Trabajo Práctico 2: Software-Defined Networks

Facultad de Ingeniería de la Universidad de Buenos Aires

Redes

1C 2025

Grupo 4

Prystupiuk, Maximiliano
94853
mprystupiuk@fi.uba.ar

Valeriani, Matías Gabriel
108570
matiasvaleriani@gmail.com

Soro, Lucas Gustavo
95665
lsoro@fi.uba.ar

Jang, Lucas
109151
lucasjang01@gmail.com

Deciancio, Nicolás
92150
nicodec89@gmail.com

Índice

1 Introducción

Nuestro objetivo fue implementar una red SDN (Software Defined Network) que utiliza el protocolo OpenFlow, y se despliega en el emulador de red Mininet, con un controlador POX.

El propósito fue la simulación y experimentación de una red parametrizable, que permitiera tener una cantidad variable de switches en una topología lineal con dos hosts a cada extremo, y la implementación de un firewall, capaz de aplicar políticas específicas de bloqueo de tráfico en la capa de enlace.

Se busca conseguir, por último, integrar herramientas como Wireshark e iPerf para monitorear dicha red.

2 Hipótesis y suposiciones realizadas

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

- 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, en particular en distribuciones basados en Debian, como Ubuntu.

3 Implementación

Se utilizó Mininet como emulador de red, POX como controlador, una topología configurable a traves de Python, y las reglas del firewall en un archivo JSON.

3.1 Topología

Para el diseño de la red, se implementó una topología personalizada utilizando Mininet mediante la creación de una subclase de la clase `Topo`. Esta topología fue diseñada para ser configurable en cuanto al número de switches intermedios, permitiendo así adaptarse a distintas configuraciones de red de manera flexible.

Se definieron cuatro hosts con direcciones MAC fijas, los cuales fueron conectados de la siguiente manera: los hosts `h1` y `h2` se conectan al primer switch de la cadena, mientras que `h3` y `h4` se conectan al último. Los switches intermedios son conectados entre sí en forma lineal, formando una topología de tipo cadena. Esta estructura facilita el análisis del tráfico entre extremos de la red a través de múltiples saltos.

El número de switches puede ajustarse mediante un parámetro al inicializar la topología, garantizando que, como mínimo, haya un switch. Además, todos los switches operan en modo `standalone` para evitar interferencias con controladores externos durante las pruebas.

Esta topología permite simular un entorno de red SDN básico pero suficientemente versátil para el desarrollo y prueba de políticas de control como las implementadas en el firewall.

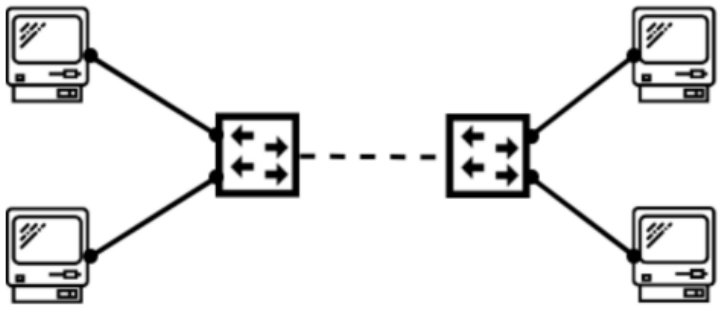


Figure 1: Topología de red implementada con hosts en los extremos y switches en cadena

3.2 Controlador POX

Para la gestión del comportamiento de la red, se utilizó el controlador POX, una plataforma desarrollada en Python orientada al desarrollo de aplicaciones SDN mediante el uso del protocolo OpenFlow. Este controlador permite definir cómo deben actuar los switches ante distintos tipos de tráfico, facilitando la implementación de políticas personalizadas.

En el contexto de este trabajo, POX fue clave para implementar reglas de control sobre los flujos de paquetes que atraviesan la red. Se desarrolló un módulo propio que actúa como un firewall, permitiendo establecer reglas específicas para bloquear o permitir tráfico en función de criterios definidos en un archivo de configuración. Gracias a la flexibilidad de POX, fue posible integrar esta funcionalidad de forma sencilla y efectiva.

3.3 Firewall

Para aplicar políticas de seguridad sobre el tráfico de la red, se desarrolló un módulo personalizado que actúa como firewall, ejecutado desde el controlador POX. Este módulo permite instalar reglas en los switches administrados, las cuales determinan qué paquetes deben ser descartados según ciertas condiciones.

Las reglas se definen externamente en un archivo de configuración JSON, lo cual permite modificar el comportamiento del firewall sin alterar el código. Este archivo contiene dos campos principales:

- **switch**: indica el switch donde se deben aplicar las políticas.
- **policies**: es una lista de objetos que definen condiciones bajo las cuales los paquetes deben bloquearse.

Cada política puede especificar diversos campos como la dirección IP de origen/destino (`nw_src, nw_dst`), puerto destino (`tp_dst`), protocolo (`nw_proto`), tipo de datagrama (`dl_type`), y direcciones MAC (`dl_src, dl_dst`). Por ejemplo, la siguiente política bloquea todo el tráfico TCP dirigido al puerto 8080 en redes IPv4:

```
{ "tp_dst": "8080", "nw_proto": "tcp", "dl_type": "ipv4" }
```

Durante la carga de políticas, si algún campo clave no está presente (como `nw_proto` o `dl_type`), el módulo genera automáticamente variantes para cubrir todos los posibles valores relevantes. Esta decisión de diseño permite escribir reglas más simples y genéricas, mientras que internamente se amplían para abarcar todos los casos necesarios. Por ejemplo, si se omite `nw_proto`, se generan automáticamente versiones de la regla para TCP, UDP, ICMP, etc.

Una vez que el switch correspondiente se conecta al controlador (`ConnectionUp`), las políticas cargadas se transforman en reglas de OpenFlow utilizando la clase `ofp_flow_mod`, y son instaladas en el switch. Cada coincidencia en el JSON se mapea a los campos correspondientes en el mensaje de OpenFlow, como `tp_dst`, `dl_src`, etc.

Además, el módulo incluye funcionalidad para mostrar en consola los paquetes entrantes con detalles como IP origen/destino, protocolo y puertos, con formato de colores para facilitar la depuración y seguimiento del tráfico.

Esta arquitectura modular y flexible facilita tanto la experimentación con diferentes políticas como la extensión del sistema en trabajos futuros.

4 Pruebas

Para realizar las pruebas usamos la siguiente topología con 3 switches en cadena, y dos hosts en cada extremo, tal como se solicita en el enunciado del trabajo practico. Ademas, para tener una mejor visualización en las capturas de wireshark y en el log de POX, decidimos mostrar unicamente paquetes IPv4.

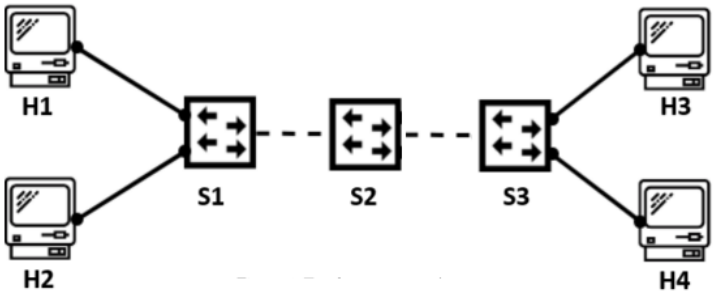


Figure 2: Topología utilizada para las pruebas

4.1 Políticas

Las reglas definidas se establecieron para satisfacer lo siguiente:

1. Se deben descartar todos los mensajes cuyo puerto destino sea 80.

2. Se deben descartar todos los mensajes que provengan del host 1, tengan como puerto destino el 5001, y estén utilizando el protocolo UDP.

3. Se deben elegir dos hosts cualesquiera y los mismos no deben poder comunicarse de ninguna forma.

Dichas reglas las almacenamos en un `.json`, lo cual nos permite la inclusión de nuevas reglas que permitan bloquear mensajes con diferentes características.

```
{
 "tp_dst": "80",
 "nw_proto": "tcp"
},
{
 "tp_dst": "80",
 "nw_proto": "udp"
},
{
 "tp_dst": "5001",
 "nw_proto": "udp",
 "dl_src": "00:00:00:00:00:01"
},
{
 "dl_src": "00:00:00:00:00:02",
 "dl_dst": "00:00:00:00:00:03"
},
{
 "dl_src": "00:00:00:00:00:03",
 "dl_dst": "00:00:00:00:00:02"
}
```

En el log del controlador visualizamos las reglas de firewall:

INFO:firewall:

Rule	Dst Port	Transport layer Protocol	Src MAC Addr	Dst MAC Addr
R_1	80	tcp	*	*
R_2	80	udp	*	*
R_3	5001	udp	00:00:00:00:00:01	*
R_4	*	*	00:00:00:00:00:02	00:00:00:00:00:03
R_5	*	*	00:00:00:00:00:03	00:00:00:00:00:02

Figure 3: Info policies

4.2 Pingall

Siguiendo las políticas que establecimos, al hacer `pingall` observamos como se permite la conexión entre casi todos los hosts, menos entre `h2` y `h3`. A continuación, pondremos a prueba nuestra políticas, y algunos casos particulares donde el switch no permite algunas comunicaciones.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 X h4
h3 -> h1 X h4
h4 -> h1 h2 h3
*** Results: 16% dropped (10/12 received)
```

Figure 4: Pingall

4.3 Casos de prueba

4.3.1 Se deben descartar todos los mensajes cuyo puerto destino sea 80.

```
mininet> h4 curl h2
```

Realizamos una consulta HTTP (puerto 80) entre h4 y h2. Podemos ver que hay 3 switches de por medio entre ambos hosts: `h2 - s1 - s2 - s3 - h4`. Se puede observar como los paquetes van pasando por el switch 3,2 y 1, pero nunca pasan por la interfaz `s1-eth2` lo cual se confirma que los paquetes recorren la topologia pero nunca llegan al host de destino `h2`

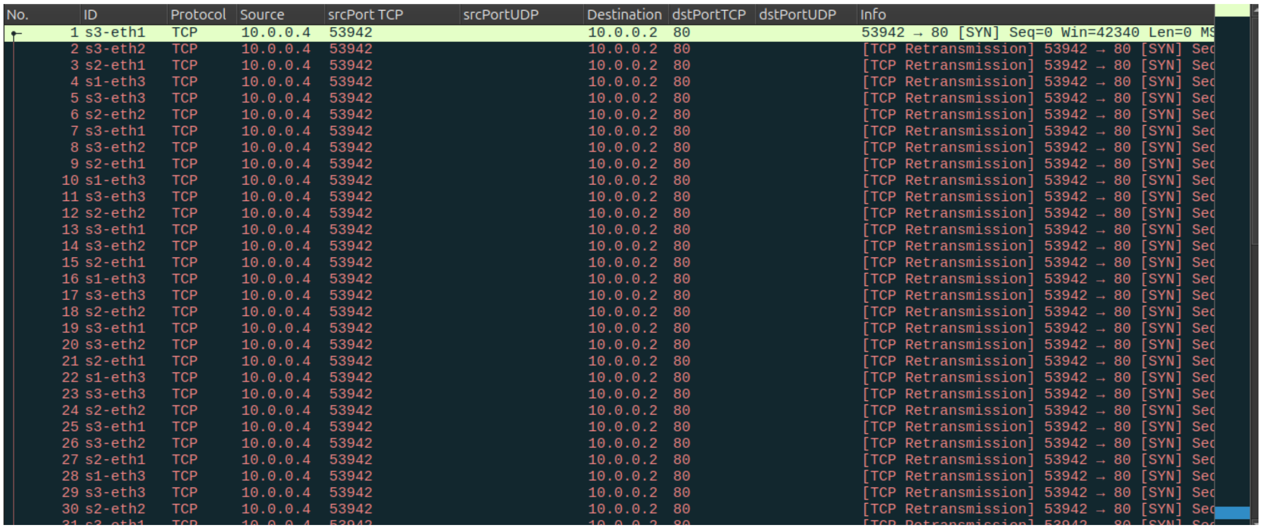


Figure 5: Captura de Wireshark de todas las interfaces de la topología

En el log del controlador POX se puede observar que se muestra constantemente los mensajes que llegan al switch 2 y 3, pero nunca al switch 1. Ya que las reglas que establecimos, hacen que el paquete sea descartado al llegar al switch 1.

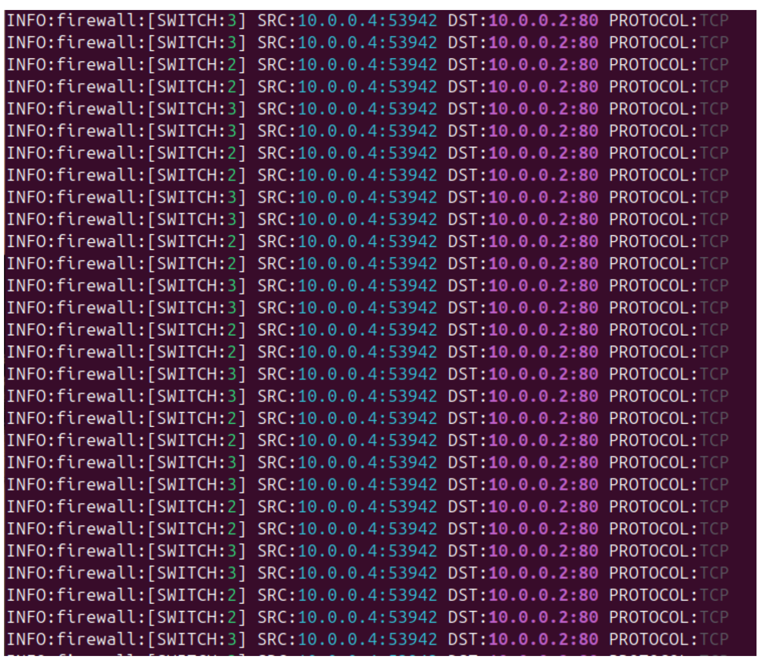


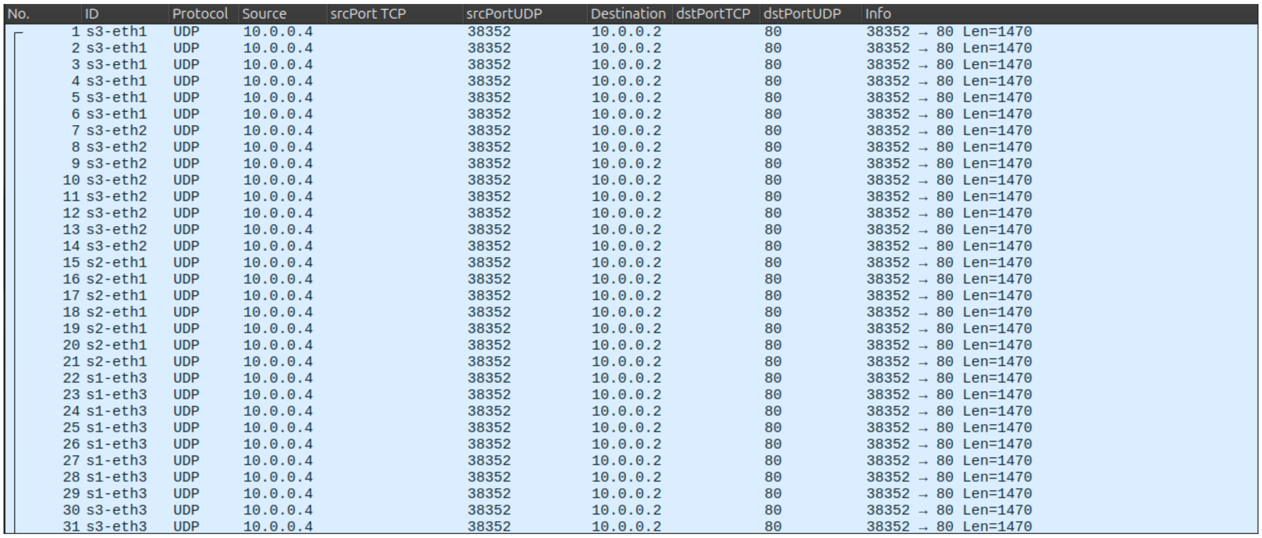
Figure 6: Logs del controlador

4.3.2 Descartar todos los mensajes cuyo puerto destino sea 80 sobre UDP

Usando `iperf`, generamos tráfico UDP entre dos hosts: `h2` actúa como servidor escuchando en el puerto 80, mientras que `h4` envía tráfico al destino configurado.

```
   mininet> h2 iperf -u -s -p 80&
mininet> h4 iperf -u -c h2 -p 80
```

Similar a la captura de Wireshark anterior, podemos observar que el switch 1 descarta los paquetes, lo cual nunca llega a pasar por la interfaz `s1-eth2`, motivo por el que nunca llega al host `h2`



No.	ID	Protocol	Source	srcPort TCP	srcPortUDP	Destination	dstPortTCP	dstPortUDP	Info
1	s3-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
2	s3-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
3	s3-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
4	s3-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
5	s3-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
6	s3-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
7	s3-eth2	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
8	s3-eth2	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
9	s3-eth2	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
10	s3-eth2	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
11	s3-eth2	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
12	s3-eth2	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
13	s3-eth2	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
14	s3-eth2	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
15	s2-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
16	s2-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
17	s2-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
18	s2-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
19	s2-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
20	s2-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
21	s2-eth1	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
22	s1-eth3	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
23	s1-eth3	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
24	s1-eth3	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
25	s1-eth3	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
26	s1-eth3	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
27	s1-eth3	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
28	s1-eth3	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
29	s1-eth3	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
30	s3-eth3	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470
31	s3-eth3	UDP	10.0.0.4		38352	10.0.0.2		80	38352 → 80 Len=1470

Figure 7: Captura de Wireshark de todas las interfaces de la topología

En el log del controlador podemos ver que solo se imprime como los paquetes pasan por los switches s2 y s3 pero son descartados por el switch s1.

```
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4:38352 DST:10.0.0.2:80 PROTOCOL:UDP
```

Figure 8: Logs del controlador

En **mininet** se confirma **h2** no recibe ningun paquete, motivo por el cual nunca pudo enviar un **ACK**.

```
mininet> h2 iperf -u -s -p 80&
mininet> h4 iperf -u -c h2 -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 38352 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
[  3] WARNING: did not receive ack of last datagram after 10 tries.
```

Figure 9: Captura de Mininet

4.3.3 Se deben descartar todos los mensajes que provengan del host 1, tengan como puerto destino el 5001, y estén utilizando el protocolo UDP

```
mininet> h3 iperf -u -s -p 5001&
mininet> h1 iperf -u -c h3 -p 5001
```

Esta regla significa que no se podrá enviar ningun mensaje cuyo origen sea el host 1, puerto destino 5001 y utilizando UDP. Para ello usando iperf, podemos generar trafico con las características solicitadas. Podemos observar que al enviar mensajes de **h1** a **h3** con estas condiciones, nunca pasara mas allá de la interfaz **s1-eth1**, es decir serán descartados ni bien lleguen al switch s1. De esta forma, Wireshark mostrara solamente paquetes de la interfaz **s1-eth1**.

No.	ID	Protocol	Source	srcPort TCP	srcPortUI	Destination	dstPortTCP	dstPortUDP	Info
1	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
2	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
3	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
4	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
5	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
6	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
7	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
8	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
9	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
10	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
11	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
12	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
13	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
14	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
15	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
16	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
17	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
18	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
19	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
20	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
21	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
22	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
23	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
24	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
25	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
26	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
27	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
28	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
29	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
30	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
31	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470
32	s1-eth1	UDP	10.0.0.1		40424	10.0.0.3		5001	40424 → 5001 Len=1470

Figure 10: Captura de Wireshark de todas las interfaces de la topología

En el log del controlador simplemente no veremos nada, ya que el switch por el comienzan a transmitirse los mensajes, es el switch s1 y este descarta los mismos. De esta forma no se muestra nada en el log.

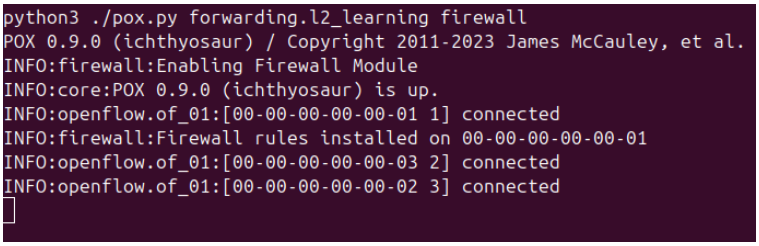


Figure 11: Logs del controlador

En **mininet** se observa que no se recibió ningún ACK.

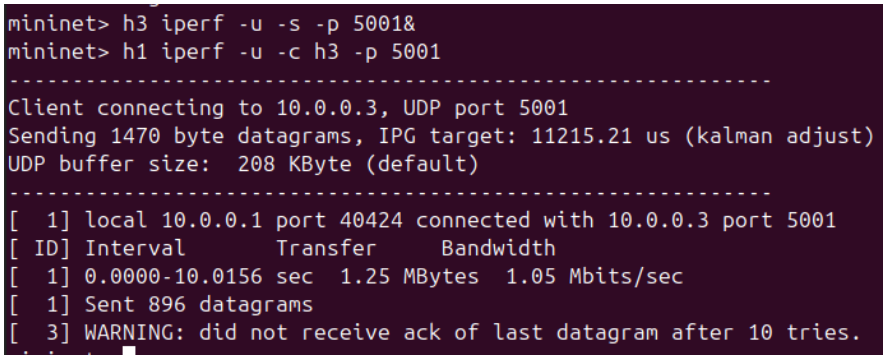


Figure 12: Captura de Mininet

4.3.4 Se deben elegir dos hosts cualesquiera y los mismos no deben poder comunicarse de ninguna forma

Para probar que la comunicación entre estos dos hosts no esta permitida, intentaremos con **curl** y **ping** entre `h2` y `h3`, y entre `h2` y `h2`. De este modo, concluiremos que no se permite comunicación en ningún sentido entre `h2` y `h3`.

4.3.5 Conexión entre h2 y h3

Comenzaremos con `h2` como origen y `h3` como destino.

```
mininet> h2 ping h3
```

Se puede observar en wireshark que nunca se llega al host 3. Esto lo vemos ya que los paquetes son descartados cuando llegan al switch s1, motivo por el cual solo vemos paquetes de la interfaz `s1-eth2` que es lo que intenta enviar el host `h2`.

No.	ID	Protocol	Source	srcPort TCP	srcPortUI	Destination	dstPortTCP	dstPortUDP	Info
1	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=1/256, ttl=64 (no resp
2	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=2/512, ttl=64 (no resp
3	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=3/768, ttl=64 (no resp
4	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=4/1024, ttl=64 (no res
5	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=5/1280, ttl=64 (no res
6	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=6/1536, ttl=64 (no res
7	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=7/1792, ttl=64 (no res
8	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=8/2048, ttl=64 (no res
9	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=9/2304, ttl=64 (no res
10	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=10/2560, ttl=64 (no re
11	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=11/2816, ttl=64 (no re
12	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=12/3072, ttl=64 (no re
13	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=13/3328, ttl=64 (no re
14	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=14/3584, ttl=64 (no re
15	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=15/3840, ttl=64 (no re
16	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=16/4096, ttl=64 (no re
17	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=17/4352, ttl=64 (no re
18	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=18/4608, ttl=64 (no re
19	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=19/4864, ttl=64 (no re
20	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=20/5120, ttl=64 (no re
21	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=21/5376, ttl=64 (no re
22	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=22/5632, ttl=64 (no re
23	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=23/5888, ttl=64 (no re
24	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=24/6144, ttl=64 (no re
25	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=25/6400, ttl=64 (no re
26	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=26/6656, ttl=64 (no re
27	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=27/6912, ttl=64 (no re
28	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=28/7168, ttl=64 (no re
29	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=29/7424, ttl=64 (no re
30	s1-eth2	ICMP	10.0.0.2			10.0.0.3			Echo (ping) request id=0x1ae3, seq=30/7680, ttl=64 (no re

Figure 13: Captura de Wireshark de todas las interfaces de la topología

En el log de POX no vemos nada, ya que al descartarse cuando llega al switch s1, no se imprimirá nada.

```
python3 ./pox.py forwarding.l2_learning firewall
POX 0.9.0 (ichthyosaur) / Copyright 2011-2023 James McCauley, et al
INFO:firewall:Enabling Firewall Module
INFO:core:POX 0.9.0 (ichthyosaur) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
INFO:firewall:Firewall rules installed on 00-00-00-00-00-01
INFO:openflow.of_01:[00-00-00-00-00-03 2] connected
INFO:openflow.of_01:[00-00-00-00-00-02 3] connected
```

Figure 14: Logs del controlador

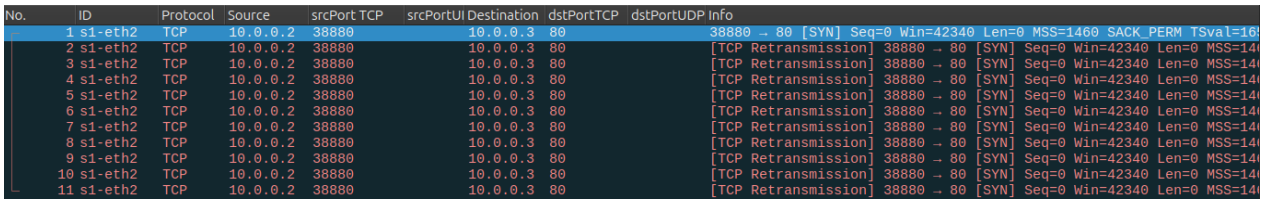
En `mininet` confirmamos que no se recibió un ACK, dando así una perdida del 100%.

```
mininet> h2 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
^C
--- 10.0.0.3 ping statistics ---
58 packets transmitted, 0 received, 100% packet loss, time 58387ms
```

Figure 15: Captura de Mininet

`mininet> h2 curl h3`

Cuando realizamos una consulta HTTP (puerto 80), vemos nuevamente que ningun paquete llega al switch 1. Es decir, solo veremos paquetes de la interfaz `s1-eth2`.



No.	ID	Protocol	Source	srcPort TCP	srcPortUI	Destination	dstPortTCP	dstPortUDP	Info
1	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM TSval=16
2	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		[TCP Retransmission] 38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=14
3	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		[TCP Retransmission] 38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=14
4	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		[TCP Retransmission] 38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=14
5	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		[TCP Retransmission] 38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=14
6	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		[TCP Retransmission] 38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=14
7	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		[TCP Retransmission] 38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=14
8	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		[TCP Retransmission] 38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=14
9	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		[TCP Retransmission] 38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=14
10	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		[TCP Retransmission] 38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=14
11	s1-eth2	TCP	10.0.0.2	38880		10.0.0.3	80		[TCP Retransmission] 38880 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=14

Figure 16: Captura de Wireshark de todas las interfaces de la topología

Similar al caso anterior, no se mostrara nada en el log de POX ya que los paquetes no llegan a pasar por ningun switch.

```
python3 ./pox.py forwarding.l2_learning firewall
POX 0.9.0 (ichthyosaur) / Copyright 2011-2023 James McCauley, et al
INFO:firewall:Enabling Firewall Module
INFO:core:POX 0.9.0 (ichthyosaur) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
INFO:firewall:Firewall rules installed on 00-00-00-00-00-01
INFO:openflow.of_01:[00-00-00-00-00-03 2] connected
INFO:openflow.of_01:[00-00-00-00-00-02 3] connected
```

Figure 17: Logs del controlador

4.3.6 Conexión entre h3 y h2

Ahora realizaremos pruebas con `h3` como origen y `h2` como destino.

```
mininet> h3 ping h2
```

En Wireshark se observa que los paquetes pasan por los switch s2 y s3, pero son descartados al llegar al switch s1. Esto lo podemos ver ya que no vemos nada que pasa por la interfaz `s1-eth2`, concluyendo que nunca se llega al host `h2`.

No.	ID	Protocol	Source	srcPort TCP	srcPortUI	Destination	dstPortTCP	dstPortUDP	Info
1	s3-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=1/256, ttl=64 (no response fo
2	s3-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=1/256, ttl=64 (no response fo
3	s2-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=1/256, ttl=64 (no response fo
4	s1-eth3	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=1/256, ttl=64 (no response fo
5	s3-eth3	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=1/256, ttl=64 (no response fo
6	s2-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=1/256, ttl=64 (no response fo
7	s3-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=2/512, ttl=64 (no response fo
8	s3-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=2/512, ttl=64 (no response fo
9	s2-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=2/512, ttl=64 (no response fo
10	s1-eth3	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=2/512, ttl=64 (no response fo
11	s2-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=2/512, ttl=64 (no response fo
12	s3-eth3	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=2/512, ttl=64 (no response fo
13	s3-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=3/768, ttl=64 (no response fo
14	s3-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=3/768, ttl=64 (no response fo
15	s2-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=3/768, ttl=64 (no response fo
16	s1-eth3	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=3/768, ttl=64 (no response fo
17	s3-eth3	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=3/768, ttl=64 (no response fo
18	s2-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=3/768, ttl=64 (no response fo
19	s3-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=4/1024, ttl=64 (no response f
20	s3-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=4/1024, ttl=64 (no response f
21	s2-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=4/1024, ttl=64 (no response f
22	s1-eth3	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=4/1024, ttl=64 (no response f
23	s3-eth3	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=4/1024, ttl=64 (no response f
24	s2-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=4/1024, ttl=64 (no response f
25	s3-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=5/1280, ttl=64 (no response f
26	s3-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=5/1280, ttl=64 (no response f
27	s2-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=5/1280, ttl=64 (no response f
28	s1-eth3	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=5/1280, ttl=64 (no response f
29	s3-eth3	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=5/1280, ttl=64 (no response f
30	s2-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=5/1280, ttl=64 (no response f
31	s3-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=6/1536, ttl=64 (no response f
32	s3-eth2	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=6/1536, ttl=64 (no response f
33	s2-eth1	ICMP	10.0.0.3			10.0.0.2			Echo (ping) request id=0x1d4c, seq=6/1536, ttl=64 (no response f

Figure 18: Captura de Wireshark de todas las interfaces de la topología

En este caso en el controlador de POX podemos ver que se imprime cuando pasa por los switch s2 y s3, pero al no imprimir nada de switch s1, concluimos que los paquetes nunca llegan al host `h2`.

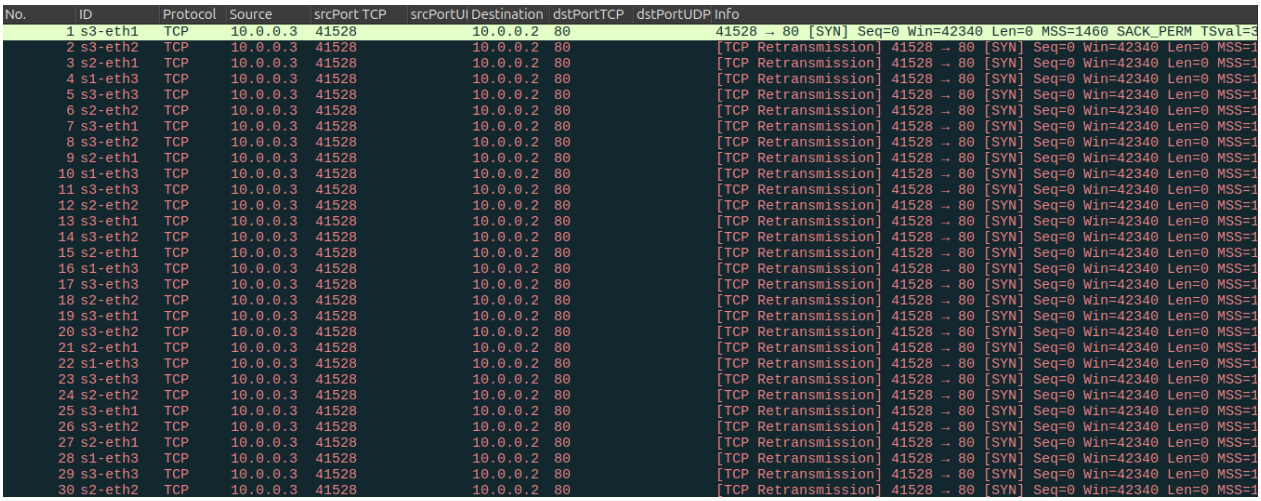
```
python3 ./pox.py forwarding.l2_learning firewall
POX 0.9.0 (ichthyosaur) / Copyright 2011-2023 James McCauley, et al.
INFO:firewall:Enabling Firewall Module
INFO:core:POX 0.9.0 (ichthyosaur) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
INFO:firewall:Firewall rules installed on 00-00-00-00-00-01
INFO:openflow.of_01:[00-00-00-00-00-03 2] connected
INFO:openflow.of_01:[00-00-00-00-00-02 3] connected
INFO:firewall:[SWITCH:3] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.3: DST:10.0.0.2: PROTOCOL:ICMP
```

Figure 19: Logs del controlador

```
mininet> h2 ping h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
^C
--- 10.0.0.3 ping statistics ---
58 packets transmitted, 0 received, 100% packet loss, time 58387ms
```

Figure 20: Captura de Mininet

`mininet> h3 curl h2`
Cuando realizamos una consulta HTTP (puerto 80), vemos nuevamente que ningun paquete llega al switch 1. Es decir, veremos paquetes que pasan por los switch s2 y s3, pero ninguno correspondiente a la interfaz `s1-eth2`.



No.	ID	Protocol	Source	srcPort TCP	srcPortUI	Destination	dstPortTCP	dstPortUDP	Info
1	s3-eth1	TCP	10.0.0.3	41528		10.0.0.2	80		41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM TSval=3
2	s3-eth2	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
3	s2-eth1	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
4	s1-eth3	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
5	s3-eth3	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
6	s2-eth2	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
7	s3-eth1	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
8	s3-eth2	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
9	s2-eth1	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
10	s1-eth3	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
11	s3-eth3	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
12	s2-eth2	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
13	s3-eth1	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
14	s3-eth2	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
15	s2-eth1	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
16	s1-eth3	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
17	s3-eth3	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
18	s2-eth2	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
19	s3-eth1	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
20	s3-eth2	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
21	s2-eth1	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
22	s1-eth3	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
23	s3-eth3	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
24	s2-eth2	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
25	s3-eth1	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
26	s3-eth2	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
27	s2-eth1	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
28	s1-eth3	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
29	s3-eth3	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
30	s2-eth2	TCP	10.0.0.3	41528		10.0.0.2	80		[TCP Retransmission] 41528 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1

Figure 21: Captura de Wireshark de todas las interfaces de la topología

Nuevamente en estos logs podemos ver como los mensajes pasan por los switch s2 y s3 pero no por el switch s1.

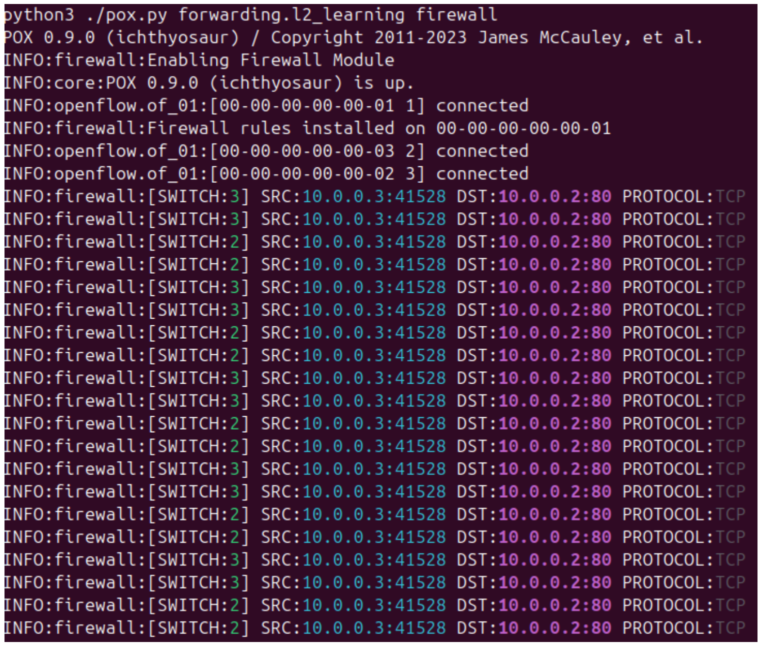


Figure 22: Logs del controlador

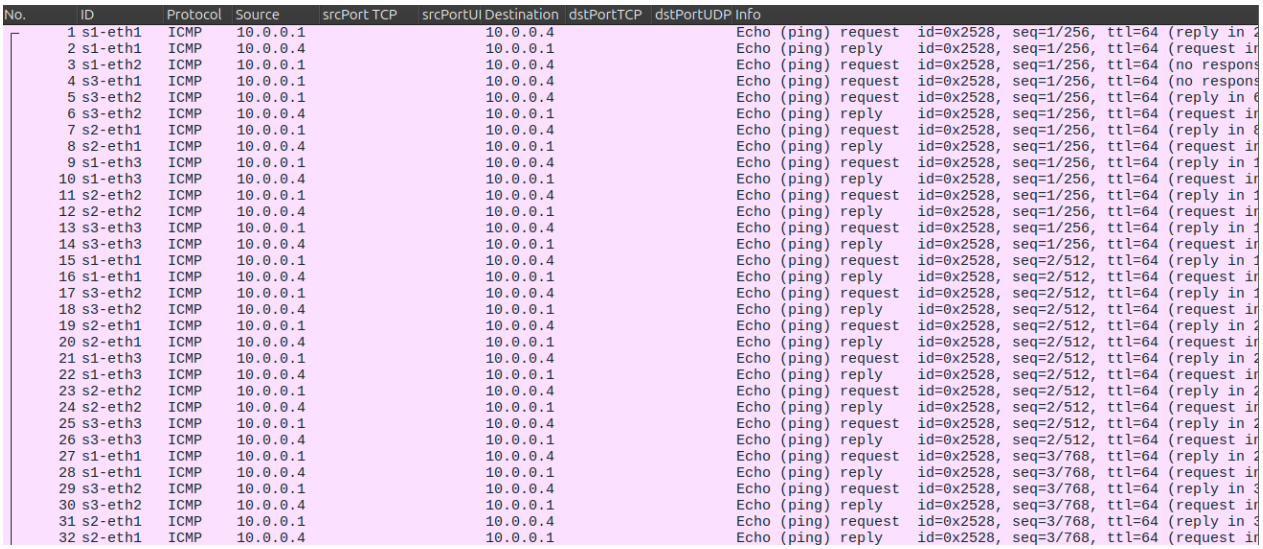
4.4 Casos permitidos

Anteriormente mostramos que con el Pingall, no se permitia ningun tipo de conexion entre h2 y h3. Ademas, fuimos viendo algunos casos de pruebas solicitados en los que se debia demostrar que no existia comunicacion. Ahora mostraremos algunos casos en los que la informacion puede ser recibida por el host destino. Con esto queremos demostrar que el switch s1 solo bloqueara los mensajes siguiendo las reglas que nosotros definimos.

Por ejemplo, podemos demostrar que se permite realizar un ping entre los host `h1` y `h4`, pasando asi por todos los switches de la topologia.

```
mininet> h1 ping h4
```

En Wireshark se observa que los paquetes ICMP son enviados y recibidos correctamente al pasar por todas las switches entre `h1` y `h4`.



No.	ID	Protocol	Source	srcPort TCP	srcPortUI	Destination	dstPortTCP	dstPortUDP	Info
1	s1-eth1	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=1/256, ttl=64 (reply in 2
2	s1-eth1	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=1/256, ttl=64 (request in
3	s1-eth2	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=1/256, ttl=64 (no respons
4	s3-eth1	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=1/256, ttl=64 (no respons
5	s3-eth2	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=1/256, ttl=64 (reply in 6
6	s3-eth2	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=1/256, ttl=64 (request in
7	s2-eth1	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=1/256, ttl=64 (reply in 8
8	s2-eth1	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=1/256, ttl=64 (request in
9	s1-eth3	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=1/256, ttl=64 (reply in 1
10	s1-eth3	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=1/256, ttl=64 (request in
11	s2-eth2	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=1/256, ttl=64 (reply in 1
12	s2-eth2	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=1/256, ttl=64 (request in
13	s3-eth3	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=1/256, ttl=64 (reply in 1
14	s3-eth3	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=1/256, ttl=64 (request in
15	s1-eth1	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=2/512, ttl=64 (reply in 1
16	s1-eth1	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=2/512, ttl=64 (request in
17	s3-eth2	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=2/512, ttl=64 (reply in 1
18	s3-eth2	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=2/512, ttl=64 (request in
19	s2-eth1	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=2/512, ttl=64 (reply in 2
20	s2-eth1	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=2/512, ttl=64 (request in
21	s1-eth3	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=2/512, ttl=64 (reply in 2
22	s1-eth3	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=2/512, ttl=64 (request in
23	s2-eth2	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=2/512, ttl=64 (reply in 2
24	s2-eth2	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=2/512, ttl=64 (request in
25	s3-eth3	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=2/512, ttl=64 (reply in 2
26	s3-eth3	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=2/512, ttl=64 (request in
27	s1-eth1	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=3/768, ttl=64 (reply in 2
28	s1-eth1	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=3/768, ttl=64 (request in
29	s3-eth2	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=3/768, ttl=64 (reply in 3
30	s3-eth2	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=3/768, ttl=64 (request in
31	s2-eth1	ICMP	10.0.0.1			10.0.0.4			Echo (ping) request id=0x2528, seq=3/768, ttl=64 (reply in 3
32	s2-eth1	ICMP	10.0.0.4			10.0.0.1			Echo (ping) reply id=0x2528, seq=3/768, ttl=64 (request in

Figure 23: Captura de Wireshark de todas las interfaces de la topología

En el log del controlador de POX se puede ver como los mensajes pasan por todos los switches entre `h1` y `h4`.

```
python3 ./pox.py forwarding.l2_learning firewall
POX 0.9.0 (ichthyosaur) / Copyright 2011-2023 James McCauley, et al.
INFO:firewall:Enabling Firewall Module
INFO:core:POX 0.9.0 (ichthyosaur) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
INFO:firewall:Firewall rules installed on 00-00-00-00-00-01
INFO:openflow.of_01:[00-00-00-00-00-03 2] connected
INFO:openflow.of_01:[00-00-00-00-00-02 3] connected
INFO:firewall:[SWITCH:1] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:1] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4: DST:10.0.0.1: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.4: DST:10.0.0.1: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4: DST:10.0.0.1: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.4: DST:10.0.0.1: PROTOCOL:ICMP
INFO:firewall:[SWITCH:1] SRC:10.0.0.4: DST:10.0.0.1: PROTOCOL:ICMP
INFO:firewall:[SWITCH:1] SRC:10.0.0.4: DST:10.0.0.1: PROTOCOL:ICMP
INFO:firewall:[SWITCH:1] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:1] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:2] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
INFO:firewall:[SWITCH:3] SRC:10.0.0.1: DST:10.0.0.4: PROTOCOL:ICMP
```

Figure 24: Logs del controlador

En `mininet` confirmamos que todos los paquetes fueron transmitidos, sin ningún tipo de perdida.

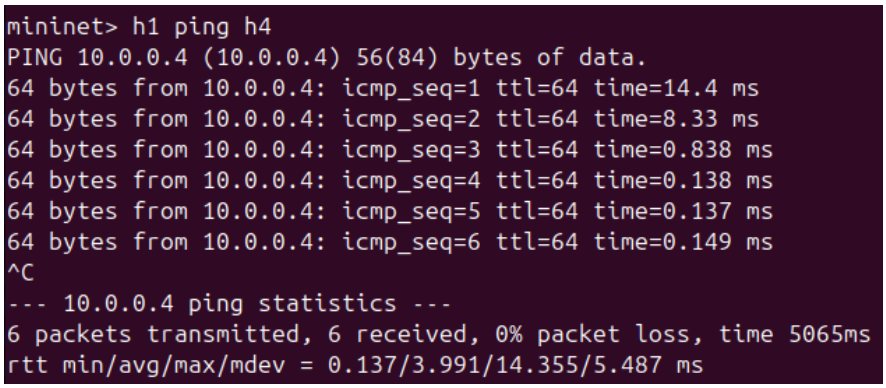


Figure 25: Captura de Mininet

5 Preguntas a responder

5.1 ¿Cuál es la diferencia entre un switch y un router? ¿Qué tienen en común?

El switch conecta dispositivos dentro de una misma red local (LAN), reenviando paquetes en función de las direcciones MAC. Construye una tabla de direcciones para saber por qué puerto enviar cada paquete, optimizando así la comunicación en una red.

En cambio, el router envía paquetes entre redes distintas. Utiliza direcciones IP para determinar la mejor ruta hacia el destino final. Los routers permiten la comunicación entre redes locales e Internet, tomando decisiones de enrutamiento basadas en tablas que pueden incluir métricas como distancia, coste o prioridad.

Ambos dispositivos comparten la tarea de interconectar hosts y direccionar tráfico, pero lo hacen en distintos niveles. Además, en escenarios modernos —como en arquitecturas SDN— las funciones de ambos pueden gestionarse de forma centralizada.

5.2 ¿Cuál es la diferencia entre un switch convencional y un switch OpenFlow?

La principal diferencia radica en el nivel de control y personalización del comportamiento del dispositivo. Un switch convencional funciona de forma autónoma, gestionando el reenvío de tramas a partir de una tabla de direcciones MAC aprendidas dinámicamente. Su lógica de funcionamiento está definida por el firmware, lo que limita su flexibilidad a la hora de modificar políticas de red.

Por el contrario, un switch OpenFlow se integra dentro del paradigma SDN (Software Defined Networking), donde el plano de control está separado del plano de datos. En este caso, el switch no toma decisiones por sí mismo, sino que obedece instrucciones enviadas desde un controlador externo, usando el protocolo OpenFlow. Esto permite implementar reglas específicas para cada flujo, como redirección, bloqueo o modificación de paquetes, en tiempo real.

Esta capacidad hace que los switches OpenFlow sean ideales para redes dinámicas, donde se requiere control centralizado, como políticas de firewall, priorización de tráfico o configuraciones personalizadas. En el trabajo realizado, por ejemplo, se utilizaron estas características para implementar un firewall configurable desde un archivo JSON, algo difícil de lograr con un switch tradicional.

5.3 ¿Se pueden reemplazar todos los routers de la Internet por switches OpenFlow?

No es factible reemplazar todos los routers de Internet por switches OpenFlow, especialmente en el contexto del enrutamiento entre sistemas autónomos (inter-AS). Si bien los switches OpenFlow ofrecen un alto nivel de flexibilidad y control dentro de redes locales o dentro de un sistema autónomo, su arquitectura no está diseñada para lidiar con la complejidad del enrutamiento global.

En particular, los routers inter-AS utilizan protocolos como BGP (Border Gateway Protocol), fundamentales para la comunicación entre organizaciones y proveedores de servicios. Estos protocolos no solo

determinan rutas, sino que también aplican políticas de acceso, acuerdos económicos y otras reglas que son difíciles de replicar con switches OpenFlow, los cuales están más orientados al control de flujo que al enrutamiento a gran escala.

Además, los routers tradicionales están optimizados para manejar grandes volúmenes de tráfico y tablas de enrutamiento extensas, lo cual es crítico para la estabilidad de Internet. Si bien es posible que dentro de un sistema autónomo se utilicen switches OpenFlow para el control interno del tráfico, la sustitución total de los routers por este tipo de dispositivos no es viable en el escenario actual de la red global.

6 Dificultades encontradas

Durante el desarrollo del trabajo, se presentaron varias dificultades que detallamos a continuación:

- Fue necesario retomar y profundizar en el uso de Mininet, ya que al no haberlo utilizado recientemente, fue complicado comprender su funcionamiento interno y la correcta configuración de topologías dinámicas para la simulación de la red.
- Se detectaron problemas de compatibilidad y estabilidad al ejecutar Mininet en ciertos entornos operativos, lo que ocasionaba fallos y desconexiones inesperadas entre los hosts y switches. Para resolverlo, optamos por realizar las pruebas en Linux nativo o en una máquina virtual bajo VirtualBox, garantizando así un entorno más controlado y fiable.
- La instalación y configuración del controlador POX también presentó dificultades, ya que su documentación es limitada y no cuenta con un instalador tradicional. Esto requirió dedicar tiempo a preparar el entorno adecuado y resolver dependencias para poder ejecutar el controlador correctamente. Además, comprender su arquitectura modular y modelo de eventos fue un desafío importante para implementar el firewall y la integración con la topología.

7 Conclusión

En este trabajo práctico, logramos implementar y comprender de manera aplicada los conceptos fundamentales de las Redes Definidas por Software (SDN) utilizando el protocolo OpenFlow. Pudimos diseñar una topología de red configurable con Mininet y configurar un controlador POX para gestionar el tráfico mediante un firewall programable.

La implementación del firewall basado en reglas definidas externamente en un archivo JSON facilitó la flexibilidad y el manejo dinámico de las políticas de seguridad, permitiendo bloquear tráfico específico sin necesidad de modificar el código fuente. Esto demuestra cómo SDN y OpenFlow simplifican la administración y el control centralizado de la red, brindando mayor agilidad y escalabilidad.

Además, el análisis del tráfico con herramientas como Wireshark nos permitió validar el correcto funcionamiento de las reglas, observando cómo los paquetes eran filtrados en tiempo real. La experiencia adquirida afianzó nuestra comprensión práctica de cómo las tecnologías SDN pueden aplicarse para resolver problemas reales de control y seguridad en redes modernas.