



Trabajo Práctico N°2

Software-Defined Networks

[75.43/95.60] Introducción a los Sistemas Distribuidos
Segundo cuatrimestre 2022
Grupo 7

Alumno	Padrón
Joaquin Hojman	102264
Facundo Mastricchio	100874
Santiago Tissoni	103856
Francisco Vazquez	104128
Andrés Zambrano	105500

Índice

1. Introducción	2
2. Preguntas a responder	2
3. Hipótesis y supuestos realizados	3
4. Código y configuración de reglas	3
5. Pruebas	5
5.1. pingall	6
5.2. Firewall	7
5.2.1. Conexión exitosa	10
5.2.1.1. TCP	10
5.2.1.2. UDP	11
5.2.2. Bloqueo de puerto de destino 80	13
5.2.2.1. TCP	13
5.2.2.2. UDP	15
5.2.3. Bloqueo de puerto 5001, UDP proveniente del host 1	17
5.2.4. Bloqueo de MAC entre host 1 y host 3	21
5.2.4.1. TCP - host 1 a host 3	21
5.2.4.2. UDP - host 1 a host 3	23
5.2.4.3. TCP - host 3 a host 1	25
5.2.4.4. UDP - host 3 a host 1	27
5.2.5. Conexión sin pasar por firewall	29
6. Dificultades encontradas	32
7. Conclusiones	32

1. Introducción

El objetivo del trabajo será construir una topología dinámica, donde se pedirá utilizar OpenFlow para poder implementar un Firewall a nivel de capa de enlace. Para poder plantear este escenario se emulará el comportamiento de la topología a través de *Mininet*. El trabajo cuenta con múltiples pasos y requisitos.

La topología que buscaremos armar será como la siguiente:

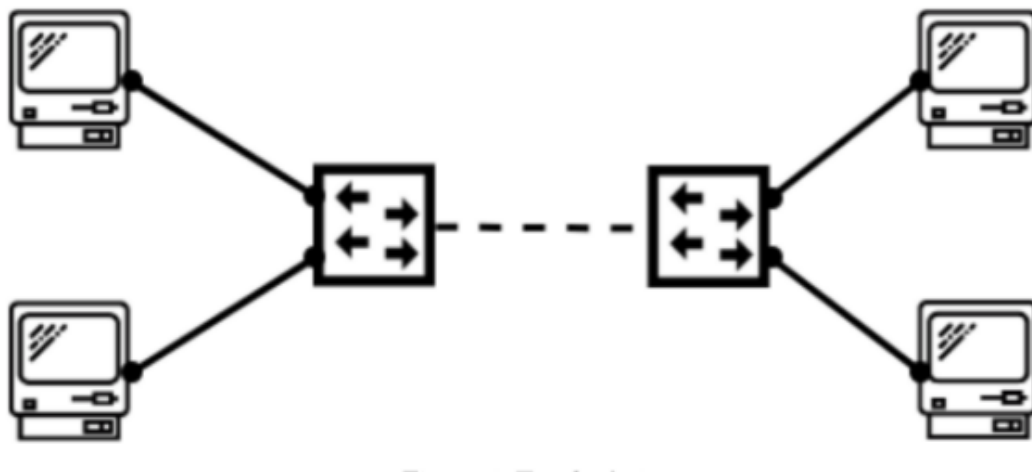


Figura 1: Topología buscada, 4 hosts y N switches

2. Preguntas a responder

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

Un switch trabaja en la capa de enlace, mientras que un router lo hace en la capa de red, tanto en el plano de datos como el de control. Además, un router puede conectar dispositivos de diferentes redes (conmuta paquetes a partir de la dirección IP), mientras que un switch solo puede conectar dispositivos de la misma red (conmuta paquetes a partir de la dirección MAC).

Un switch no necesita configuración para comenzar a funcionar. Para prevenir que se retransmitan paquetes infinitamente, una red de switches no puede tener ciclos. Como las direcciones MAC no tienen ningún tipo de jerarquía, para cada una de ellas se necesita guardar una entrada en sus tablas.

Un router necesita que se le asigne una dirección IP por cada interfaz (y también a los hosts a los que está conectado) para poder funcionar. Como funciona con direcciones IP que siguen una estructura jerárquica, y además utiliza la información del header IP para evitar retransmisiones infinitas, no necesita que su red no tenga ciclos.

Por otro lado, ambos dispositivos tienen en común que son capaces de enrutar (o conmutar) paquetes de datos, ofrecen conectividad entre dispositivos, y pueden determinar por donde tiene que salir un paquete en función de sus direcciones de origen y de destino.

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 en un switch convencional, las funcionalidades de la capa de datos y de la capa de control se realizan en el mismo dispositivo.

Sin embargo, en los switches OpenFlow el plano de control se realiza en un controlador externo, mientras que el plano de datos se realiza en el switch, y ambos se comunican por medio del protocolo OpenFlow. Esta metodología, conocida como SDN permite implementar políticas de retransmisión basadas en los flujos de manera más generalizada, utilizando información de las capas superiores, de red y transporte.

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

Si para responder nos basamos en nuestro conocimiento sobre el escenario interASes, podemos decir que los routers que permiten la comunicación tienen la obligación de implementar el protocolo BGP, por ende se necesitaría que cada dispositivo tenga una enorme cantidad de entradas BGP almacenadas.

Luego el control centralizado que proponen los switches OpenFlow sirve para redes triviales, lo cual no nos sirve pensando en la masividad de internet, por ende decimos que si bien sería 'posible' es poco probable que se pueda llevar a cabo en la realidad.

3. Hipótesis y supuestos realizados

Para el presente trabajo práctico se plantearon las siguientes hipótesis y supuestos:

- Suponemos que si el usuario desea definir nuevas reglas, las mismas serán similares a las ya utilizadas para la realización del trabajo práctico, esto a fin de no tener que realizar cambios en el código para que estas reglas nuevas puedan activarse.
- El programa fue testeado en ambientes con distribuciones Linux. No contemplamos que un usuario desee usarlo en Windows o macOS y no podemos garantizar que funcionara.
- Asumimos que aquel que desee utilizar nuestro programa, habrá instalado previamente las dependencias (ver sección Pruebas).

4. Código y configuración de reglas

Los pasos seguidos para armar topología de la red fueron los siguientes, utilizando el paquete *mininet* de Python:

1. Se reciben por parámetro la cantidad de switches que vamos a tener.
2. Se agregan 4 hosts a la topología.
3. Se agregan los N switches a la topología. En este punto todavía no conectamos nada.
4. Conectamos únicamente los switches entre sí, en formato de "cadena" o "hilera".
5. Luego conectamos los dos hosts de "la izquierda" al primero switch, posteriormente conectamos los dos hosts de "la derecha" al último switch.
6. De esta manera la topología queda armada y lista para usarse.

El código puede encontrarse en el directorio *src/*, con el nombre de *topology.py*.

Luego programamos el firewall en el archivo *firewall.py* (el cual se encuentra en la raíz del proyecto) y funciona de la siguiente manera: se reciben dos parámetros, el archivo de reglas y el id del switch donde estará ubicado el firewall. Luego de determinar esto, en el switch del firewall se configurarán todas las reglas del archivo de reglas.

Nuestro archivo de reglas está presentado como un JSON donde está ubicado un array de reglas, donde cada elemento del array es un diccionario que tiene campos que nos indican qué

se debe bloquear. Por ejemplo, si un elemento tiene como claves "protocol" y "dst_port", deben bloquearse los mensajes del protocolo especificado que vayan al puerto especificado. Si en cambio dijera "src_port", serán los mensajes que salgan de ese puerto, independientemente del protocolo de transporte. Es decir, las reglas pueden tener uno o más elementos a bloquear por el firewall.

- *src_ip* bloquea los paquetes que provengan de esa dirección IP.
- *dst_ip* bloquea los paquetes que vayan a esa dirección IP.
- *src_port* bloquea los paquetes que provengan de ese puerto.
- *dst_port* bloquea los paquetes que vayan a ese puerto.
- *src_mac* bloquea los paquetes que provengan de esa dirección MAC.
- *dst_mac* bloquea los paquetes que vayan a esa dirección MAC.
- *protocol* bloquea los paquetes que usen ese protocolo.

Para implementar esto se utiliza la librería POX, que incluye OpenFlow. En particular, se hizo uso de la función `_handle_ConnectionUp` donde cada switch es configurado en el momento que se conecta a la red. Además, se utilizaron los módulos de `forwarding.l2_learning` (los switches aprenden automáticamente la topología) y `openflow.of_01`. Al mismo tiempo, se decidió comentar (es decir, hacer un simple "pass") la función `_handle_PacketIn` de manera de poder tomar las capturas de Wireshark sin ver los paquetes provenientes de los eventos `PacketIn` y `PacketOut`.

Cabe destacar que por la forma en que está diseñado el código, bloquear algo para una regla no implica que ese bloqueo persista para las otras reglas, es decir si por ejemplo bloqueamos `src_port 80` para UDP, no quiero decir que se bloquee también para TCP, a menos que también lo especifiquemos en otra regla.

Nuestro archivo de reglas para cumplir las especificaciones del trabajo práctico es el siguiente:



```
{
  "rules": [
    {
      "protocol": "tcp",
      "dst_port": 80
    },
    {
      "protocol": "udp",
      "dst_port": 80
    },
    {
      "protocol": "udp",
      "dst_port": 5001,
      "src_ip": "10.0.0.1"
    },
    {
      "src_mac": "00:00:00:00:00:01",
      "dst_mac": "00:00:00:00:00:03"
    },
    {
      "src_mac": "00:00:00:00:00:03",
      "dst_mac": "00:00:00:00:00:01"
    }
  ]
}
```

Figura 2: Formato del archivo de reglas

Las primeras dos reglas permiten al firewall cumplir con el punto 1 del trabajo, que implica que se deben descartar todos los mensajes cuyo puerto destino sea 80.

La tercera regla genera el cumplimiento del segundo punto, el 2, del trabajo, descartar todos los mensajes que provengan del host 1, tengan como puerto destino el 5001, y estén utilizando el protocolo UDP.

Por último, las reglas 4 y 5 permiten implementar el punto 3 del trabajo. El mismo dice que se debe elegir dos hosts cualquiera, y los mismos no deben poder comunicarse de ninguna forma. En nuestro caso las reglas indican que los hosts que no podrán comunicarse son los hosts 1 y 3, pero eso fácilmente se puede cambiar modificando las direcciones MAC.

Téngase presente que para evitar que dos hosts cualesquiera no puedan comunicarse entre sí, deben usarse dos reglas, una donde uno de los hosts es el origen y la otra donde es el destino. Esto nos brinda el beneficio y la posibilidad de, usando una sola regla, bloquear una comunicación de forma unidireccional.

5. Pruebas

En el repositorio se adjuntan diversos archivos *.sh* que permiten correr el programa sin tener que preocuparse por las especificaciones propias de la implementación. En el *README* se encuentra como correr cada y que parámetros reciben. Se deberán levantar dos terminales: una para la topología y una para el firewall.

Es menester instalar las dependencias especificadas en el archivo *requirements.txt*, puede usarse el comando:

```
> pip install -r requirements.txt
```

Por otro lado, se debe dar permiso de ejecución a cada uno de los scripts dentro del directorio de *scripts/*

```
> chmod +x ./scripts/*
```

5.1. pingall

Para la ejecución de este ejercicio, se utilizó una topología de 2 switches, capturando con Wireshark por la interfaz switch_1-eth1 del primer switch:

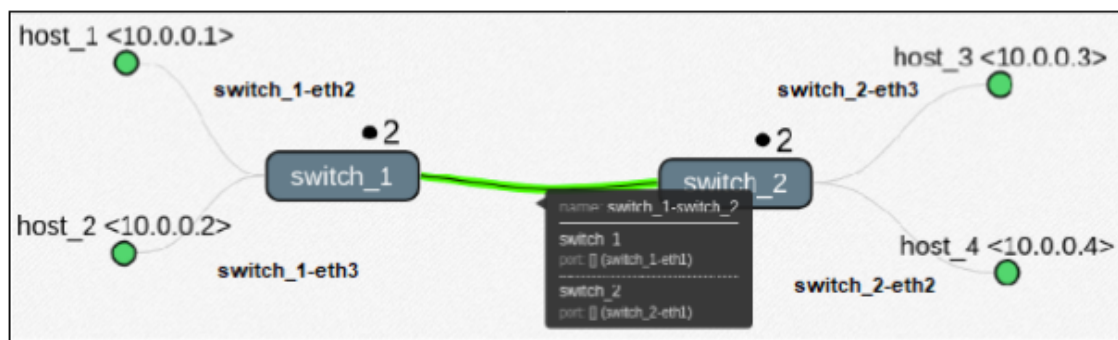


Figura 3: Topología de prueba con 2 switches

Levantamos el controlador sin el firewall:

```
> ./scripts/pox.sh
```

Levantamos la topología con 2 switches:

```
> ./scripts/topo.sh 2
```

```

mininet> pingall
*** Ping: testing ping reachability
host_1 -> host_2 host_3 host_4
host_2 -> host_1 host_3 host_4
host_3 -> host_1 host_2 host_4
host_4 -> host_1 host_2 host_3
*** Results: 0% dropped (12/12 received)

```

Figura 4: Output de pingall en Mininet

Como no había ningún firewall activado, no se pierde ningún paquete.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=8xae7, seq=1/256, ttl=64 (reply in 2)
2	0.003243317	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=8xae7, seq=1/256, ttl=64 (request in 1)
3	0.009691321	10.0.0.1	10.0.0.4	ICMP	98	Echo (ping) request id=8x85a6, seq=1/256, ttl=64 (reply in 4)
4	0.012556650	10.0.0.4	10.0.0.1	ICMP	98	Echo (ping) reply id=8x85a6, seq=1/256, ttl=64 (request in 3)
5	0.020328297	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) request id=8xf0d7, seq=1/256, ttl=64 (reply in 6)
6	0.020844005	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) reply id=8xf0d7, seq=1/256, ttl=64 (request in 5)
7	0.038728943	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) request id=8xd6ad, seq=1/256, ttl=64 (reply in 8)
8	0.039478715	10.0.0.4	10.0.0.2	ICMP	98	Echo (ping) reply id=8xd6ad, seq=1/256, ttl=64 (request in 7)
9	0.045254299	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) request id=8xf274, seq=1/256, ttl=64 (reply in 10)
10	0.047483518	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) reply id=8xf274, seq=1/256, ttl=64 (request in 9)
11	0.053080903	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=8x556a, seq=1/256, ttl=64 (reply in 12)
12	0.056278496	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) reply id=8x556a, seq=1/256, ttl=64 (request in 11)
13	0.067471195	10.0.0.4	10.0.0.1	ICMP	98	Echo (ping) request id=8x6ecf, seq=1/256, ttl=64 (reply in 14)
14	0.07338197	10.0.0.1	10.0.0.4	ICMP	98	Echo (ping) reply id=8x6ecf, seq=1/256, ttl=64 (request in 13)
15	0.076465924	10.0.0.4	10.0.0.2	ICMP	98	Echo (ping) request id=8x0060, seq=1/256, ttl=64 (reply in 16)
16	0.079121601	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) reply id=8x0060, seq=1/256, ttl=64 (request in 15)

Figura 5: Captura de Wireshark (pingall) sobre switch_1-eth1

La captura de Wireshark en la interfaz switch_1-eth1 nos permite visualizar los paquetes enviados del host 1 y 2 al 3 y 4, y viceversa. Los paquetes que van del host 1 al 2, del 2 al 1, del 3 al 4 o del 4 al 3 no pasan por esta interfaz, y por lo tanto no lo podemos capturar.

Cada uno de los 2 hosts de los 2 lados le envía un paquete a los 2 del otro lado, quienes a su vez responden. Esto resulta en los 16 paquetes que podemos visualizar en la captura.

5.2. Firewall

Usando la misma topología que en el inciso anterior, y aplicando el firewall en el primer switch, tenemos lo siguiente:

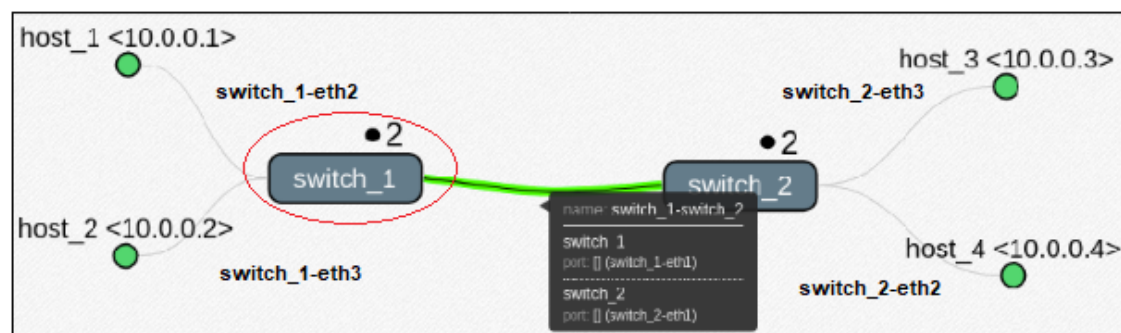
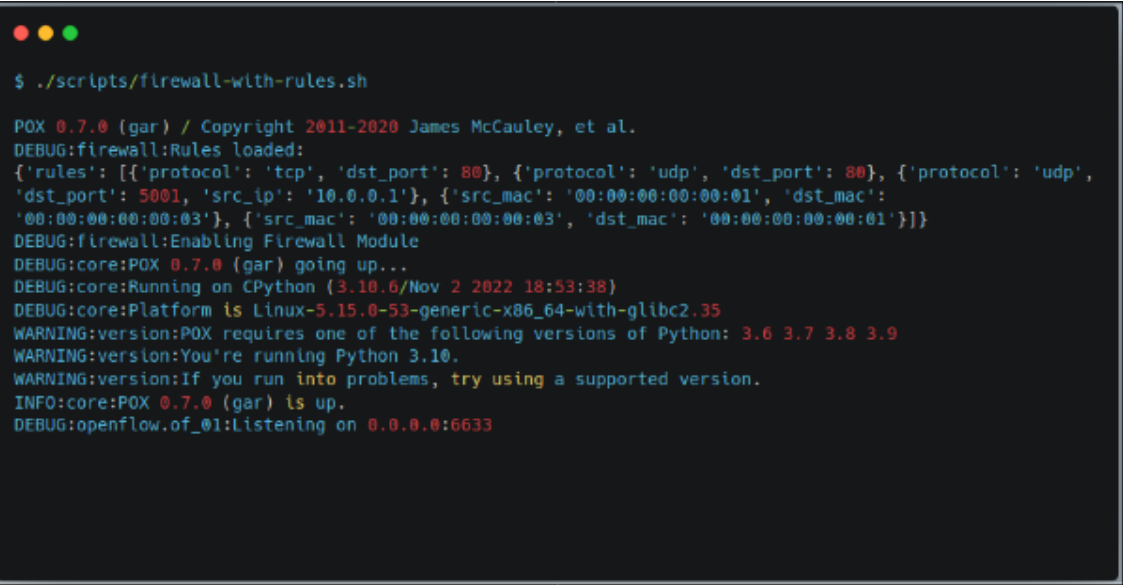


Figura 6: Topología con Firewall en switch 1

Para comprobar el correcto funcionamiento del firewall se hizo uso de *iperf* sobre mininet para enviar paquetes sobre la red.

Primero, se comienza corriendo el **firewall** en el **primer switch** en una consola:

```
> ./scripts/firewall-with-rules.sh 1
```



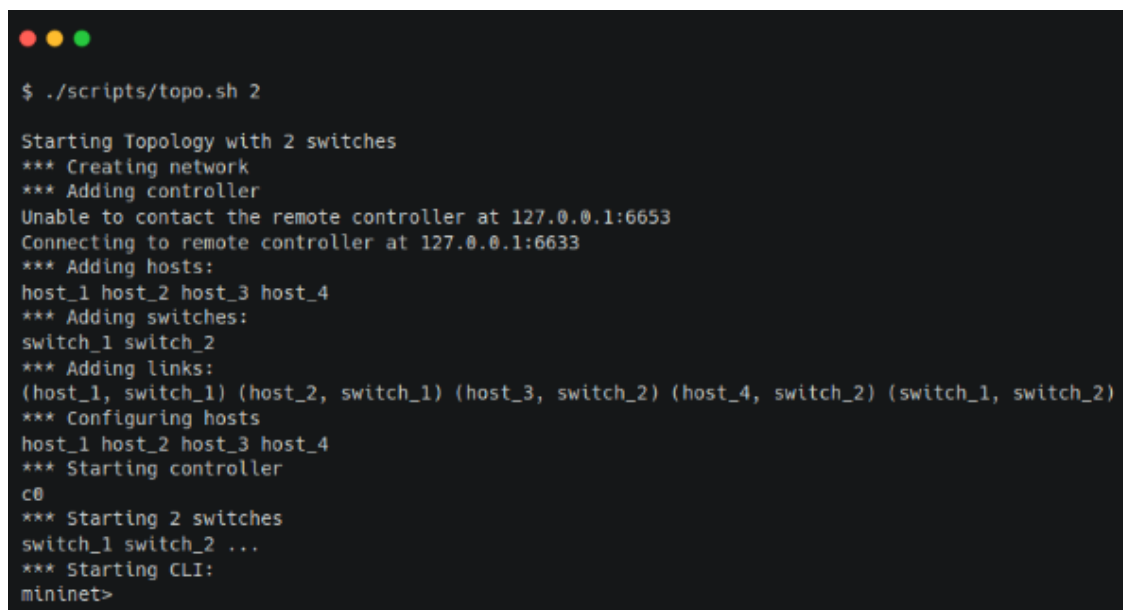
```
$ ./scripts/firewall-with-rules.sh

POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
DEBUG:firewall:Rules loaded:
{'rules': [{'protocol': 'tcp', 'dst_port': 80}, {'protocol': 'udp', 'dst_port': 80}, {'protocol': 'udp', 'dst_port': 5001, 'src_ip': '10.0.0.1'}, {'src_mac': '00:00:00:00:00:01', 'dst_mac': '00:00:00:00:00:03'}, {'src_mac': '00:00:00:00:00:03', 'dst_mac': '00:00:00:00:00:01'}]}
DEBUG:firewall:Enabling Firewall Module
DEBUG:core:POX 0.7.0 (gar) going up...
DEBUG:core:Running on CPython (3.10.6/Nov 2 2022 18:53:38)
DEBUG:core:Platform is Linux-5.15.0-53-generic-x86_64-with-glibc2.35
WARNING:version:POX requires one of the following versions of Python: 3.6 3.7 3.8 3.9
WARNING:version:You're running Python 3.10.
WARNING:version:If you run into problems, try using a supported version.
INFO:core:POX 0.7.0 (gar) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
```

Figura 7: Output de controlador con firewall

En una terminal aparte, levantamos la topología con 2 switches:

```
> ./scripts/topo.sh 2
```

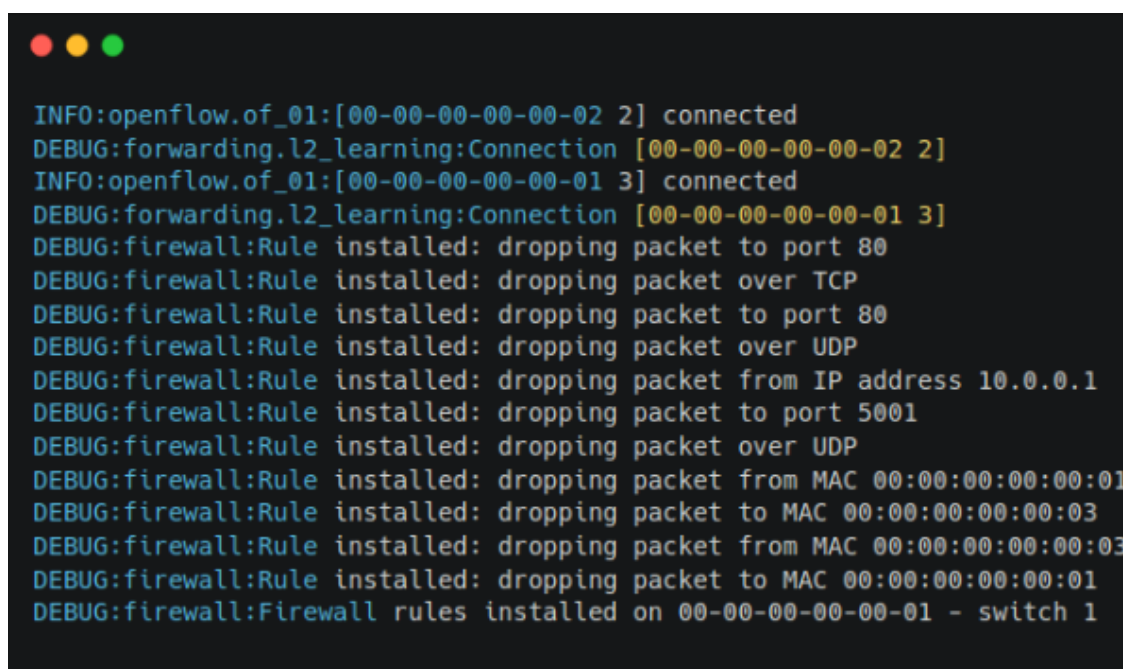
A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays the output of the command `./scripts/topo.sh 2`. The output shows the steps to create a Mininet topology with 2 switches, including creating the network, adding a controller, adding hosts (host_1 to host_4), adding switches (switch_1, switch_2), adding links, configuring hosts, starting the controller, and starting the switches and CLI. The prompt `mininet>` is visible at the bottom.

```
$ ./scripts/topo.sh 2

Starting Topology with 2 switches
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
host_1 host_2 host_3 host_4
*** Adding switches:
switch_1 switch_2
*** Adding links:
(host_1, switch_1) (host_2, switch_1) (host_3, switch_2) (host_4, switch_2) (switch_1, switch_2)
*** Configuring hosts
host_1 host_2 host_3 host_4
*** Starting controller
c0
*** Starting 2 switches
switch_1 switch_2 ...
*** Starting CLI:
mininet>
```

Figura 8: Output red creada en Mininet

Al iniciar la topología, podremos apreciar en la consola del firewall la configuración de las reglas.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays log messages from an OpenFlow controller and a firewall. The messages show the connection of OpenFlow controllers and the installation of various firewall rules on switch 1, including rules for dropping packets to port 80, over TCP, from IP address 10.0.0.1, to port 5001, over UDP, from MAC 00:00:00:00:00:01, to MAC 00:00:00:00:00:03, from MAC 00:00:00:00:00:03, and to MAC 00:00:00:00:00:01.

```
INFO:openflow.of_01:[00-00-00-00-00-02 2] connected
DEBUG:forwarding.l2_learning:Connection [00-00-00-00-00-02 2]
INFO:openflow.of_01:[00-00-00-00-00-01 3] connected
DEBUG:forwarding.l2_learning:Connection [00-00-00-00-00-01 3]
DEBUG:firewall:Rule installed: dropping packet to port 80
DEBUG:firewall:Rule installed: dropping packet over TCP
DEBUG:firewall:Rule installed: dropping packet to port 80
DEBUG:firewall:Rule installed: dropping packet over UDP
DEBUG:firewall:Rule installed: dropping packet from IP address 10.0.0.1
DEBUG:firewall:Rule installed: dropping packet to port 5001
DEBUG:firewall:Rule installed: dropping packet over UDP
DEBUG:firewall:Rule installed: dropping packet from MAC 00:00:00:00:00:01
DEBUG:firewall:Rule installed: dropping packet to MAC 00:00:00:00:00:03
DEBUG:firewall:Rule installed: dropping packet from MAC 00:00:00:00:00:03
DEBUG:firewall:Rule installed: dropping packet to MAC 00:00:00:00:00:01
DEBUG:firewall:Firewall rules installed on 00-00-00-00-00-01 - switch 1
```

Figura 9: Configuración de reglas

Para las pruebas que se presentan a continuación, la captura con Wireshark se realizó en la interfaz 1 del switch 1, switch_1-eth1. En el caso que sea necesario, se especificará sobre cuál otra interfaz se realizó la captura.

5.2.1. Conexión exitosa

5.2.1.1. TCP

Comenzamos levantando un servidor TCP en el host 2 conectado al puerto 1234 y vamos a conectarnos a él desde el host 4. Esta interacción no está prohibida por el firewall que se configuró por lo que el servidor debería recibir los paquetes.

```
> xterm host_2 host_4
```

En la terminal para el host 2 (servidor).

```
> iperf -s -p 1234 -i 1
```

```
=====
Server listening on TCP port 1234
TCP window size: 85.3 KByte (default)
=====
[  1] local 10.0.0.2 port 1234 connected with 10.0.0.4 port 45130
[ ID] Interval      Transfer    Bandwidth
[  1] 0.0000-1.0000 sec  1.53 GBytes 13.1 Gbits/sec
[  1] 1.0000-2.0000 sec  1.61 GBytes 13.8 Gbits/sec
[  1] 2.0000-3.0000 sec  1.53 GBytes 13.1 Gbits/sec
[  1] 3.0000-4.0000 sec  1.60 GBytes 13.7 Gbits/sec
[  1] 4.0000-4.4711 sec   757 MBytes 13.5 Gbits/sec
[  1] 0.0000-4.4711 sec  7.00 GBytes 13.4 Gbits/sec
=====
```

Figura 10: Output server

En la terminal para el host 4 (cliente).

```
> iperf -c 10.0.0.2 -p 1234
```

```

Client connecting to 10.0.0.2, TCP port 1234
TCP window size: 85.3 KByte (default)

[ 1] local 10.0.0.4 port 45130 connected with 10.0.0.2 port 1234
^C[ ID] Interval      Transfer      Bandwidth
[ 1] 0.0000-4.4815 sec 7.00 GBytes  13.4 Gbits/sec

```

Figura 11: Output cliente

Mientras, en Wireshark podemos ver que el host 2 recibe paquetes del host 4 y viceversa.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.0.4	10.0.0.2	TCP	74	41554 → 1234 [SYN] Seq=0 Win=42496 Len=0 MSS=1460 SACK_PERM=1 TSval=237799014
2	0.004441	10.0.0.2	10.0.0.4	TCP	74	1234 → 41554 [SYN, ACK] Seq=0 Ack=1 Min=43440 Len=0 MSS=1460 SACK_PERM=1 TSva
3	0.007176	10.0.0.4	10.0.0.2	TCP	66	41554 → 1234 [ACK] Seq=1 Ack=1 Min=42496 Len=0 TSval=237799023 TSecr=10447913
4	0.007405	10.0.0.4	10.0.0.2	TCP	126	41554 → 1234 [PSH, ACK] Seq=1 Ack=1 Min=42496 Len=60 TSval=237799024 TSecr=10
5	0.007526	10.0.0.2	10.0.0.4	TCP	66	1234 → 41554 [ACK] Seq=1 Ack=61 Min=43520 Len=0 TSval=1044791342 TSecr=237799
6	0.007610	10.0.0.4	10.0.0.2	TCP	2962	41554 → 1234 [PSH, ACK] Seq=61 Ack=1 Min=42496 Len=2896 TSval=237799024 TSecr
7	0.007627	10.0.0.2	10.0.0.4	TCP	66	1234 → 41554 [ACK] Seq=1 Ack=2957 Min=40960 Len=0 TSval=1044791342 TSecr=2377
8	0.007653	10.0.0.4	10.0.0.2	TCP	2962	41554 → 1234 [PSH, ACK] Seq=2957 Ack=1 Min=42496 Len=2896 TSval=237799024 TSec
9	0.007671	10.0.0.4	10.0.0.2	TCP	2962	41554 → 1234 [PSH, ACK] Seq=5853 Ack=1 Min=42496 Len=2896 TSval=237799024 TSec
10	0.007684	10.0.0.2	10.0.0.4	TCP	94	1234 → 41554 [PSH, ACK] Seq=1 Ack=2957 Min=40960 Len=28 TSval=1044791342 TSec
11	0.007689	10.0.0.4	10.0.0.2	TCP	2962	41554 → 1234 [PSH, ACK] Seq=8749 Ack=1 Min=42496 Len=2896 TSval=237799024 TSec
12	0.007705	10.0.0.4	10.0.0.2	TCP	1514	41554 → 1234 [ACK] Seq=11645 Ack=1 Min=42496 Len=1448 TSval=237799024 TSecr=10
13	0.008001	10.0.0.2	10.0.0.4	TCP	66	1234 → 41554 [ACK] Seq=29 Ack=13093 Win=42496 Len=0 TSval=1044791342 TSecr=23
14	0.009905	10.0.0.4	10.0.0.2	TCP	2962	41554 → 1234 [PSH, ACK] Seq=13093 Ack=1 Min=42496 Len=2896 TSval=237799028 TSecr=23
15	0.010014	10.0.0.2	10.0.0.4	TCP	66	1234 → 41554 [ACK] Seq=39 Ack=15000 Min=43440 Len=0 TSval=1044791344 TSecr=23

* Frame 14: 2962 bytes on wire (23696 bits), 2962 bytes captured (23696 bits) on interface switch_1-eth1, id 0
 * Ethernet II, Src: 00:00:00:00:00:04 (00:00:00:00:00:04), Dst: 00:00:00:00:00:02 (00:00:00:00:00:02)
 * Internet Protocol Version 4, Src: 10.0.0.4, Dst: 10.0.0.2
 * Transmission Control Protocol, Src Port: 41554, Dst Port: 1234, Seq: 13093, Ack: 1, Len: 2896
 * Data (2896 bytes)

Figura 12: Captura Wireshark sobre switch_1-eth1

5.2.1.2. UDP

A continuación, haremos la misma prueba pero utilizando UDP.

```
> xterm host_2 host_4
```

En la terminal para el host 2 (servidor).

```
> iperf -s -p 1234 -u -i 1
```

```

-----
Server listening on UDP port 1234
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.2 port 1234 connected with 10.0.0.4 port 46074
[ ID] Interval      Transfer      Bandwidth      Jitter    Lost/Total Datagrams
[ 1] 0.0000-1.0000 sec   131 KBytes    1.07 Mbits/sec  0.006 ms  0/91 (0%)
[ 1] 1.0000-2.0000 sec   128 KBytes    1.05 Mbits/sec  0.011 ms  0/89 (0%)
[ 1] 2.0000-3.0000 sec   128 KBytes    1.05 Mbits/sec  0.002 ms  0/89 (0%)
[ 1] 3.0000-3.8871 sec   115 KBytes    1.06 Mbits/sec  0.006 ms  0/80 (0%)
[ 1] 0.0000-3.8871 sec   501 KBytes    1.06 Mbits/sec  0.006 ms  0/349 (0%)

```

Figura 13: Output server

En la terminal para el host 4 (cliente).

```
> iperf -c 10.0.0.2 -p 1234 -u
```

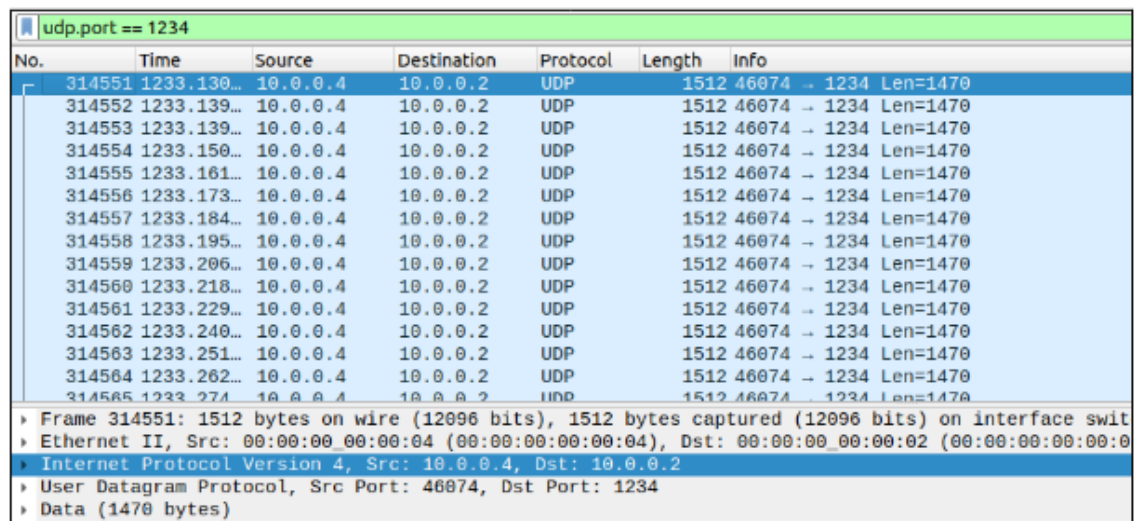
```

Client connecting to 10.0.0.2, UDP port 1234
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.4 port 46074 connected with 10.0.0.2 port 1234
^C[ ID] Interval      Transfer      Bandwidth
[ 1] 0.0000-3.8918 sec   501 KBytes    1.05 Mbits/sec
[ 1] Sent 350 datagrams
[ 1] Server Report:
[ ID] Interval      Transfer      Bandwidth      Jitter    Lost/Total Datagrams
[ 1] 0.0000-3.8871 sec   501 KBytes    1.06 Mbits/sec  0.005 ms  0/349 (0%)

```

Figura 14: Output cliente

Mientras, en Wireshark podemos ver que el host 2 recibe paquetes del host 4.



No.	Time	Source	Destination	Protocol	Length	Info
314551	1233.130...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314552	1233.139...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314553	1233.139...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314554	1233.150...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314555	1233.161...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314556	1233.173...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314557	1233.184...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314558	1233.195...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314559	1233.206...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314560	1233.218...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314561	1233.229...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314562	1233.240...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314563	1233.251...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314564	1233.262...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470
314565	1233.274...	10.0.0.4	10.0.0.2	UDP	1512	46074 → 1234 Len=1470

▶ Frame 314551: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface swit
 ▶ Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
 ▶ Internet Protocol Version 4, Src: 10.0.0.4, Dst: 10.0.0.2
 ▶ User Datagram Protocol, Src Port: 46074, Dst Port: 1234
 ▶ Data (1470 bytes)

Figura 15: Captura Wireshark sobre UDP

5.2.2. Bloqueo de puerto de destino 80

5.2.2.1. TCP

Ahora, probamos los casos en los cuales el Firewall debe bloquear el envío de paquetes destinados al puerto 80. Comenzamos por el caso de un servidor TCP en el host 2 escuchando en el puerto 80 y con un cliente en el host 4.

```
> xterm host_2 host_4
```

En la terminal para el host 2 (servidor).

```
> iperf -s -p 80 -i 1
```

```

-----
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
-----
█

```

Figura 16: Output server

En la terminal para el host 4 (cliente).

```
> iperf -c 10.0.0.2 -p 80
```

```

tcp connect failed: Connection timed out
-----
Client connecting to 10.0.0.2, TCP port 80
TCP window size: -1.00 Byte (default)
-----
[ 1] local 0.0.0.0 port 0 connected with 10.0.0.2 port 80

```

Figura 17: Output cliente

tcp.port == 80						
No.	Time	Source	Destination	Protocol	Length	Info
314908	1806.712..	10.0.0.4	10.0.0.2	TCP	74	33688 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1
314909	1807.724..	10.0.0.4	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 33688 → 80 [
314910	1809.743..	10.0.0.4	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 33688 → 80 [
314911	1813.995..	10.0.0.4	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 33688 → 80 [
314912	1822.187..	10.0.0.4	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 33688 → 80 [
314913	1838.318..	10.0.0.4	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 33688 → 80 [
314915	1871.341..	10.0.0.4	10.0.0.2	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 33688 → 80 [
<p> ▶ Frame 314909: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface switch_1-eth1, id 0 ▶ Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02) ▶ Internet Protocol Version 4, Src: 10.0.0.4, Dst: 10.0.0.2 ▶ Transmission Control Protocol, Src Port: 33688, Dst Port: 80, Seq: 0, Len: 0 </p>						

Figura 18: Captura Wireshark TCP puerto 80

Como se puede ver, en este caso el servidor 2 no recibe nada. Esto se debe a que el firewall entró en acción bloqueando los paquetes del host 4 ya que cumplen con la primera regla del firewall, y por lo tanto, el host 4 retransmite paquetes hasta llegar el timeout.

Si vemos la interfaz del switch 1 que directamente está linkeada con el host 2 vemos que efectivamente no recibe ningún paquete que tenga como destino el puerto 80 y el protocolo sea TCP.



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

Figura 19: Captura Wireshark TCP puerto 80 switch_1-eth3

5.2.2.2. UDP

Probemos el siguiente caso, puerto 80 pero con el protocolo UDP.

```
> xterm host_2 host_4
```

En la terminal para el host 2 (servidor).

```
> iperf -s -p 80 -u -i 1
```



```
-----  
Server listening on UDP port 80  
UDP buffer size: 208 KByte (default)  
-----  
[
```

Figura 20: Output server

En la terminal para el host 4 (cliente).

```
> iperf -c 10.0.0.2 -u -p 80
```

```
-----  
Client connecting to 10.0.0.2, UDP port 80  
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)  
UDP buffer size: 208 KByte (default)  
-----  
[  1] local 10.0.0.4 port 45114 connected with 10.0.0.2 port 80  
[ ID] Interval      Transfer    Bandwidth  
[  1] 0.0000-10.0153 sec 1.25 MBytes 1.05 Mbits/sec  
[  1] Sent 896 datagrams  
[  5] WARNING: did not receive ack of last datagram after 10 tries.
```

Figura 21: Output cliente

udp.port == 80						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
2	0.011335...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
3	0.011351...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
4	0.021232...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
5	0.032485...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
6	0.043709...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
7	0.054924...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
8	0.066157...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
9	0.077353...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
10	0.088549...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
11	0.099848...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
12	0.110986...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
13	0.122200...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
14	0.133456...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
15	0.144718...	10.0.0.4	10.0.0.2	UDP	1512	56694 → 80 Len=1470
▶ Frame 1: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface switch_1-eth1, ▶ Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02) ▶ Internet Protocol Version 4, Src: 10.0.0.4, Dst: 10.0.0.2 ▶ User Datagram Protocol, Src Port: 56694, Dst Port: 80 ▶ Data (1470 bytes)						

Figura 22: Captura Wireshark UDP puerto 80

En un caso similar al explicado anteriormente, el servidor 2 no recibe nada pero podemos ver que en la interfaz que proviene del lado del host 4 se capturan los paquetes UDP con puerto de destino 80. El firewall entró en acción bloqueando los paquetes del host 4 ya que cumplen con la primera regla del firewall, y por lo tanto, el host 4 retransmite paquetes hasta llegar el timeout.

Si vemos la interfaz del switch 1 que directamente está linkeada con el host 2 vemos que efectivamente no recibe ningún paquete que tenga como puerto de destino el 80 y el protocolo sea UDP.

udp.port == 80						
No.	Time	Source	Destination	Protocol	Length	Info

Figura 23: Captura Wireshark UDP puerto 80 switch_1-eth3

5.2.3. Bloqueo de puerto 5001, UDP proveniente del host 1

A continuación se hace una prueba con un servidor UDP en host 2 en el puerto 5001 con el host 4 como cliente. A priori, esta comunicación debería funcionar ya que si bien se cumple que el puerto de destino sea el 5001 y el protocolo sea UDP, el firewall está especificado para el host 1.

```
> xterm host_2 host_4
```

En la terminal para el host 2 (servidor).

```
> iperf -s -p 5001 -u -i 1
```

```

-----
Server listening on UDP port 5001
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.2 port 5001 connected with 10.0.0.4 port 33039
[ ID] Interval      Transfer      Bandwidth      Jitter    Lost/Total Datagrams
[ 1] 0.0000-1.0000 sec   131 KBytes   1.07 Mbits/sec  0.012 ms  0/91 (0%)
[ 1] 1.0000-2.0000 sec   128 KBytes   1.05 Mbits/sec  0.009 ms  0/89 (0%)
[ 1] 2.0000-3.0000 sec   128 KBytes   1.05 Mbits/sec  0.011 ms  0/89 (0%)
[ 1] 3.0000-4.0000 sec   129 KBytes   1.06 Mbits/sec  0.010 ms  0/90 (0%)
[ 1] 4.0000-5.0000 sec   128 KBytes   1.05 Mbits/sec  0.005 ms  0/89 (0%)
[ 1] 5.0000-6.0000 sec   128 KBytes   1.05 Mbits/sec  0.008 ms  0/89 (0%)
[ 1] 6.0000-7.0000 sec   128 KBytes   1.05 Mbits/sec  0.005 ms  0/89 (0%)
[ 1] 7.0000-8.0000 sec   128 KBytes   1.05 Mbits/sec  0.005 ms  0/89 (0%)
[ 1] 8.0000-9.0000 sec   128 KBytes   1.05 Mbits/sec  0.014 ms  0/89 (0%)
[ 1] 9.0000-10.0000 sec  129 KBytes   1.06 Mbits/sec  0.014 ms  0/90 (0%)
[ 1] 0.0000-10.0101 sec  1.25 MBytes  1.05 Mbits/sec  0.013 ms  0/895 (0%)

```

Figura 24: Output servidor

En la terminal para el host 4 (cliente).

```
> iperf -c 10.0.0.2 -u -p 5001
```

```

-----
Client connecting to 10.0.0.2, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.4 port 33039 connected with 10.0.0.2 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 1] 0.0000-10.0154 sec  1.25 MBytes   1.05 Mbits/sec
[ 1] Sent 896 datagrams
[ 1] Server Report:
[ ID] Interval      Transfer      Bandwidth      Jitter    Lost/Total Datagrams
[ 1] 0.0000-10.0101 sec  1.25 MBytes   1.05 Mbits/sec  0.013 ms  0/895 (0%)

```

Figura 25: Output cliente

udp.port == 5001						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
2	0.000053...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
3	0.000885...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
4	0.019023...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
5	0.031090...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
6	0.042306...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
7	0.053517...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
8	0.064717...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
9	0.075948...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
10	0.087160...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
11	0.098384...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
12	0.109600...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
13	0.120884...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
14	0.132026...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
15	0.143305...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
16	0.154508...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
17	0.165657...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
18	0.176873...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
19	0.188093...	10.0.0.4	10.0.0.2	UDP	1512	33039 → 5001 Len=1470
▶ Frame 1: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface switch_1-eth1, ▶ Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02) ▶ Internet Protocol Version 4, Src: 10.0.0.4, Dst: 10.0.0.2 ▶ User Datagram Protocol, Src Port: 33039, Dst Port: 5001 ▶ Data (1470 bytes)						

Figura 26: Captura Wireshark UDP puerto 5001

Observamos que los paquetes pasan por la interfaz switch_1-eth1. Para asegurarnos, vamos a mirar la interfaz que está linkeada con el host 2 para ver si efectivamente le llegaron los paquetes.

udp.port == 5001						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
2	0.006696...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
3	0.006743...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
4	0.017415...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
5	0.020660...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
6	0.039882...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
7	0.051242...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
8	0.062395...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
9	0.073551...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
10	0.084739...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
11	0.096038...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
12	0.107262...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
13	0.118524...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
14	0.129680...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
15	0.140906...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
16	0.152172...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
17	0.163386...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
18	0.174553...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
19	0.185816...	10.0.0.4	10.0.0.2	UDP	1512	58653 → 5001 Len=1470
▶ Frame 1: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface switch_1-eth3, ▶ Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02) ▶ Internet Protocol Version 4, Src: 10.0.0.4, Dst: 10.0.0.2 ▶ User Datagram Protocol, Src Port: 58653, Dst Port: 5001 ▶ Data (1470 bytes)						

Figura 27: Captura Wireshark UDP puerto 5001 switch_1-eth3

Vemos que en la interfaz switch_1-eth3, los paquetes llegan correctamente y se dirigen hacia el host 2.

Ahora si, probemos con un caso similar al anterior pero usando al host 1 como cliente.

```
> xterm host_2 host_1
```

En la terminal para el host 2 (servidor)

```
> iperf -s -p 5001 -u -i 1
```

```
-----  
Server listening on UDP port 5001  
UDP buffer size: 208 KByte (default)  
-----  
█
```

Figura 28: Output server

En la terminal para el host 1 (cliente)

```
> iperf -c 10.0.0.2 -u -p 5001
```

```
-----  
Client connecting to 10.0.0.2, 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 41877 connected with 10.0.0.2 port 5001  
[ ID] Interval      Transfer    Bandwidth  
[ 1] 0.0000-10.0154 sec 1.25 MBytes 1.05 Mbits/sec  
[ 1] Sent 896 datagrams  
[ 5] WARNING: did not receive ack of last datagram after 10 tries.
```

Figura 29: Output cliente

udp.port == 5001						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
2	0.011417...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
3	0.011441...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
4	0.022572...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
5	0.033880...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
6	0.045083...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
7	0.056314...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
8	0.067525...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
9	0.078732...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
10	0.089955...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
11	0.101175...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
12	0.112384...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
13	0.123648...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
14	0.134723...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
15	0.145916...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
16	0.157266...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
17	0.168466...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
18	0.179663...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
19	0.190886...	10.0.0.1	10.0.0.2	UDP	1512	55118 → 5001 Len=1470
▶ Frame 1: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface switch_1-eth2, ▶ Ethernet II, Src: 00:80:00_00:00:01 (00:80:00:00:00:01), Dst: 00:80:00_00:00:02 (00:80:00:00:00:02) ▶ Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.2 ▶ User Datagram Protocol, Src Port: 55118, Dst Port: 5001 ▶ Data (1470 bytes)						

Figura 30: Captura Wireshark UDP puerto 5001

En la interfaz switch_1-eth2 linkeada del lado del host 1, vemos que los paquetes llegan al switch 1.

udp.port == 5001						
No.	Time	Source	Destination	Protocol	Length	Info

Figura 31: Captura Wireshark UDP puerto 5001 switch_1-eth3

Sin embargo, si vamos a la interfaz switch_1-eth3 (linkeada del lado del host 2) vemos que no llega ningún paquete proveniente del host 1 que tengan como puerto de destino 5001 y sean por el protocolo UDP.

5.2.4. Bloqueo de MAC entre host 1 y host 3

5.2.4.1. TCP - host 1 a host 3

En este caso vamos a probar enviar paquetes entre el host 1 y el host 3. Utilizaremos un servidor TCP en el host 3 y como cliente el host 1.

```
> xterm host_1 host_3
```

En la terminal para el host 3 (servidor).

```
> iperf -s -p 1234 -i 1
```

```
-----
Server listening on TCP port 1234
TCP window size: 85.3 KByte (default)
-----
█
```

Figura 32: Output server

En la terminal para el host 1 (cliente).

```
> iperf -c 10.0.0.3 -p 1234
```

```
tcp connect failed: Connection timed out
-----
Client connecting to 10.0.0.3, TCP port 1234
TCP window size: -1.00 Byte (default)
-----
[ 1] local 0.0.0.0 port 0 connected with 10.0.0.3_port 1234
```

Figura 33: Output cliente

tcp.port == 1234						
No.	Time	Source	Destination	Protocol	Length	Info

Figura 34: Captura Wireshark TCP puerto 1234

Vemos que ningún paquete llega al host 3, por lo cual el firewall está cumpliendo su función. Sin embargo, si capturamos paquetes en la interfaz switch_1-eth2 (linkeada del lado del host 1) vemos que los paquetes sí llegan al switch 1 para luego ser filtrados como se ve en la imagen anterior.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.0.1	10.0.0.3	TCP	74	46278 → 1234 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=
2	1.011094	10.0.0.1	10.0.0.3	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 46278 → 1234
3	3.031068	10.0.0.1	10.0.0.3	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 46278 → 1234
4	7.155829	10.0.0.1	10.0.0.3	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 46278 → 1234
5	15.35102	10.0.0.1	10.0.0.3	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 46278 → 1234
6	31.47585	10.0.0.1	10.0.0.3	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 46278 → 1234
7	63.99184	10.0.0.1	10.0.0.3	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 46278 → 1234

Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface switch 1-eth2, 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
 Transmission Control Protocol, Src Port: 46278, Dst Port: 1234, Seq: 0, Len: 0

Figura 35: Captura Wireshark TCP puerto 1234 switch_1-eth2

5.2.4.2. UDP - host 1 a host 3

Ahora, probemos el mismo escenario pero usando UDP en lugar de TCP.

```
> xterm host_1 host_3
```

En la terminal para el host 3 (servidor).

```
> iperf -s -p 1234 -u -i 1
```

```

-----
Server listening on UDP port 1234
UDP buffer size: 208 KByte (default)
-----

```

Figura 36: Output servidor

En la terminal para el host 1 (cliente).

```
> iperf -c 10.0.0.3 -u -p 1234
```



```
-----
Client connecting to 10.0.0.3, UDP port 1234
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.1 port 45667 connected with 10.0.0.3 port 1234
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-10.0153 sec 1.25 MBytes 1.05 Mbits/sec
[ 1] Sent 896 datagrams
[ 5] WARNING: did not receive ack of last datagram after 10 tries.
```

Figura 37: Output cliente

udp.port == 1234						
No.	Time	Source	Destination	Protocol	Length	Info

Figura 38: Captura Wireshark UDP puerto 1234

Vemos que ningún paquete llega al host 3, por lo cual el firewall está cumpliendo su función. Sin embargo, si capturamos paquetes en la interfaz switch_1-eth2 (linkeada del lado del host 1) vemos que los paquetes sí llegan al switch 1 para luego ser filtrados como se ve en la imagen anterior.

Go to the last packet						
udp.port == 1234						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
2	0.011483...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
3	0.011510...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
4	0.022572...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
5	0.033808...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
6	0.045026...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
7	0.056313...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
8	0.067526...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
9	0.078741...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
10	0.089927...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
11	0.101123...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
12	0.112365...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
13	0.123523...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
14	0.134729...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
15	0.146002...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
16	0.157244...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
17	0.168400...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
18	0.179615...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
19	0.190885...	10.0.0.1	10.0.0.3	UDP	1512	37604 → 1234 Len=1470
▶ Frame 1: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface switch_1-eth2, ▶ 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: 37604, Dst Port: 1234 ▶ Data (1470 bytes)						

Figura 39: Captura Wireshark UDP puerto 1234 switch_1-eth2

Ahora probemos la vuelta, es decir, host 3 como cliente y host 1 como servidor.

5.2.4.3. TCP - host 3 a host 1

Vamos a probar enviar paquetes entre el host 3 y el host 1. Utilizaremos un servidor TCP en el host 1 y como cliente el host 3.

```
> xterm host_1 host_3
```

En la terminal para el host 1 (servidor).

```
> iperf -s -p 1234 -i 1
```

```
-----
Server listening on TCP port 1234
TCP window size: 85.3 KByte (default)
-----
█
```

Figura 40: Output servidor

En la terminal para el host 3 (cliente).

```
> iperf -c 10.0.0.1 -p 1234
```

```
tcp connect failed: Connection timed out
-----
Client connecting to 10.0.0.1, TCP port 1234
TCP window size: -1.00 Byte (default)
-----
[ 1] local 0.0.0.0 port 0 connected with 10.0.0.1 port 1234
```

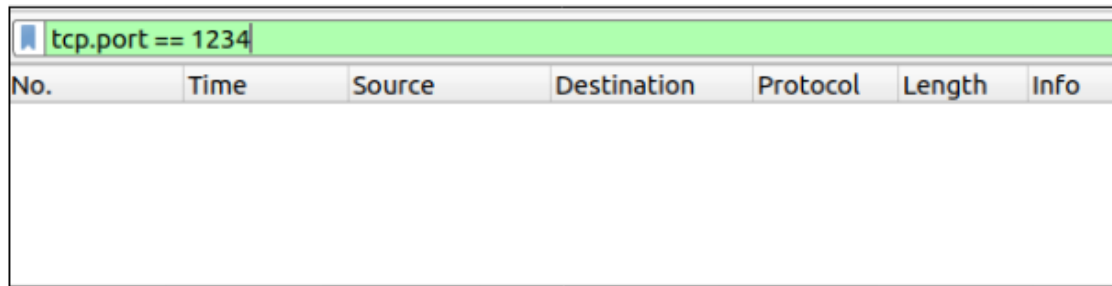
Figura 41: Output cliente

tcp.port == 1234						
No.	Time	Source	Destination	Protocol	Length	Info
9	17.73449	10.0.0.3	10.0.0.1	TCP	74	40672 → 1234 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=
10	18.74639	10.0.0.3	10.0.0.1	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 40672 → 1234
11	20.76231	10.0.0.3	10.0.0.1	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 40672 → 1234
12	24.83030	10.0.0.3	10.0.0.1	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 40672 → 1234
15	33.61036	10.0.0.3	10.0.0.1	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 40672 → 1234
22	49.14895	10.0.0.3	10.0.0.1	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 40672 → 1234
23	82.68490	10.0.0.3	10.0.0.1	TCP	74	[TCP Retransmission] [TCP Port numbers reused] 40672 → 1234

▶ Frame 9: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface switch_1-eth1, id 0 ▶ Ethernet II, Src: 00:00:00:00:00:03 (00:00:00:00:00:03), Dst: 00:00:00:00:00:01 (00:00:00:00:00:01) ▶ Internet Protocol Version 4, Src: 10.0.0.3, Dst: 10.0.0.1 ▶ Transmission Control Protocol, Src Port: 40672, Dst Port: 1234, Seq: 0, Len: 0						
---	--	--	--	--	--	--

Figura 42: Captura Wireshark TCP puerto 1234

Vemos que ningún paquete llega al host 1 y que llegan los paquetes de retransmisión al switch 1 por medio de la interfaz switch_1-eth1 por lo cual el firewall está cumpliendo su función. Si capturamos paquetes en la interfaz switch_1-eth2 (linkeada del lado del host 1) vemos que ningún paquete llega al host 1.



The image shows a Wireshark capture window. At the top, a green filter bar contains the text 'tcp.port == 1234'. Below this is a table with the following columns: No., Time, Source, Destination, Protocol, Length, and Info. The table is currently empty.

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

Figura 43: Captura Wireshark TCP puerto 1234 switch_1-eh2


5.2.4.4. UDP - host 3 a host 1

Ahora, probemos el mismo escenario pero usando UDP en lugar de TCP.

```
> xterm host_1 host_3
```

En la terminal para el host 1 (servidor).

```
> iperf -s -p 1234 -u -i 1
```



The image shows a terminal window with the following output:

```
-----  
Server listening on UDP port 1234  
UDP buffer size: 208 KByte (default)  
-----  
□
```

Figura 44: Output server

En la terminal para el host 3 (cliente).

```
> iperf -c 10.0.0.1 -u -p 1234
```

```

-----
Client connecting to 10.0.0.1, UDP port 1234
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.3 port 45430 connected with 10.0.0.1 port 1234
[ ID] Interval      Transfer      Bandwidth
[ 1] 0.0000-10.0154 sec 1.25 MBytes  1.05 Mbits/sec
[ 1] Sent 896 datagrams
[ 5] WARNING: did not receive ack of last datagram after 10 tries.

```

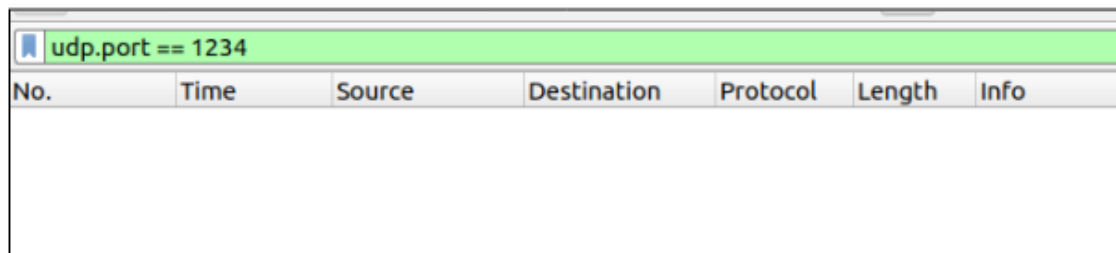
Figura 45: Output cliente

udp.port == 1234						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
2	0.009366...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
3	0.009383...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
4	0.020257...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
5	0.031502...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
6	0.042814...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
7	0.054807...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
8	0.065225...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
9	0.076436...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
10	0.087586...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
11	0.098859...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
12	0.110011...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
13	0.121270...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
14	0.132542...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
15	0.143772...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
16	0.154991...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
17	0.166137...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
18	0.177408...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470
19	0.188553...	10.0.0.3	10.0.0.1	UDP	1512	45430 → 1234 Len=1470

> Frame 1: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits) on interface switch_1-eth1,
 > Ethernet II, Src: 00:00:00_00:00:03 (00:00:00:00:00:03), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
 > Internet Protocol Version 4, Src: 10.0.0.3, Dst: 10.0.0.1
 > User Datagram Protocol, Src Port: 45430, Dst Port: 1234
 > Data (1470 bytes)

Figura 46: Captura Wireshark UDP puerto 1234

Vemos que ningún paquete llega al host 1 y que llegan los paquetes de retransmisión al switch 1 por medio de la interfaz switch_1-eth1 por lo cual el firewall está cumpliendo su función. Si capturamos paquetes en la interfaz switch_1-eth2 (linkeada del lado del host 1) vemos que ningún paquete llega al host 1.



No.	Time	Source	Destination	Protocol	Length	Info

Figura 47: Captura Wireshark UDP puerto 1234 switch_1-eth2

5.2.5. Conexión sin pasar por firewall

La siguiente prueba apunta a ver que si bien el firewall, de acuerdo a las reglas definidas, debe bloquear todo el tráfico que esté destinado al puerto 80, esto no quiere decir que se descarte el tráfico si el mismo no pasa por el switch configurado con el firewall. En particular, si intentamos comunicar mediante TCP al host 3 (como servidor) y al host 4 como cliente entonces deberíamos ver una comunicación exitosa.

```
> xterm host_3 host_4
```

En la terminal para el host 3 (servidor).

```
> iperf -s -p 80 -i 1
```

```
=====
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
=====
[ 1] local 10.0.0.3 port 80 connected with 10.0.0.4 port 59624
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-1.0000 sec  2.15 GBytes 18.5 Gbits/sec
[ 1] 1.0000-2.0000 sec  2.23 GBytes 19.2 Gbits/sec
[ 1] 2.0000-3.0000 sec  2.26 GBytes 19.4 Gbits/sec
[ 1] 3.0000-4.0000 sec  2.25 GBytes 19.3 Gbits/sec
[ 1] 4.0000-5.0000 sec  2.27 GBytes 19.5 Gbits/sec
[ 1] 5.0000-6.0000 sec  2.27 GBytes 19.5 Gbits/sec
[ 1] 6.0000-7.0000 sec  2.23 GBytes 19.2 Gbits/sec
[ 1] 7.0000-8.0000 sec  2.24 GBytes 19.3 Gbits/sec
[ 1] 8.0000-9.0000 sec  2.24 GBytes 19.3 Gbits/sec
[ 1] 9.0000-10.0000 sec 2.26 GBytes 19.4 Gbits/sec
[ 1] 0.0000-10.0005 sec 22.4 GBytes 19.2 Gbits/sec
[ ]
```

Figura 48: Output server

En la terminal para el host 4 (cliente).

```
> iperf -c 10.0.0.3 -p 80
```

```
-----  
Client connecting to 10.0.0.3, TCP port 80  
TCP window size: 85.3 KByte (default)  
-----  
[  1] local 10.0.0.4 port 59624 connected with 10.0.0.3 port 80  
[ ID] Interval      Transfer    Bandwidth  
[  1] 0.0000-10.0318 sec 22.4 GBytes 19.2 Gbits/sec
```

Figura 49: Output cliente

tcp.port == 80						
No.	Time	Source	Destination	Protocol	Length	Info

Figura 50: Captura Wireshark TCP puerto 80 switch_1-eth1

Lógicamente, por la interfaz switch_1-eth1 no transita ningún paquete.

tcp.port == 80						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000...	10.0.0.4	10.0.0.3	TCP	74	35470 → 80 [SYN] Seq=0 Win=42340
2	0.000030...	10.0.0.3	10.0.0.4	TCP	74	80 → 35470 [SYN, ACK] Seq=0 Ack=1
3	0.002194...	10.0.0.4	10.0.0.3	TCP	66	35470 → 80 [ACK] Seq=1 Ack=1 Win=
4	0.002344...	10.0.0.4	10.0.0.3	TCP	126	35470 → 80 [PSH, ACK] Seq=1 Ack=1
5	0.002366...	10.0.0.3	10.0.0.4	TCP	66	80 → 35470 [ACK] Seq=1 Ack=61 Win
6	0.002392...	10.0.0.3	10.0.0.4	TCP	94	80 → 35470 [PSH, ACK] Seq=1 Ack=6
7	0.002421...	10.0.0.4	10.0.0.3	TCP	4410	35470 → 80 [PSH, ACK] Seq=61 Ack=
8	0.002449...	10.0.0.4	10.0.0.3	TCP	4410	35470 → 80 [PSH, ACK] Seq=4405 A
9	0.002469...	10.0.0.4	10.0.0.3	TCP	4410	35470 → 80 [PSH, ACK] Seq=8749 A
10	0.002627...	10.0.0.3	10.0.0.4	TCP	66	80 → 35470 [ACK] Seq=29 Ack=13093
11	0.003868...	10.0.0.4	10.0.0.3	TCP	2962	35470 → 80 [PSH, ACK] Seq=13093 A
12	0.003900...	10.0.0.3	10.0.0.4	TCP	66	80 → 35470 [ACK] Seq=29 Ack=15989
13	0.004801...	10.0.0.4	10.0.0.3	TCP	66	35470 → 80 [ACK] Seq=15989 Ack=29
14	0.005785...	10.0.0.4	10.0.0.3	TCP	11650	35470 → 80 [PSH, ACK] Seq=15989 A
15	0.005791...	10.0.0.4	10.0.0.3	TCP	7306	35470 → 80 [PSH, ACK] Seq=27573 A
16	0.005815...	10.0.0.4	10.0.0.3	TCP	7306	35470 → 80 [PSH, ACK] Seq=34813 A
17	0.006280...	10.0.0.3	10.0.0.4	TCP	66	80 → 35470 [ACK] Seq=29 Ack=42053
18	0.006345...	10.0.0.4	10.0.0.3	TCP	5858	35470 → 80 [PSH, ACK] Seq=42053 A
19	0.006356...	10.0.0.3	10.0.0.4	TCP	66	80 → 35470 [ACK] Seq=29 Ack=47845
▶ Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface switch_2-eth2, ▶ Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: 00:00:00_00:00:03 (00:00:00:00:00:03) ▶ Internet Protocol Version 4, Src: 10.0.0.4, Dst: 10.0.0.3 ▶ Transmission Control Protocol, Src Port: 35470, Dst Port: 80, Seq: 0, Len: 0						

Figura 51: Captura Wireshark TCP puerto 80 switch_2-eth2

Por la interfaz switch_2-eth2 (del lado del host 4), se ve que llegan los paquetes al switch 2.

tcp.port == 80						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000...	10.0.0.4	10.0.0.3	TCP	74	34012 → 80 [SYN] Seq=0 Win=42340 Len=0
2	0.004827...	10.0.0.3	10.0.0.4	TCP	74	80 → 34012 [SYN, ACK] Seq=0 Ack=1 Win=42340
3	0.004871...	10.0.0.4	10.0.0.3	TCP	66	34012 → 80 [ACK] Seq=1 Ack=1 Win=42496
4	0.005009...	10.0.0.4	10.0.0.3	TCP	126	34012 → 80 [PSH, ACK] Seq=1 Ack=1 Win=42496
5	0.005069...	10.0.0.4	10.0.0.3	TCP	5858	34012 → 80 [PSH, ACK] Seq=61 Ack=1 Win=42496
6	0.005091...	10.0.0.4	10.0.0.3	TCP	5858	34012 → 80 [PSH, ACK] Seq=5853 Ack=1 Win=42496
7	0.005109...	10.0.0.4	10.0.0.3	TCP	1514	34012 → 80 [ACK] Seq=11645 Ack=1 Win=42496
8	0.006448...	10.0.0.3	10.0.0.4	TCP	66	80 → 34012 [ACK] Seq=1 Ack=61 Win=43520
9	0.006473...	10.0.0.4	10.0.0.3	TCP	2962	34012 → 80 [PSH, ACK] Seq=13093 Ack=1 Win=43520
10	0.006935...	10.0.0.3	10.0.0.4	TCP	66	[TCP Dup ACK 0#1] 80 → 34012 [ACK] Seq=1 Ack=5853 Win=3992
11	0.007473...	10.0.0.3	10.0.0.4	TCP	66	80 → 34012 [ACK] Seq=1 Ack=5853 Win=3992
12	0.007499...	10.0.0.4	10.0.0.3	TCP	10202	34012 → 80 [PSH, ACK] Seq=15989 Ack=1 Win=43520
13	0.007610...	10.0.0.3	10.0.0.4	TCP	66	[TCP Previous segment not captured] 80 → 34012 [ACK] Seq=1 Ack=5853 Win=3992
14	0.007625...	10.0.0.4	10.0.0.3	TCP	8754	34012 → 80 [PSH, ACK] Seq=26125 Ack=1 Win=43520
15	0.007634...	10.0.0.4	10.0.0.3	TCP	20338	34012 → 80 [PSH, ACK] Seq=34813 Ack=1 Win=43520
16	0.007661...	10.0.0.3	10.0.0.4	TCP	66	80 → 34012 [ACK] Seq=29 Ack=34813 Win=5840
17	0.007663...	10.0.0.3	10.0.0.4	TCP	66	80 → 34012 [ACK] Seq=29 Ack=55085 Win=10240
18	0.007669...	10.0.0.4	10.0.0.3	TCP	1514	34012 → 80 [PSH, ACK] Seq=55085 Ack=1 Win=43520
19	0.007671...	10.0.0.4	10.0.0.3	TCP	21786	34012 → 80 [PSH, ACK] Seq=56533 Ack=1 Win=43520
▶ Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface switch_2-eth3, id 0 ▶ Ethernet II, Src: 00:00:00_00:00:04 (00:00:00:00:00:04), Dst: 00:00:00_00:00:03 (00:00:00:00:00:03) ▶ Internet Protocol Version 4, Src: 10.0.0.4, Dst: 10.0.0.3 ▶ Transmission Control Protocol, Src Port: 34012, Dst Port: 80, Seq: 0, Len: 0						

Figura 52: Captura Wireshark TCP puerto 80 switch_2-eth3

Al mismo tiempo, por la interfaz switch_2-eth3 (del lado del host 3), se ve que llegan los paquetes al host 3.

6. Dificultades encontradas

- Determinar la mejor manera de armar el archivo *rules.json* para hacerlo lo más extensible posible sin necesidad de cambiar el código, a fin de poder usar más combinaciones de reglas.
- Debimos usar variables globales para los parámetros de entrada del firewall (el archivo de reglas y cuál switch sería el firewall) debido a que no pudimos pasarlos directamente como parámetros.

7. Conclusiones

En el trabajo práctico pudimos observar el comportamiento de una red al que se le instaló un cortafuegos en uno de sus switches, y siendo configurado gracias a las Software Defined Networks, implementada en el protocolo de OpenFlow.

De esta manera vimos que al enviar paquetes desde los diferentes hosts, estos mismos no llegaban a su destino debido a las diferentes reglas configuradas dentro del cortafuegos. Además vimos como OpenFlow y SDN permiten que la configuración de los routers sea de una manera sencilla, determinista y escalable, permitiendo al administrador de la red (nosotros) gestionarla desde un nivel más alto, utilizando software para conseguirlo.

Consideramos el trabajo práctico y la experiencia de realizarlo muy enriquecedora para la correcta comprensión de cómo pueden funcionar este tipo de sistemas en la vida real y las posibilidades que se ofrecen, además de profundizar en los conceptos teóricos y poder llevarlos correctamente a la práctica.