartar todos los mensajes cuyo puerto destino sea 80.
- Se deben descartar todos los mensajes que provengan del host 1, tengan como puerto destino el 5001, y estén utilizando el protocolo UDP.
- Se debe elegir dos *hosts* cualquiera, y los mismos no deben poder comunicarse de ninguna forma. En este caso tomamos los *hosts* 2 y 3, puede configurarse.

En la implementación se realizan especificaciones de protocolos adicionales debido a que al dejarlos libres la biblioteca hace *match* independientemente de los demás filtros, caso de IP, o ignora los campos, caso de TCP y UDP.

Estas reglas las definimos a través de un json que es parseado al iniciar el controlador y que cuando llega un evento son aplicadas.

Formato de las reglas

El formato para definir las reglas es el siguiente.

```
{
  "firewall_switch" : 1,
  "rules": [
    {
      "name": "Block TCP port 80",
      "rule": {
        "data_link": {
          "ip_type": "ipv4"
        },
        "network": {
          "protocol": "tcp"
        },
        "transport": {
          "dst_port": 80
        }
      }
    },
    ...
  ]
}
```

Los campos iniciales son:

- **firewall_switch** Indica que *switch* es el que aplica las reglas del firewall.
- **rules** Se definen las reglas a utilizar

Las reglas se definen adentro de un array de acuerdo a la siguiente estructura.

- **name** Nombre de la regla
- **rule** Comienza la definición de la regla en si
 - **data_link** Objeto para definir a nivel *data link*.
 - **ip_type** Puede tomar los valores de *ipv4* o *ipv6*
 - **mac** Objeto para definir la MAC address
 - ◇ **src** MAC origen
 - ◇ **dst** MAC destino
 - **network** Objeto para definir a nivel *network*.
 - **protocol** Puede tomar los valores *tcp,udp* o *icmp*
 - **src_ip** IP de origen
 - **dst_ip** IP de destino
 - **transport** Objeto para definir a nivel *transport*.
 - **src_port** Puerto origen
 - **dst_port** Puerto destino

Nota: Algunos campos se definen en estructuras que tienen nombres de una capa distinta a la que pertenecen para seguir la convención de POX.

4. Ejecución

Para poder ejecutar el trabajo es necesario tener *mininet* y *openvswitch*. Pox esta incluido en los archivos entregados.

Para *mininet* ejecutar.

```
sudo apt install mininet
```

```
sudo pip install mininet
```

Para *openvswitch*:

```
sudo apt install openvswitch-switch
```

```
systemctl start openvswitch
```

Una vez con las dependencias instaladas, se puede levantar el controlador con la siguiente linea. De esta forma se ejecutara el controlador con las reglas definidas en **rules.json**.

```
python3 pox.py log.level --DEBUG openflow.of_01 forwarding.l2_learning controller
```

Para abrir terminales desde *mininet* es necesario *xterm*. Estas se van a poder abrir desde *mininet* con *xterm nombreHost* o agregando la opción *-x* a la linea de *mininet*.

```
sudo apt install xterm
```

Para *mininet* se puede usar la siguiente linea. La cadena en el ejemplo es de 2 *switches*.

```
sudo mn --custom ./topology.py --topo chain,number_of_switches=2
      --mac --arp --switch ovsk --controller remote
```

5. Pruebas

A continuación se realizarán pruebas mediante la herramienta *iperf*. Gracias a la misma podremos configurar un servidor y un cliente en nuestra red virtual creada con mininet, y observar su comunicación a través de las correspondientes interfaces en *Wireshark*. Para dichas pruebas, nos enfocaremos en las tres reglas mencionadas en el enunciado del TP, y configuraremos al primer switch en la cadena de la topología como firewall.

5.1. Bloqueo a cualquier host con puerto destino 80

Primero probaremos el correcto funcionamiento de esta regla utilizando *iperf* con el protocolo de transporte TCP, y luego corroboraremos que también está vigente en UDP. El host4 de nuestra red funcionará en modo servidor, y el host1 en modo cliente.

```

"Node: host_4"
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -s -p 80
-----
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
-----
[]

"Node: host_1"
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -c 10.0.0.4 -p 80

```

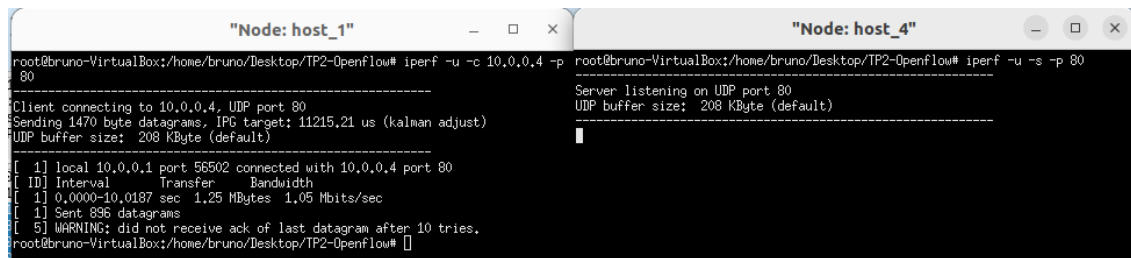
Figura 2: Podemos observar que al configurar tanto host como cliente por la terminal, el servidor no recibe mensaje alguno, ya que está siendo filtrado por nuestro firewall en el switch 1.

tcp.stream eq 1						
No.	Time	Source	Destination	Protocol	Length	Info
3	32.864814594	10.0.0.1	10.0.0.4	TCP	74	56424 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSv
4	33.887718151	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 56424 → 80 [SYN]
5	35.909483285	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 56424 → 80 [SYN]
6	39.941582189	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 56424 → 80 [SYN]
17	48.129385189	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 56424 → 80 [SYN]
29	64.257108419	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 56424 → 80 [SYN]
33	98.384122890	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 56424 → 80 [SYN]

▶ Frame 3: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface switch_1-eth1, id 0 ▶ Ethernet II, Src: 00:00:00:00:00:01 (00:00:00:00:00:01), Dst: 00:00:00:00:00:04 (00:00:00:00:00:04) ▶ Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.4 ▶ Transmission Control Protocol, Src Port: 56424, Dst Port: 80, Seq: 0, Len: 0						
---	--	--	--	--	--	--

0000	00 00 00 00 00 04 00 00	00 00 00 01 08 00 45 00E..
0010	00 3c 55 cf 40 00 40 06	d0 e8 0a 00 00 01 0a 00	<U@.@.....
0020	00 04 dc 68 00 50 d4 cc	c1 9a 00 00 00 00 a0 02	..hP.....
0030	a5 64 14 33 00 00 02 04	05 b4 04 02 08 0a 26 72	.d.3.....&r
0040	16 79 00 00 00 00 01 03	03 09	.y.....

Figura 3: Si además capturamos este flujo por wireshark en el lado del cliente, veremos que se hacen reiterados intentos por establecer un handshake con el servidor. Sin embargo, esto nunca se concreta justamente por las reglas establecidas.



```

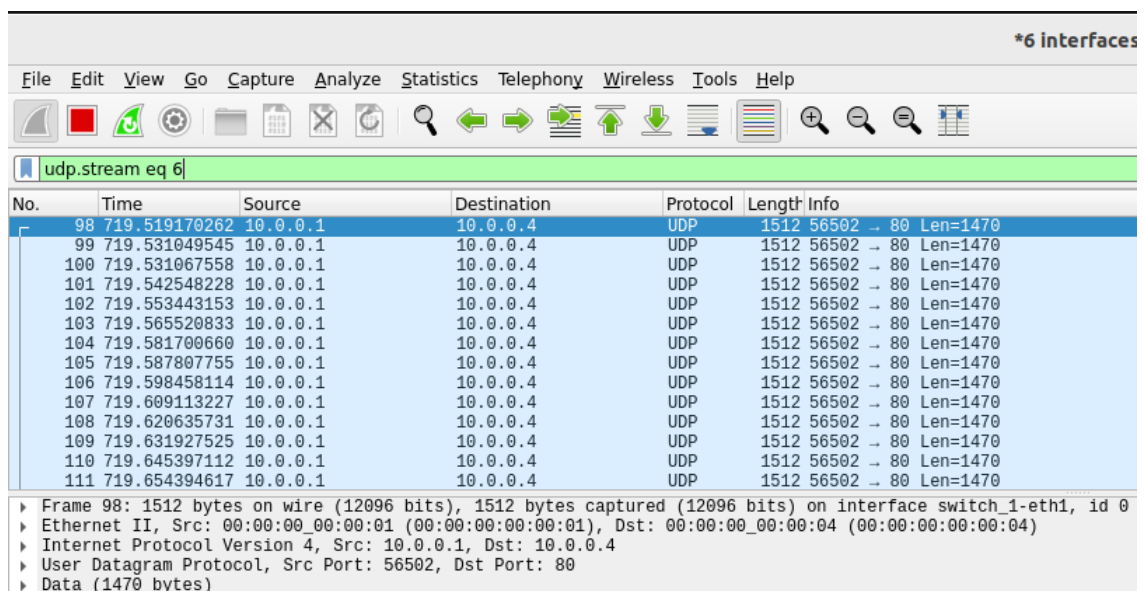
"Node: host_1"
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -u -c 10.0.0.4 -p 80
Client connecting to 10.0.0.4, UDP port 80
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)

[ 1] local 10.0.0.1 port 56502 connected with 10.0.0.4 port 80
[ ID] Interval      Transfer      Bandwidth
[ 1] 0.0000-10.0187 sec  1.25 MBytes  1.05 Mbits/sec
[ 1] Sent 896 datagrams
[ 5] WARNING: did not receive ack of last datagram after 10 tries.
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow#

"Node: host_4"
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -u -s -p 80
Server listening on UDP port 80
UDP buffer size: 208 KByte (default)

```

Figura 4: Pasando ahora a la ejecución de esta regla en UDP, podemos ver que si bien el cliente envía el paquete, este nunca llega a destino. Nos indica que el firewall sigue correctamente en funcionamiento.



No.	Time	Source	Destination	Protocol	Length	Info
98	719.519170262	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
99	719.531049545	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
100	719.531067558	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
101	719.542548228	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
102	719.553443153	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
103	719.565520833	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
104	719.581700660	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
105	719.587807755	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
106	719.598458114	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
107	719.609113227	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
108	719.620635731	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
109	719.631927525	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
110	719.645397112	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470
111	719.654394617	10.0.0.1	10.0.0.4	UDP	1512	56502 → 80 Len=1470

▶ Frame 98: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface switch_1-eth1, id 0
 ▶ Ethernet II, Src: 00:00:00:00:00:01 (00:00:00:00:00:01), Dst: 00:00:00:00:00:04 (00:00:00:00:00:04)
 ▶ Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.4
 ▶ User Datagram Protocol, Src Port: 56502, Dst Port: 80
 ▶ Data (1470 bytes)

Figura 5: Si nos colocamos en la interfaz de wireshark correspondiente a la entrada del switch firewall *switch₁ – eth1*, es el último momento donde podemos visualizar estos paquetes antes de ser filtrados.

```

root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -u -c 10.0.0.4 -p 81
Client connecting to 10.0.0.4, UDP port 81
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)
[ 1] local 10.0.0.1 port 51249 connected with 10.0.0.4 port 81
[ ID] Interval      Transfer    Bandwidth      Jitter    Lost/Total Datagrams
[ 1] 0.0000-10.0192 sec 1.25 MBytes 1.05 Mbits/sec  0.053 ms 0/895 (0%)
[ 1] Sent 896 datagrams
[ 1] Server Report:
[ ID] Interval      Transfer    Bandwidth      Jitter    Lost/Total Datagrams
[ 1] 0.0000-10.0110 sec 1.25 MBytes 1.05 Mbits/sec  0.053 ms 0/895 (0%)
[ 1] 0.0000-10.0110 sec 2 datagrams received out-of-order
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow#

```

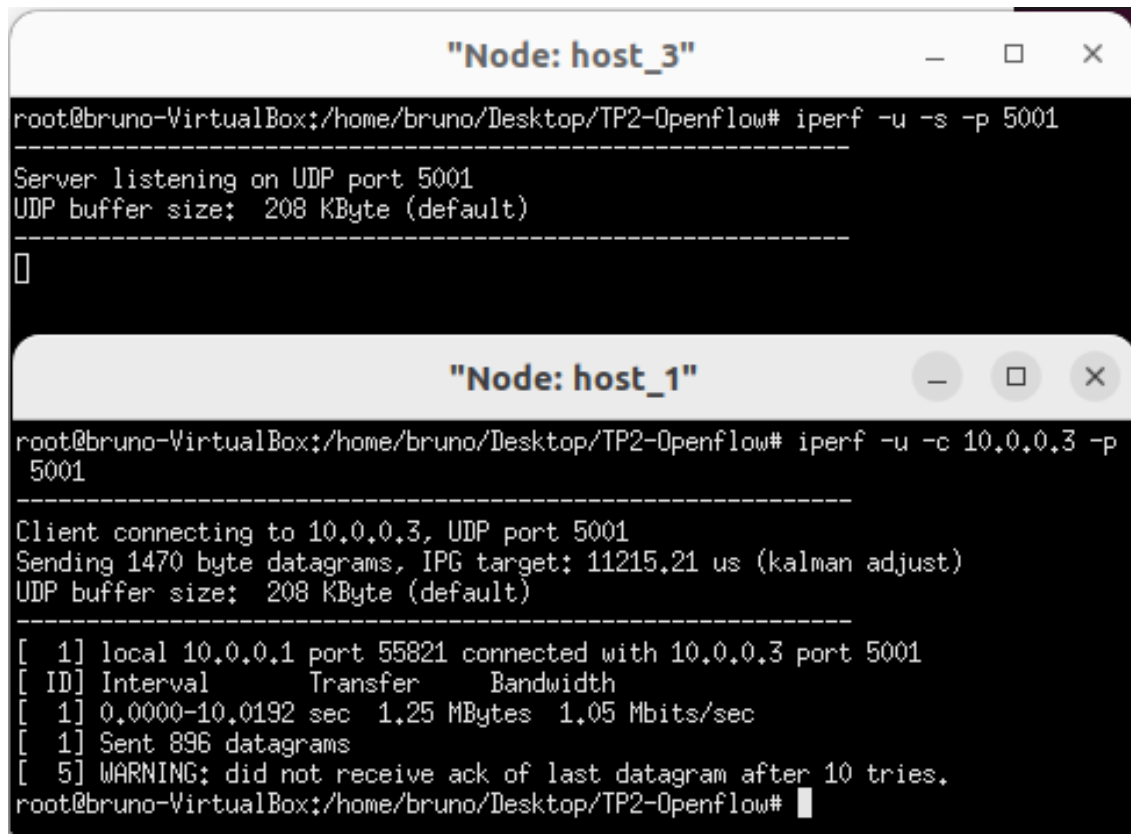
Figura 6: Si ahora cambiamos a otro puerto, podemos ver que los paquetes llegan del cliente al servidor, indicando que la aplicación de la regla no fue más restrictiva de lo intencionado.

udp.stream eq 7						
No.	Time	Source	Destination	Protocol	Length	Info
4770	1126.1168493...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4771	1126.1320885...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4772	1126.1388917...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4773	1126.1519053...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4774	1126.1607532...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4775	1126.1757546...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4776	1126.1892131...	10.0.0.4	10.0.0.1	UDP	170	81 → 51249 Len=128
4777	1126.1168400...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4778	1126.1320829...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4779	1126.1388840...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4780	1126.1518989...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4781	1126.1607465...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4782	1126.1757468...	10.0.0.1	10.0.0.4	UDP	1512	1512 51249 → 81 Len=1470
4783	1126.1919467...	10.0.0.4	10.0.0.1	UDP	170	81 → 51249 Len=128
▶ Frame 4782: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface switch_1-eth3, id 3 ▶ Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:04 (00:00:00:00:00:04) ▶ Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.4 ▶ User Datagram Protocol, Src Port: 51249, Dst Port: 81 ▶ Data (1470 bytes)						

Figura 7: Para corroborar esto, nos podemos poner en el otro lado del switch firewall *switch₁-eth3*, y esta vez sí veremos paquetes.

5.2. Bloqueo de paquetes provenientes del host1, con protocolo UDP y puerto destino 5001

Esta vez el host3 actuará como servidor, y el host1 como cliente, volviendo a ser necesario atravesar la cadena de switches para comunicarse. El switch 1 nuevamente actuará como firewall.



```
"Node: host_3"
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -u -s -p 5001
-----
Server listening on UDP port 5001
UDP buffer size: 208 KByte (default)
-----
[]

"Node: host_1"
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -u -c 10.0.0.3 -p 5001
-----
Client connecting to 10.0.0.3, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.1 port 55821 connected with 10.0.0.3 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-10.0192 sec 1.25 MBytes 1.05 Mbits/sec
[ 1] Sent 896 datagrams
[ 5] WARNING: did not receive ack of last datagram after 10 tries.
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow#
```

Figura 8: Nuevamente podemos observar que al intentar enviar paquetes que cumplen con los clientes filtros de la regla, estos nunca llegan a destino y somos advertidos del timeout ocurrido.

udp.stream eq 8						
No.	Time	Source	Destination	Protocol	Length	Info
4832	2782.2352790...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4833	2782.2532663...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4834	2782.2533325...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4835	2782.2650188...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4836	2782.2708903...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4837	2782.2821938...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4838	2782.2920969...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4839	2782.3054089...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4840	2782.3279174...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4841	2782.3279375...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4842	2782.3374404...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4843	2782.3542856...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4844	2782.3594737...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470
4845	2782.3753466...	10.0.0.1	10.0.0.3	UDP	1512	55821 → 5001 Len=1470

▶ Frame 4832: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface switch_1-eth1, id 0
 ▶ Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:03 (00:00:00:00:00:03)
 ▶ Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.3
 ▶ User Datagram Protocol, Src Port: 55821, Dst Port: 5001
 ▶ Data (1470 bytes)

Figura 9: Observando wireshark vemos que antes de ser filtrados por el switch firewall, los paquetes UDP sí habían salido correctamente desde su origen, el host1, pero nunca llegaron mas allá de esta interfaz.

```

"Node: host_3"
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -s -p 5001
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 1] local 10.0.0.3 port 5001 connected with 10.0.0.1 port 54658
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-10.0001 sec 3.99 GBytes 3.43 Gbits/sec
[]

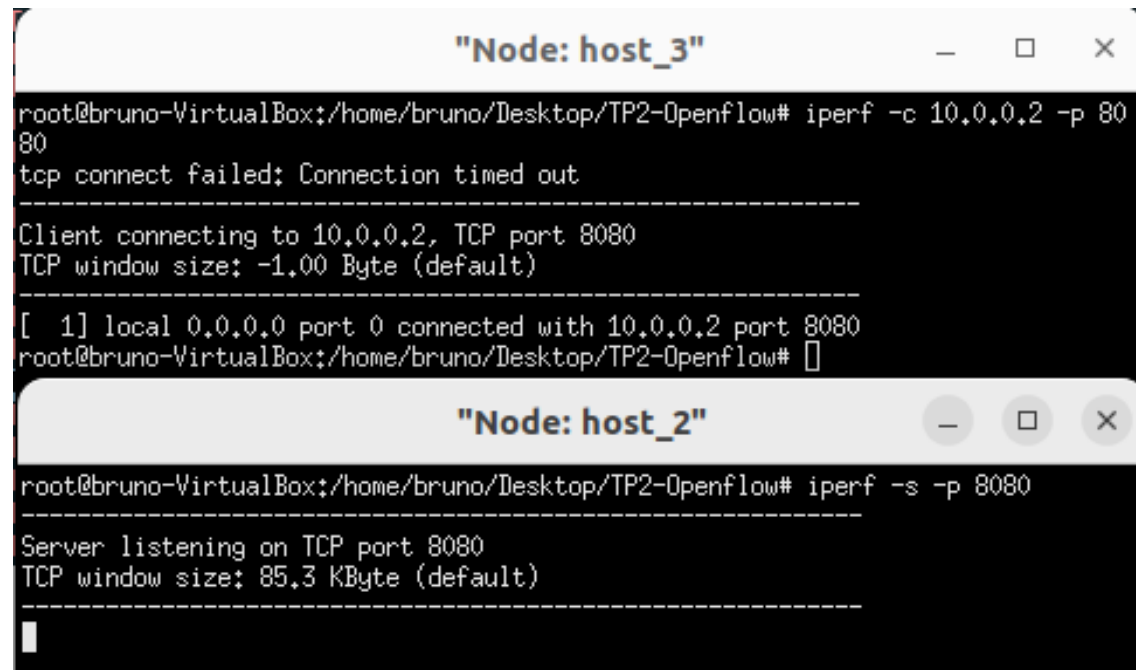
"Node: host_1"
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -c 10.0.0.3 -p 5001
Client connecting to 10.0.0.3, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 1] local 10.0.0.1 port 54658 connected with 10.0.0.3 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-10.0111 sec 3.99 GBytes 3.42 Gbits/sec
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow#

```

Figura 10: Por otro lado, si ahora cambiamos el protocolo de comunicación a TCP, veremos que el paquete proveniente del host1 con puerto destino 5001 en el servidor host3, llega sano y salvo. Esto se debe a que ya no cumple uno de los criterios necesarios para ser filtrado en el switch firewall.

5.3. Bloqueo de paquetes entre dos hosts elegidos arbitrariamente

Finalmente la tercera regla contenida en el enunciado requiere que podamos bloquear completamente la comunicación entre dos hosts, cualesquiera sean. Para esto decidimos bloquear la comunicación entre el host2 y el host3, los mismos actuando como servidor y cliente respectivamente por *iperf*.



```
"Node: host_3"
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -c 10.0.0.2 -p 8080
tcp connect failed: Connection timed out
-----
Client connecting to 10.0.0.2, TCP port 8080
TCP window size: -1.00 Byte (default)
-----
[ 1] local 0.0.0.0 port 0 connected with 10.0.0.2 port 8080
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow#

"Node: host_2"
root@bruno-VirtualBox:/home/bruno/Desktop/TP2-Openflow# iperf -s -p 8080
-----
Server listening on TCP port 8080
TCP window size: 85.3 KByte (default)
-----
```

Figura 11: Al utilizar iperf para configurar la situación descrita, vemos que nunca se logra establecer la conexión TCP entre el host2 y el host3, verificando de esta forma que nuestra regla de firewall está en funcionamiento.

tcp.stream eq 0									
No.	Time	Source	Destination	Protocol	Length Info				
213	385.356670723	10.0.0.3	10.0.0.2	TCP	74	49066 → 8080 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSval=3115			
214	385.356665734	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
215	385.353758401	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
216	386.370818784	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
217	386.371290906	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
218	386.371294513	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
219	388.383186965	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
220	388.383205976	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
221	388.383209482	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
222	392.449650324	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
223	392.449762443	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
224	392.449758076	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
225	400.639173607	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
226	400.639143575	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
227	400.639178806	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
228	416.770190352	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
229	416.770186415	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
230	416.767728405	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
238	450.562054996	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
239	450.559353522	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			
240	450.562059384	10.0.0.3	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 49066 → 8080 [SYN] Seq=0			

Figura 12: Acompañando al mensaje de timeout de iperf, reafirmamos ese comportamiento viendo todos los reintentos de handshake en la captura de wireshark a utilizar host2 como servidor y host3 como cliente.

Adicionalmente utilizamos el comando *pingall* dentro de mininet y observamos que no logran encontrarse entre sí los hosts especificados en la regla.

```
mininet> pingall
*** Ping: testing ping reachability
host_1 -> host_2 host_3 host_4
host_2 -> host_1 X host_4
host_3 -> host_1 X host_4
host_4 -> host_1 host_2 host_3
*** Results: 16% dropped (10/12 received)
```

Figura 13: Al ejecutar pingall en mininet, podemos observar que los únicos intentos de ping que hicieron timeout fueron los de host2->host3, y en sentido contrario, de host3->host2.

ip.src == 10.0.0.2									
No.	Time	Source	Destination	Protocol	Length Info				
15	4.223231515	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply	id=0xe48b, seq=1/256, ttl=64	(request in 14)	
22	4.221258379	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply	id=0xe48b, seq=1/256, ttl=64	(request in 21)	
46	4.392470394	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) request	id=0xf681, seq=1/256, ttl=64	(reply in 47)	
48	4.284669623	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) request	id=0xf681, seq=1/256, ttl=64	(reply in 49)	
50	4.314916152	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) request	id=0x9d86, seq=1/256, ttl=64	(no response found!)	
74	14.326130723	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) request	id=0xf679, seq=1/256, ttl=64	(reply in 75)	
76	14.332231419	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) request	id=0xf679, seq=1/256, ttl=64	(reply in 77)	
78	14.323514103	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) request	id=0xf679, seq=1/256, ttl=64	(reply in 79)	
80	14.326135413	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) request	id=0xf679, seq=1/256, ttl=64	(reply in 81)	
94	24.468993483	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) reply	id=0x0258, seq=1/256, ttl=64	(request in 93)	
100	24.471367783	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) reply	id=0x0258, seq=1/256, ttl=64	(request in 99)	
113	24.473716676	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) reply	id=0x0258, seq=1/256, ttl=64	(request in 112)	
116	24.471363443	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) reply	id=0x0258, seq=1/256, ttl=64		

Figura 14: Analizando en wireshark los paquetes emitidos como causa de este *pingall*, vemos que aquellos que tenían como IP de origen la de host2, nunca recibieron respuesta del host3 al hacer ping.

ip.src == 10.0.0.3									
No.	Time	Source	Destination	Protocol	Length Info				
26	4.248417693	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x9914, seq=1/256, ttl=64	(request in 25)	
31	4.251134631	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x9914, seq=1/256, ttl=64	(request in 30)	
39	4.251123658	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x9914, seq=1/256, ttl=64	(request in 38)	
43	4.252779465	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply	id=0x9914, seq=1/256, ttl=64	(request in 42)	
82	14.345586693	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) request	id=0x0813, seq=1/256, ttl=64	(reply in 83)	
84	14.408062630	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request	id=0x95ab, seq=1/256, ttl=64	(no response found!)	
85	14.354596045	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) request	id=0x0813, seq=1/256, ttl=64	(reply in 86)	
87	14.342101957	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) request	id=0x0813, seq=1/256, ttl=64	(reply in 88)	
89	14.405334866	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request	id=0x95ab, seq=1/256, ttl=64	(no response found!)	
90	14.345591594	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) request	id=0x0813, seq=1/256, ttl=64	(reply in 91)	
92	14.408069214	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request	id=0x95ab, seq=1/256, ttl=64	(no response found!)	
101	24.410287234	10.0.0.3	10.0.0.4	ICMP	98	Echo (ping) request	id=0x9a4a, seq=1/256, ttl=64	(reply in 102)	
104	24.485045874	10.0.0.3	10.0.0.4	ICMP	98	Echo (ping) reply	id=0x6181, seq=1/256, ttl=64	(request in 103)	
108	24.413974290	10.0.0.3	10.0.0.4	ICMP	98	Echo (ping) request	id=0x9a4a, seq=1/256, ttl=64	(reply in 109)	
115	24.489342279	10.0.0.3	10.0.0.4	ICMP	98	Echo (ping) reply	id=0x6181, seq=1/256, ttl=64	(request in 114)	

Figura 15: Del lado del host3, vemos en Wireshark que cuando fue el emisor de los mensajes ICMP de ping, nunca recibió respuesta de host2.

6. Pruebas Adicionales

Adicionalmente a las pruebas de *iperf* verificamos el funcionamiento con *nc* para tener pruebas más interactivas al leer por terminal. El switch configurado como firewall es el primero en la cadena.

6.1. Bloqueo a cualquier host con puerto destino 80

En el caso de la primera regla, testearmos la comunicación entre los hosts 2 y 4 (que se encuentran en extremos opuestos de la topología) utilizando el comando *nc* dentro de sus respectivas *xterms*. Probaremos con TCP.

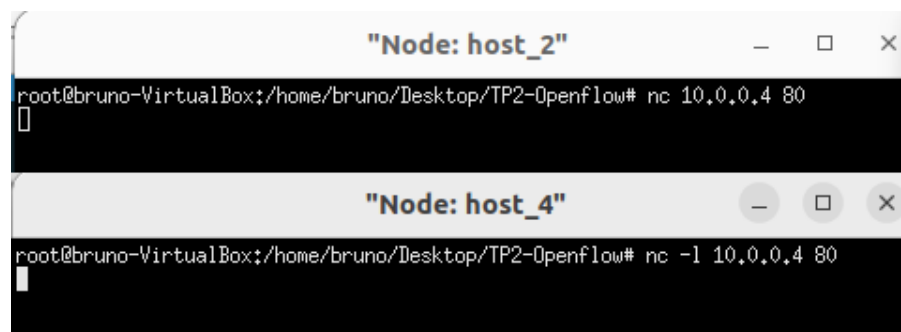


Figura 16: Enviando con *nc* un mensaje con puerto destino 80 desde el host2 al host4 mediante TCP. Podemos ver que del lado del listener en el host4, no se recibe el paquete.

tcp.stream eq 0						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.2	10.0.0.4	TCP	74	42292 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1460
2	1.007791051	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused]
3	3.020406545	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused]
4	7.178922674	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused]
5	15.376435237	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused]
6	31.499541900	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused]
13	64.267372267	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission] [TCP Port numbers reused]

Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface switch 1-eth2, id 1
Ethernet II, Src: 00:00:00:00:00:02 (00:00:00:00:00:02), Dst: 00:00:00:00:00:04 (00:00:00:00:00:04)
Internet Protocol Version 4, Src: 10.0.0.2, Dst: 10.0.0.4
Transmission Control Protocol, Src Port: 42292, Dst Port: 80, Seq: 0, Len: 0

Figura 17: Observación de los paquetes enviados en wireshark. Vemos que aparece un SYN reintentado varias veces, debido a que no se puede establecer el handshake.

6.2. Bloqueo de paquetes UDP provenientes del host1 con puerto destino 5001

Para la segunda prueba, testeamos la comunicación nuevamente entre los hosts 1 y 4 mediante UDP con puerto destino 5001.

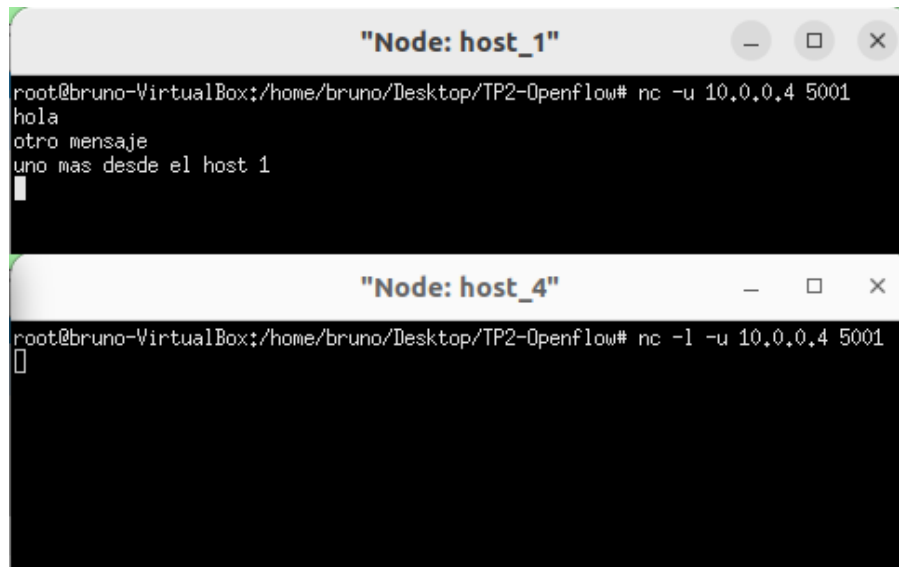


Figura 18: Enviando con nc un mensaje udp con puerto destino 5001 desde el host1 al host4. Podemos ver que del lado del listener en el host4, no se recibe el paquete.

udp.stream eq 0					
No.	Time	Source	Destination	Protocol	Length Info
1	0.000000000	10.0.0.1	10.0.0.4	UDP	47 37426 → 5001 Len=5
50	125.702485246	10.0.0.1	10.0.0.4	UDP	55 37426 → 5001 Len=13
51	134.151149562	10.0.0.1	10.0.0.4	UDP	66 37426 → 5001 Len=24

Frame 1: 47 bytes on wire (376 bits), 47 bytes captured (376 bits) on interface switch_1-eth1, id 0 Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:04 (00:00:00:00:00:04) Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.4 User Datagram Protocol, Src Port: 37426, Dst Port: 5001 Data (5 bytes)					
0000	00 00 00 00 00 04 00 00	00 00 00 01 08 00 45 00E		
0010	00 21 ef 40 40 00 40 11	37 87 0a 00 00 01 0a 00	..1.00.0.7.....		
0020	00 04 92 32 13 89 00 0d	14 23 68 6f 6c 61 0a	...2...#hola		

Figura 19: Observación de los paquetes enviados en wireshark.

7. Preguntas a responder

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

Lo que tienen en común los *switches* y los *routers* es que permiten conectar distintos dispositivos entre sí.

La diferencia principal entre ambos es que un *switch* permite conectar dispositivos dentro de una misma red local, por ejemplo conectar una computadora con una impresora dentro de la misma red, mientras que un *router* permite conectar múltiples *switches* y sus respectivas redes para formar una red mas grande. Dicho de otra forma, permite a un dispositivo conectarse con el resto del internet, otras redes. Por ejemplo un conjunto de *routers* permite conectar una computadora con un servidor.

Otras cuestiones que los diferencia es que los *routers* proveen un aislamiento más robusto del tráfico, mejor control de *broadcasts*, y un uso más *inteligente* de las rutas entre los *hosts* de la red pero necesitan mayor configuración y tienen más tiempo de procesamiento. Los *switches* en cambio son del estilo *plug-and-play* y tienen un mejor rendimiento de *performance* en redes con pocos *hosts*.

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

La diferencia entre un *switch* convencional y un *switch OpenFlow* es que un *switch OpenFlow* se comunica a través de un canal de *OpenFlow* a un controlador externo para realizar diferentes acciones especificadas en tablas de flujo. Estas acciones pueden ser inspección de paquetes o reenvío. De esta forma se esta separando en plano de datos y plano de control. En el *switch* convencional el plano de datos y el de control están en el mismo dispositivo.

7.3. ¿Se pueden reemplazar todos los routers de la Internet por Switches OpenFlow?

Piense en el escenario *interASes* para elaborar su respuesta

No se pueden reemplazar por varios motivos:

- Los *routers* de borde están diseñados para la respuesta rápida en base a la IP. *Openflow* no trabaja con IPs, sino con forwardo en base al flujo. Lo cual provocaría tablas gigantes a consultar.
- No se puede evitar que un tercero dirija su propia ruta ya que *Openflow* cuenta con un controlador centralizado permitiendo el ataque de “man-in-the-middle”.

Respecto al escenario *interASes*:

- Los *routers* que permiten la comunicación implementan el protocolo BGP. Por lo que se necesitaría tener una enorme cantidad de entradas BGP almacenadas en el dispositivo.
- El control centralizado que proponen los *switches* OpenFlow sirve para redes triviales, no escala como se necesita para Internet. Si bien, es teóricamente posible, no es para nada útil en la práctica.

8. Dificultades encontradas

Durante la realización del trabajo práctico nos encontramos con las siguientes dificultades:

- Se tuvieron dificultades en el uso de la herramienta *pox* y *mininet* en su instalación.

9. Conclusiones

Se llegaron a las siguientes conclusiones.

- Se logro adquirir un mayor entendimiento sobre el funcionamiento y desafíos que presentan las SDNs.
- Se logro observar a través de *wireshark* como es la transmisión de los paquetes por las diferentes interfaces, y así tener un mayor conocimiento de esta herramienta.

10. Referencias

- [Documentacion de mininet](#)
- [Documentacion de pox](#)
- [Visualizador de topologias](#)
- [Openflow](#)
- [Open vSwitch](#)