.UBAfiuba
FACULTAD DE INGENIERÍA

Introducción a los Sistemas Distribuidos
(75.43) Curso Hamelin

Trabajo Práctico 2
Software-Defined Networks

26 de junio de 2025

Leticia Figueroa
110510

Andrea Figueroa
110450

Josue Martel
110696

Jesabel Pugliese
110860

Kevin Vallejo
109975

Índice

1. Introducción

Las *Software-Defined Networks* (*S.D.N.*) han sido el tema dominante de la investigación y la innovación en Internet desde su aparición en 2008. Su irrupción en escena surgió de la necesidad de poder flexibilizar el uso del hardware para poder satisfacer las nuevas demandas de Internet.

El presente documento abarca la entrega y exposición de este segundo trabajo práctico grupal para la materia Introducción a los Sistemas Distribuidos (75.43/75.33/95.60) donde nuestros objetivos son:

- Familiarizarnos con los desafíos por los cuales surgen las *S.D.N.* y el protocolo *OpenFlow*.
- Aprender a controlar el funcionamiento de los switches a través de una API

Este informe se compone tanto por las respuestas a las preguntas teóricas encargadas en la consigna como por el detalle del software desarrollado para implementar una **topología dinámica** que use **OpenFlow** y un Firewall de capa de enlace.

Las herramientas fundamentales que se emplearon para asegurar el correcto funcionamiento de la implementación presentada son:

- *Mininet*: para la creación y simulación de topologías de red dinámicas sobre las cuales se aplicaron las políticas de control de flujo mediante un controlador OpenFlow.
- *POX*: controlador *SDN* que proporciona nos proporcionó una API en Python, que nos permite programar el comportamiento de los *switches OpenFlow*. En este trabajo, se utiliza este para desarrollar un módulo de firewall de capa 2 que bloquea o permite tráfico según reglas definidas externamente en un archivo `.json`.
- *Wireshark*: herramienta de análisis de tráfico de red, utilizada para verificar visualmente la transmisión o el bloqueo de paquetes a través de la red, y para corroborar el cumplimiento de las políticas del firewall en tiempo real.
- *iperf*: herramienta para la medición del rendimiento de red. Se emplea para generar tráfico entre hosts virtuales y comprobar el funcionamiento del firewall frente a distintos tipos de flujos (por ejemplo, UDP hacia un puerto específico).

El desarrollo del trabajo se enfoca en el diseño e implementación de una red emulada con switches en cadena y controladores externos, con el objetivo de establecer una infraestructura *SDN* funcional, testeable y fácilmente extensible a nuevas políticas definidas por software.

2. Conceptos previos

Los siguientes conceptos se tuvieron en cuenta como base teórica para la implementacipon presentada:

- **Flujo**: Secuencia de paquetes que comparten características en común:
  1. Dirección IP de origen.
  2. Dirección IP de destino.
  3. Puerto de origen.
  4. Puerto de destino.

  En particular, un flujo en Openflow cuenta también con los siguiente campos:
  1. Puerto físico o lógico de entrada de switch.
  2. Dirección MAC de origen
  3. Dirección MAC de destino

4. Tipo de protocolo Ethernet (por ejemplo IPv4)
5. Protocolo de nivel de red (por ejemplo TCP)
6. ID de red virtual (si aplica)

Openflow permite individualizar cada flujo por medio de la lectura de estos 10 campos

- **IP Blackholing**: Técnica para descartar automáticamente todo el tráfico asociado a una IP dado un ataque (sin alguna notificación a este mismo)

3. Supuestos

La presente entrega para garantizar el correcto y esperable funcionamiento de lo desarrollado toma como supuestos los siguientes puntos:

- El ambiente en el que el usuario ejecuta el software desarrollado tiene sistema operativo basado en Linux
- Las reglas que un nuevo usuario quiera introducir deben cumplir con el formato definido.
- El usuario cuenta con las dependencias: Docker, OVS y POX (ver README)

4. Implementación

4.1. Topología de red

Respecto al software desarrollado para la generación de topologías de red parametrizables: Se hizo uso de la herramienta mininet misma que nos proporciona una interfaz con la cual pudimos interactuar en Python para precisamente proporcionar un programa que levante topologías virtuales con la cantidad de switches en el medio parametrizada y dos hosts en los extremos de la topología lineal.

Siendo posible la parmetrización requerida a partir de los argumentos del programa:

```
class Topology(Topo):
    def build(self, *args, **params):
        switches_count = params.get('switches_count', 2)
        ...
        switches = []
        for i in range(switches_count):
            switches.append(self.addSwitch("s%s" % (i + 1)))
        ...
```

Por ejemplo, sea $n = 4$ el valor elegido para la cantidad de switches, la topología resultante es la siguiente:

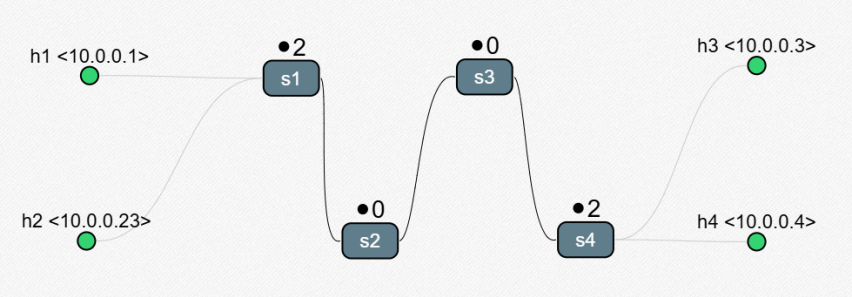

Figura 1: Ejemplo de Topología lineal con 4 switches. Visualización disponible en Mininet Topology Visualizer

Una vez levantada la topología podemos verificar el correcto funcionamiento de la red mediante el comando pingall ejecutándolo en el entorno de mininet:

```
$ mininet> pingall
```

```
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
host1 -> host2 host3 host4
host2 -> host1 host3 host4
host3 -> host1 host2 host4
host4 -> host1 host2 host3
*** Results: 0% dropped (12/12 received)
```

Figura 2: Output de pingall sobre la topología

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0178, seq=1/256, ttl=64 (reply in 2)
2	0.006129539	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0178, seq=1/256, ttl=64 (request in 1)
3	0.007888021	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=0x0179, seq=1/256, ttl=64 (reply in 4)
4	0.026064671	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0179, seq=1/256, ttl=64 (request in 3)
5	0.027611315	10.0.0.1	10.0.0.4	ICMP	98	Echo (ping) request id=0x017a, seq=1/256, ttl=64 (reply in 6)
6	0.045601890	10.0.0.4	10.0.0.1	ICMP	98	Echo (ping) reply id=0x017a, seq=1/256, ttl=64 (request in 5)
7	0.048141862	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) request id=0x017b, seq=1/256, ttl=64 (reply in 8)
8	0.048152235	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) reply id=0x017b, seq=1/256, ttl=64 (request in 7)
9	0.004655278	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) request id=0x0178, seq=1/256, ttl=64 (reply in 10)
10	0.004667767	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) reply id=0x0178, seq=1/256, ttl=64 (request in 9)
11	0.047130685	10.0.0.2	10.0.0.1	ICMP	98	Echo (ping) request id=0x017b, seq=1/256, ttl=64 (reply in 12)
12	0.049540678	10.0.0.1	10.0.0.2	ICMP	98	Echo (ping) reply id=0x017b, seq=1/256, ttl=64 (request in 11)
13	0.050904336	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x017c, seq=1/256, ttl=64 (reply in 14)
14	0.069690377	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) reply id=0x017c, seq=1/256, ttl=64 (request in 13)
15	0.071066310	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) request id=0x017d, seq=1/256, ttl=64 (reply in 16)
16	0.089457584	10.0.0.4	10.0.0.2	ICMP	98	Echo (ping) reply id=0x017d, seq=1/256, ttl=64 (request in 15)
17	0.036419639	10.0.0.1	10.0.0.4	ICMP	98	Echo (ping) request id=0x017a, seq=1/256, ttl=64 (reply in 18)
18	0.036432592	10.0.0.4	10.0.0.1	ICMP	98	Echo (ping) reply id=0x017a, seq=1/256, ttl=64 (request in 17)
19	0.080069053	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) request id=0x017d, seq=1/256, ttl=64 (reply in 20)
20	0.080080015	10.0.0.4	10.0.0.2	ICMP	98	Echo (ping) reply id=0x017d, seq=1/256, ttl=64 (request in 19)
21	0.131458522	10.0.0.3	10.0.0.4	ICMP	98	Echo (ping) request id=0x0180, seq=1/256, ttl=64 (reply in 22)

Figura 3: Pingall sobre la topología capturado con Wireshark

Además, para cumplir con el requerimiento de que los switches aprendan las direcciones MAC dinámicamente la forma de ejecutar POX debe ser tal que cargue el módulo *forwarding.l2_lerning* (*módulo de aprendizaje de capa 2- enlace)* como parte de la **lógica del controlador** (además del firewall implementado y otras configuraciones adicionales). Ejecutándose POX de la siguiente forma:

```
$ cd pox
$ python pox.py forwarding.l2_learning firewall
```

4.2. Firewall

Utilizamos la herramienta **pox** para el control e implementación de reglas para la aplicación del firewall sobre nuestra topología.

Definimos las reglas en un archivo de configuración *json*, cumpliendo el siguiente formato:

```
{
  "rules": [
    {
      "name":"Nombre de la regla",
      "src_mac": "00:00:00:00:00:01",
      "dst_mac": "00:00:00:00:00:02"
      "dst_port": 1234,
      "transport_protocol": "UDP",
      "ip_version": "IPV4"
    },
    ...
  ]
}
```

Cada regla y sus campos definen qué paquetes serán descartados por el firewall en un switch.

Además tuvimos en cuenta que, así sea en un futuro lejano, pueden aparecer nuevas versiones del protocolo IP e incluso nuevos protocolos de transporte, por lo que será posible agregar éstos apenas sean definidos.

```
def __init__(self):
    self.ofp_rules = [] # Reglas a aplicar
...
    self.protocols = [
    (UDP_UPPER, pkt.ipv4.UDP_PROTOCOL),
    (TCP_UPPER, pkt.ipv4.TCP_PROTOCOL)
    ]
    self.ip_versions = [
    (IPV4_UPPER, pkt.ethernet.IP_TYPE),
    (IPV6_UPPER, pkt.ethernet.IPV6_TYPE)
    ]
...
```

Las reglas en el archivo de configuración pueden **no especificar un protocolo IP o un protocolo de transporte**, lo cuál significa que deberían poder aplicarse para cualquiera se éstos. Es por ello que, dadas las funciones provistas por **pox**, aplicamos dichas reglas explícitamente para cada protocolo IP y cada protocolo de transporte posible.

Con este paso previo, podemos proceder a aplicar las reglas de nuestro firewall:

```

def _handle_ConnectionUp(self, event):
    for rule in self.ofp_rules:
        event.connection.send(rule)
```

5. Pruebas

5.1. Configuracion inicial

En principio, instalamos OVS (*Open vSwitch*) en Linux. Luego nos paramos en el directorio donde tenemos el `Dockerfile` para abrir un contenedor de docker con la configuración necesaria para el correcto funcionamiento del programa. Entre las configuraciones se encuentran la instalación de `python2`, mininet, iperf y el controlador POX, entre otras dependencias. A su vez, copiamos al contenedor los archivos necesarios para la ejecución de la topología y correr el firewall con las reglas.

```
docker build -t firewall-mininet .
```

```
docker run -it --rm --privileged --net=host firewall-mininet
```

Una vez dentro del contenedor, ingresamos a la dirección donde se encuentra guardado el programa:

```
cd /pox
```

Y corremos el firewall configurando los switches para que sean *self-learning*:

```
python pox.py forwarding.l2_learning firewall
```

Una vez hecho esto, podemos ver los logs del programa impresos en el contenedor de docker. Estos indican que se procesaron todas las reglas exitosamente.

```
POX 0.6.0 (fangtooth) / Copyright 2011-2018 James McCauley, et al.
INFO:firewall:Enabling Firewall Module
INFO:firewall:Setting up rules
INFO:firewall:Processing rule: Se deben descartar todos los mensajes cuyo puerto
    destino sea 80.
INFO:firewall:Processing rule: Se deben descartar todos los mensajes que provengan
    del host 1, tengan como puerto destino el 5001, y esten utilizando el protocolo
     UDP
INFO:firewall:Processing rule: Dos hosts no se pueden comunicar.
INFO:firewall:Processing rule: Dos hosts no se pueden comunicar.
INFO:core:POX 0.6.0 (fangtooth) is up.
```

Por otro lado, abrimos una segunda terminal para ejecutar la topología con mininet aplicando el firewall:

```
docker run -it --rm   --privileged   --net=host   -v /var/run/openvswitch:/var/run/
    openvswitch   firewall-mininet
```

```
mn --custom /pox/ext/topology.py --topo topologia,switches_count=3 --mac --arp --
    switch ovsk --controller remote
```

Una vez en la CLI de mininet, comprobamos que la topología fue cargada exitosamente:

```
mininet> links
host1-eth0<->s1-eth2 (OK OK)
host2-eth0<->s1-eth3 (OK OK)
host3-eth0<->s3-eth2 (OK OK)
host4-eth0<->s3-eth3 (OK OK)
s1-eth1<->s2-eth1 (OK OK)
s2-eth2<->s3-eth1 (OK OK)
```

Siendo las IP's de cada host:

```
<Host host1: host1-eth0:10.0.0.1 pid=25>
<Host host2: host2-eth0:10.0.0.2 pid=27>
<Host host3: host3-eth0:10.0.0.3 pid=29>
<Host host4: host4-eth0:10.0.0.4 pid=31>
```

Y observamos en el contenedor:

```
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:firewall:ConnectionUp for switch 00-00-00-00-00-01:
INFO:openflow.of_01:[00-00-00-00-00-02 4] connected
INFO:firewall:ConnectionUp for switch 00-00-00-00-00-02:
INFO:openflow.of_01:[00-00-00-00-00-03 3] connected
INFO:firewall:ConnectionUp for switch 00-00-00-00-00-03:
```

5.2. Ejecución de las pruebas

Utilizamos iperf para simular el envió de paquetes en la red.

Para que un host `B` comience a escuchar en un puerto `PORT` con el protocolo UDP (flag `-u`, en caso de no estar se enviara por TCP), y luego enviar paquetes de un host `A` hacia un host `B` ejecutamos:

```
mininet> hostB iperf -u -s -p PORT &
------------------------------------------------------------
Server listening on UDP port PORT
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
mininet> hostA iperf -u -c hostB -p PORT
------------------------------------------------------------
Client connecting to <IP_hostB>, UDP port 80
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local <IP_hostA> port 57140 connected with <IP_hostB> port PORT
[  3] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[  3] Sent 892 datagrams
```

En caso de que a los paquetes a enviar no los restrinja ninguna regla, estos se enviarán exitosamente y podrán ser vistos en Wireshark desde cualquier interfaz de cualquier switch:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	fe80::4411…	ff02::fb	MDNS	203	Standard query 0x0000 P
3	12.863111986	10.0.0.1	10.0.0.4	UDP	1512	35690 → 12345 Len=1470
4	12.864715349	10.0.0.1	10.0.0.4	UDP	1512	35690 → 12345 Len=1470
5	12.869353876	10.0.0.1	10.0.0.4	UDP	1512	35690 → 12345 Len=1470
6	12.880858246	10.0.0.1	10.0.0.4	UDP	1512	35690 → 12345 Len=1470
7	12.891716191	10.0.0.1	10.0.0.4	UDP	1512	35690 → 12345 Len=1470
8	12.910763797	10.0.0.1	10.0.0.4	UDP	1512	35690 → 12345 Len=1470
9	12.915144700	10.0.0.1	10.0.0.4	UDP	1512	35690 → 12345 Len=1470
10	12.926087464	10.0.0.1	10.0.0.4	UDP	1512	35690 → 12345 Len=1470

Figura 4: Captura de Wireshark de paquetes enviados exitosamente.

5.2.1. Comunicación de un host a otro sin que cumpla una regla del firewall

Probamos el caso de enviar paquetes al puerto 12345 utilizando el protocolo de transporte UDP.

```
host4 iperf -u -s -p 12345 &
host1 iperf -u -c host4 -p 12345
```

Como en nuestro firewall no tenemos reglas de restricción que se apliquen a este caso, entonces podemos ver en Wireshark los paquetes enviándose.

5.2.2. Descartar mensajes con puerto destino 80

UDP:

```
host4 iperf -u -s -p 80 &
host1 iperf -u -c host4 -p 80
```

En este caso Wireshark no muestra paquetes enviándose a través de la interfaz del switch 3 (`s3-eth3`), pero sí podemos verlos en la interfaz conectada al host desde donde se envían (`s1-eth3`).

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.4	UDP	1512	53167 → 80 Len=1470
2	0.011357706	10.0.0.1	10.0.0.4	UDP	1512	53167 → 80 Len=1470
3	0.022452969	10.0.0.1	10.0.0.4	UDP	1512	53167 → 80 Len=1470
4	0.033729042	10.0.0.1	10.0.0.4	UDP	1512	53167 → 80 Len=1470
5	0.044950212	10.0.0.1	10.0.0.4	UDP	1512	53167 → 80 Len=1470
6	0.056120537	10.0.0.1	10.0.0.4	UDP	1512	53167 → 80 Len=1470
7	0.067339132	10.0.0.1	10.0.0.4	UDP	1512	53167 → 80 Len=1470
8	0.078554841	10.0.0.1	10.0.0.4	UDP	1512	53167 → 80 Len=1470
9	0.089768407	10.0.0.1	10.0.0.4	UDP	1512	53167 → 80 Len=1470
10	0.101003052	10.0.0.1	10.0.0.4	UDP	1512	53167 → 80 Len=1470

Figura 5: Captura de Wireshark en la interfaz `s1-eth2`.

TCP:

```
host4 iperf -s -p 80 &
host1 iperf -c host4 -p 80
```

En este caso podemos ver cómo la interfaz conectada al switch que envía los paquetes intenta enviar paquetes TCP. La regla del firewall se aplica en este caso por lo que, al enviar el primer mensaje del handshake, nunca recibe respuesta (*timeout*). Luego de retransmitirlo múltiples veces deja de enviar mensajes.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.4	TCP	74	47694 → 80 [SYN] Seq=0
2	1.019145847	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission]
3	2.043149001	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission]
4	3.067133013	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission]
5	4.091132329	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission]
6	5.115148019	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission]
7	7.163152293	10.0.0.1	10.0.0.4	TCP	74	[TCP Retransmission]

Figura 6: Captura de Wireshark en la interfaz `s3-eth2`.

5.2.3. Descartar todos los mensajes que provengan del host 1, tengan como puerto destino el 5001, y estén utilizando el protocolo UDP

```
host4 iperf -u -s -p 5001 &
host1 iperf -u -c host4 -p 5001
```

Podemos ver en la captura de Wireshark cómo los paquetes se enviaron exitosamente por la interfaz del host 1 (`s1-eth2`), pero como estos paquetes son restringidos por el firewall entonces no los vemos en las interfaces de los demás switches.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	fe80::948…	ff02::2	ICMPv6	70	Router Solicitation fr
2	0.413771679	10.0.0.1	10.0.0.4	UDP	1512	52438 → 5001 Len=1470
3	0.425112197	10.0.0.1	10.0.0.4	UDP	1512	52438 → 5001 Len=1470
4	0.436257347	10.0.0.1	10.0.0.4	UDP	1512	52438 → 5001 Len=1470
5	0.447472458	10.0.0.1	10.0.0.4	UDP	1512	52438 → 5001 Len=1470
6	0.458682811	10.0.0.1	10.0.0.4	UDP	1512	52438 → 5001 Len=1470
7	0.469904424	10.0.0.1	10.0.0.4	UDP	1512	52438 → 5001 Len=1470
8	0.481123103	10.0.0.1	10.0.0.4	UDP	1512	52438 → 5001 Len=1470
9	0.492345027	10.0.0.1	10.0.0.4	UDP	1512	52438 → 5001 Len=1470
10	0.503584735	10.0.0.1	10.0.0.4	UDP	1512	52438 → 5001 Len=1470

Figura 7: Captura de Wireshark en la interfaz `s1-eth2`

5.2.4. Dos hosts no se pueden comunicar

Para aplicar esta regla elegimos los hosts: host2 y host4.

UDP:

```
host4 iperf -u -s -p 12345 &
host2 iperf -u -c host4 -p 12345
```

Los paquetes no llegan al host4, pero sí podemos ver en la captura de Wireshark que se mandan por la interfaz conectada al host2.

No.	Time	Source	Destination	Protocol	Length	Info
2	4.601721137	10.0.0.2	10.0.0.4	UDP	1512	51703 → 12345 Len=1470
3	4.613051152	10.0.0.2	10.0.0.4	UDP	1512	51703 → 12345 Len=1470
4	4.624220274	10.0.0.2	10.0.0.4	UDP	1512	51703 → 12345 Len=1470
5	4.635465970	10.0.0.2	10.0.0.4	UDP	1512	51703 → 12345 Len=1470
6	4.646658006	10.0.0.2	10.0.0.4	UDP	1512	51703 → 12345 Len=1470
7	4.657886089	10.0.0.2	10.0.0.4	UDP	1512	51703 → 12345 Len=1470
8	4.669115534	10.0.0.2	10.0.0.4	UDP	1512	51703 → 12345 Len=1470
9	4.680291299	10.0.0.2	10.0.0.4	UDP	1512	51703 → 12345 Len=1470
10	4.691495417	10.0.0.2	10.0.0.4	UDP	1512	51703 → 12345 Len=1470

Figura 8: Captura de Wireshark en una interfaz del host2

TCP:

```
host4 iperf -s -p 12345 &
host2 iperf -c host4 -p 12345
```

Utilizando TCP ocurre lo mismo que lo explicado anteriormente: vemos el fallo en los envíos de paquetes en la interfaz del host2.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.2	10.0.0.4	TCP	74	56086 → 12345 [SYN] S
2	1.041253818	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission]
3	2.065210588	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission]
4	3.089228098	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission]
5	4.113239493	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission]
6	5.138207518	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission]
7	7.185220845	10.0.0.2	10.0.0.4	TCP	74	[TCP Retransmission]

Figura 9: Captura de Wireshark en una interfaz del host2

Al aplicar la regla en la dirección opuesta obtenemos resultados similares: los hosts host2 y host4 no se pueden comunicar de ninguna manera.

6. Preguntas a responder

En esta sección responderemos detalladamente las preguntas requeridas para este trabajo.

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

   Tanto los switches como los routers almacenan (*storing*) paquetes y los reenvían (*forwarding*) por la red, pero se diferencian fundamentalmente en que los routers reenvían paquetes usando direcciones de la capa de red, mientras que los switches usan direcciones MAC, de la capa de enlace.
   Por otro lado, a diferencia de los routers, los switches son *plug-and-play*, lo que significa que funcionan automáticamente al conectarlos, sin necesidad de configuración adicional. En los routers es necesario configurar sus direcciones IP y las de los routers que se conectan a ellos.

A su vez, las tasas de filtrado (*filtering*) y reenvío en los switches son relativamente altas frente a las de los routers, esto debido a que los switches procesan paquetes solo hasta la capa de enlace, mientras que los routers llegan hasta la capa de red.
Los switches son susceptibles a tormentas de paquetes de broadcast (*storms*), ya que si un host transmite múltiples paquetes de broadcast, los switches los reenviarán a todos los puertos, lo que puede causar que la red colapse. Los routers proveen protección con *firewall* contra estas tormentas de broadcast.
A su vez, la topología con switches puede generar bucles infinitos de paquetes broadcast si hay caminos redundantes en la red, es por esto que la topología activa de la misma se restringe a un spanning tree. Para ello, se utiliza el Spanning Tree Protocol (STP), que desactiva ciertos enlaces para evitar ciclos en la red. Esto puede disminuir el rendimiento de la red, al disminuir la cantidad de enlaces activos.
Los bucles infinitos de paquetes broadcast no suelen producirse en los routers, ya que el reenvío basado en direcciones de la capa de red mediante el protocolo IP es jerárquico, gracias a la segmentación de la red en subredes. Debido a esto, la topología no necesita restringirse a un *spanning tree*, y los routers pueden utilizar el mejor camino (*path*) disponible entre el host origen y el destino. Esto representa una ventaja frente a los switches, ya que permite construir una topología de red más rica y escalable, como la que conforma Internet.
Otra ventaja de los routers frente a los switches es que una red grande que utilice switches requerirá que los hosts y routers mantengan tablas ARP de mayor tamaño, incrementando así el tráfico y el procesamiento asociados, mientras que en los routers la resolución de direcciones se limita al ámbito de cada subred, reduciendo el uso de ARP y mejorando la eficiencia.

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

   En topologías con switches, el protocolo OpenFlow es un protocolo que opera entre el controlador SDN y el switch SDN. El protocolo permite que el controlador externo programe la lógica de reenvío del switch utilizando una tabla de flujo programable que puede analizar más capas. Así, mientras los switches convencionales pueden ser solo utilizados para reenviar paquetes de la capa de enlace, los switches OpenFlow pueden inspeccionar headers tanto de la capa de enlace como la de red y la de transporte.
   Los switches OpenFlow presentan una ventaja por sobre los switches convencionales respecto a que el protocolo otorga una mayor flexibilidad y control sobre el comportamiento de la red: permite definir reglas que determinen cómo deben tratarse distintos tipos de tráfico según sus características.

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

   No, no se pueden reemplazar todos los routers de Internet por switches OpenFlow, especialmente en el escenario inter-AS (inter-domain o entre Sistemas Autónomos) por las siguientes razones:

   - **BGP sigue siendo esencial:** En Internet, la comunicación entre Sistemas Autónomos (AS) se realiza casi exclusivamente con BGP (Border Gateway Protocol). BGP maneja políticas de enrutamiento, anuncios de prefijos y relaciones comerciales entre proveedores, clientes y pares. OpenFlow no implementa directamente BGP, lo que imposibilita su uso directo en este entorno sin capas adicionales complejas.
   - **Control distribuido vs centralizado:** Internet es una red altamente distribuida con múltiples entidades administrativas (cada AS es independiente). SDN se basa en un control centralizado, lo que no se adapta fácilmente a la estructura política, técnica y administrativa de múltiples ASes independientes.
   - **Escalabilidad:** La cantidad de decisiones y rutas en la tabla de enrutamiento global de Internet es masiva (cientos de miles de rutas BGP). Coordinar esto desde un único controlador (o incluso una jerarquía de controladores SDN) sería altamente complejo y poco escalable.
   - **Resiliencia y tolerancia a fallos:** Los routers BGP pueden reconfigurarse dinámica-

mente ante fallos, sin depender de un punto central. En SDN, si el controlador falla o se desconecta, los switches pierden inteligencia, lo que podría ser catastrófico en la red de Internet.

La arquitectura de Internet está dividida en Sistemas Autónomos (los ya mencionados AS), que son dominios administrativos independientes (por ejemplo, un ISP como Telefónica, un proveedor de contenido como Google, etc.). Cada AS tiene autonomía completa sobre sus decisiones de enrutamiento, utiliza BGP para intercambiar rutas con otros AS y define políticas de enrutamiento basadas en relaciones económicas (cliente, proveedor, peer).

Un switch OpenFlow no puede replicar esta funcionalidad sin una coordinación centralizada entre ASes, lo cual rompe la independencia administrativa y la escalabilidad del modelo de Internet.

7. Dificultades encontradas

A lo largo del desarrollo del trabajo práctico nos encontramos con ciertas dificultades para cumplir con los requerimientos de la entrega. A grandes rasgos, destacamos las siguientes:

- **Problemas para ejecutar el controlador POX en nuestros distintos entornos de desarrollo:** Si bien conocíamos de antemano la existencia de este obstáculo, el manejo del versionado de POX y la implementación utilizada no siempre resultaron compatibles con los entornos en los que trabajábamos. La primera versión que probamos (la más reciente) presentó conflictos al ejecutarse con la versión recomendada de Python. Para resolver este inconveniente, optamos por utilizar una versión particular de POX compatible con Python 2.7, lo que nos permitió ejecutar el controlador sin errores en la consola.
- **Dificultades para encontrar información sobre el uso correcto de la API de POX:** Inicialmente intentamos definir las reglas de seguridad de la forma más genérica posible, evitando especificar campos particulares de los flujos OpenFlow para ampliar el dominio de aplicación de las reglas. Sin embargo, al encontrar escasa documentación sobre cómo definir reglas utilizando la API de POX, nos vimos obligados a adoptar un enfoque de prueba y error para entender las restricciones que impone el controlador al establecer reglas de seguridad.
- **Desconocimiento inicial de las funcionalidades provistas por POX**: Al comienzo del trabajo asumimos que la funcionalidad de aprendizaje de direcciones MAC debía ser implementada manualmente, debido a nuestra inexperiencia con la librería. Sin embargo, tras discutirlo y guiarnos por la intuición, decidimos investigar más a fondo y descubrimos que POX ya incluía esta capacidad integrada, lo que simplificó significativamente la implementación.

8. Conclusión

A través del uso de OpenFlow, fue posible definir políticas de ruteo mediante software, comunicándonos con un controlador central (POX) que nos permitió establecer reglas específicas sobre los flujos en función de múltiples campos (MAC, IP, puerto, protocolo, etc.). Esto nos brindó un control fino y programable sobre el comportamiento de la red.

Implementamos un firewall definido por software capaz de monitorear el tráfico entrante y saliente mediante una configuración inicial del controlador. Fue clave comprender qué servicios estaban ya implementados en POX y cómo utilizar su API para definir nuestras reglas de seguridad. Herramientas como el aprendizaje automático de MACs incorporado en el controlador y la API de OpenFlow resultaron fundamentales para este propósito.

Para validar el funcionamiento del firewall, nos apoyamos principalmente en:

- El simulador de topologías de red **Mininet**, que nos permitió construir entornos virtuales basados en OpenFlow con controladores externos (como POX con API en Python).

- El analizador de paquetes **Wireshark**, útil para inspeccionar el tráfico y verificar el cumplimiento de las reglas.

Mininet nos permitió levantar diversas topologías adaptadas a los casos de prueba, lo que facilitó la experimentación con diferentes configuraciones y políticas de seguridad.

En conclusión, las herramientas utilizadas nos permitieron implementar las reglas de seguridad propuestas por la cátedra. A pesar de algunos inconvenientes vinculados al versionado y la falta de retrocompatibilidad interna de POX, estas herramientas representan una primera línea de defensa eficaz en seguridad de red, ofreciendo la flexibilidad que exige la actual interconectividad global.

Referencias

[1] James F. Kurose and Keith W. Ross, *Computer Networking: A Top-Down Approach*, 8th Edition, Pearson, 2021.