

Trabajo Práctico 2 — Software-Defined Networks

[TA048] Redes
Curso 2

| Alumno | Número de padrón | Email |
|--------------------------|------------------|----------------------|
| Lucas Oshiro | 107024 | loshiro@fi.uba.ar |
| Martin Reimundo | 106716 | mreimundo@fi.uba.ar |
| Franco Agustin Rodriguez | 108799 | frodriguez@fi.uba.ar |
| Mateo Riat Sapulia | 106031 | mriat@fi.uba.ar |
| Ignacio Ezequiel Vetrano | 106129 | ivetrano@fi.uba.ar |

Índice

| | |
|--|-----------|
| 1. Introducción | 2 |
| 2. Hipótesis y supuestos realizados | 2 |
| 3. Implementación | 2 |
| 3.1. Topología | 2 |
| 3.1.1. Código | 2 |
| 3.1.2. Implementación | 3 |
| 3.2. Firewall | 3 |
| 3.2.1. Código | 3 |
| 3.2.2. Implementación | 4 |
| 4. Pruebas | 5 |
| 4.1. Funcionamiento normal cuando no matchea con ninguna regla | 5 |
| 4.2. Regla 1: No se aceptan mensajes al puerto 80 | 6 |
| 4.3. Regla 2: Se descartan los mensajes del host 1 con puerto de destino 5001, usando UDP . . | 7 |
| 4.4. Regla 3: No se pueden conectar el host 1 y el host 3 | 8 |
| 5. Preguntas | 9 |
| 5.1. ¿Cuál es la diferencia entre un Switch y un router? ¿Qué tienen en común? | 9 |
| 5.2. ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow? | 9 |
| 5.3. ¿Se pueden reemplazar todos los routers de la Internet por switches OpenFlow? Piense en el escenario inter-ASes para elaborar su respuesta. | 9 |
| 6. Dificultades encontradas | 10 |
| 7. Conclusion | 10 |

1. Introducción

Este trabajo práctico tiene como objetivo introducir los conceptos fundamentales detrás de SDN y OpenFlow, y familiarizarse con la programación de dispositivos de red a través de una API. Se buscará comprender cómo estos enfoques permiten un control más preciso sobre el funcionamiento de los switches y habilitan una infraestructura de red más ágil, adaptable y preparada para las necesidades de hoy en día.

2. Hipótesis y supuestos realizados

- La topología cuenta con al menos un switch.
- El firewall está configurado en un único switch, por lo que las reglas se aplicarán únicamente si el tráfico entre el host origen y el host destino pasa a través de dicho switch.
- Tanto las reglas 1 como 2 no se pueden bloquear en IPv6 debido al soporte que ofrece POX (Open-Flow 1.0, ya que no soporta matching de puertos IPv6)

3. Implementación

A continuación se mostrará la implementación realizada. Se realizaron dos implementaciones, una para la topología y otra para el firewall.

3.1. Topologia

3.1.1. Código

```
1 from mininet.topo import Topo
2
3
4 class TopoTP2 (Topo):
5     def __init__(self, cantidad_switches):
6
7         if cantidad_switches < 1:
8             print("Error: La cantidad de switches debe ser al menos 1.")
9             return
10
11         Topo.__init__(self)
12
13         # Crear hosts
14         h1 = self.addHost('h1', mac='00:00:00:00:00:01')
15         h2 = self.addHost('h2', mac='00:00:00:00:00:02')
16         h3 = self.addHost('h3', mac='00:00:00:00:00:03')
17         h4 = self.addHost('h4', mac='00:00:00:00:00:04')
18
19         # Crear switch
20         s1 = self.addSwitch('s1')
21
22         switches = [s1]
23         for i in range(2, cantidad_switches + 1):
24             s = self.addSwitch('s' + str(i))
25             switches.append(s)
26
27         # Add links between switches and hosts self . addLink ( s1 , s2 )
28         self.addLink(s1, h1)
29         self.addLink(s1, h2)
30
31         for i in range(1, cantidad_switches):
32             self.addLink(switches[i - 1], switches[i])
33
34         self.addLink(switches[-1], h3)
35         self.addLink(switches[-1], h4)
36
37
38 topos = {'customTopo': TopoTP2}
```

3.1.2. Implementación

- Nuestra topología recibe por parametro la cantidad de switches que se necesitan.
- Creamos 4 host especificandoles sus MAC address.
- Luego creamos la cantidad de switches correspondientes.
- Creamos la conexión con el primer switch y los hosts 1 y 2.
- Conectamos todos los switches y al último switch le añadimos las conexiones con los hosts 3 y 4.

3.2. Firewall

3.2.1. Código

```
1 '''
2 Coursera:
3 - Software Defined Networking (SDN) course
4 -- Programming Assignment: Layer-2 Firewall Application
5
6 Professor: Nick Feamster
7 Teaching Assistant: Arpit Gupta
8 '''
9
10 from pox.core import core
11 import pox.openflow.libopenflow_01 as of
12 from pox.lib.revent import *
13 from pox.lib.util import dpidToStr
14 from pox.lib.addresses import EthAddr
15 from pox.lib.addresses import IPAddr
16 from collections import namedtuple
17 import pox.lib.packet as pkt
18 import os
19 import json
20
21 log = core.getLogger()
22
23 DPID_FIREWALL_SWITCH = 1
24
25 NOMBRE_ARCHIVO_CONFIGURACION = os.path.join(
26     os.path.dirname(__file__),
27     "..", "config.json"
28 )
29
30
31 def obtener_dl_type(version_ip):
32     '''
33     Returns the dl_type based on the version of the protocol.
34     '''
35     if version_ip == "4":
36         return pkt.ethernet.IP_TYPE # IPv4
37     elif version_ip == "6":
38         return pkt.ethernet.IPV6_TYPE # IPv6
39     else:
40         return None
41
42
43 def cargar_reglas(nombre_archivo):
44     with open(nombre_archivo) as file:
45         config = json.load(file)
46     return config.get("reglas", [])
47
48
49 def obtener_protocolo_transporte(protocolo_transporte):
50     if protocolo_transporte == "TCP":
51         return pkt.ipv4.TCP_PROTOCOL
52     elif protocolo_transporte == "UDP":
53         return pkt.ipv4.UDP_PROTOCOL
54     else:
55         return None
56
```

```

57
58 def crear_regla_drop(dl_src=None, dl_dst=None, src_ip=None, dst_ip=None,
59                      version_ip=None, protocolo_transporte=None, dst_port=None,
60                      src_port=None):
61     log.debug("Creo regla de drop para: dl_src=%s, dl_dst=%s, version_ip=%s,
62              protocolo_transporte=%s, dst_port=%s",
63              dl_src, dl_dst, version_ip, protocolo_transporte, dst_port)
64     msg = of.ofp_flow_mod()
65     msg.match.dl_src = EthAddr(dl_src) if dl_src else None
66     msg.match.dl_dst = EthAddr(dl_dst) if dl_dst else None
67     msg.match.nw_src = IPAddr(src_ip) if src_ip else None
68     msg.match.nw_dst = IPAddr(dst_ip) if dst_ip else None
69     msg.match.dl_type = obtener_dl_type(version_ip) if version_ip else None
70     msg.match.nw_proto = obtener_protocolo_transporte(protocolo_transporte) if
71     protocolo_transporte else None
72     msg.match.tp_dst = int(dst_port) if dst_port else None
73     msg.match.tp_src = int(src_port) if src_port else None
74     msg.actions = [] # Drop
75     return msg
76
77 class Firewall(EventMixin):
78
79     def __init__(self):
80         self.listenTo(core.openflow)
81         log.debug("Enabling Firewall Module")
82
83     def _handle_ConnectionUp(self, event):
84         ''' Add your logic here ... '''
85         if event.dpid == DPID_FIREWALL_SWITCH:
86             log.info("Instalando reglas de bloqueo en s" + str(DPID_FIREWALL_SWITCH))
87
88             for regla in cargar_reglas(NOMBRE_ARCHIVO_CONFIGURACION):
89                 msg = crear_regla_drop(dl_src=regla.get("src_mac"),
90                                       dl_dst=regla.get("dst_mac"),
91                                       src_ip=regla.get("src_ip"),
92                                       dst_ip=regla.get("dst_ip"),
93                                       version_ip=regla.get("ip_version"),
94                                       protocolo_transporte=regla.get("transport_protocol
95                                       "),
96                                       dst_port=regla.get("dst_port"),
97                                       src_port=regla.get("src_port"))
98                 event.connection.send(msg)
99
100 def launch():
101     '''
102     Starting the Firewall module
103     '''
104     core.registerNew(Firewall)

```

3.2.2. Implementación

Utilizamos como base para nuestro Firewall, un archivo extraído del curso SDN de Coursera propuesto por la cátedra.

Para empezar, `_handle_ConnectionUp` carga las reglas para un switch determinado. En nuestro caso, decidimos usar el switch 1.

Las reglas se obtienen de un archivo `config.json` y creamos un mensaje por cada regla, pasándole por parámetro los campos de la regla a la función `crear_regla_drop`. Los campos posibles serán:

- `src_mac`: la dirección MAC de origen
- `dst_mac`: la dirección MAC de destino
- `src_ip`: la dirección IP de origen
- `dst_ip`: la dirección IP de destino
- `ip_version`: la versión de IP
- `transport_protocol`: el tipo de protocolo de transporte

- *src_port*: el puerto de origen
- *dst_port*: el puerto de destino

Luego, a la hora de crear el mensaje para configurar el switch, se matchean los parámetros necesarios para la regla, y en caso de no haber recibido, se asigna ese parámetro a *None*. Mediante *event.connection.send(msg)* se envía al switch para que lo configure.

4. Pruebas

En esta sección se mostrarán las pruebas realizadas. Primero que nada y con el fin de verificar el funcionamiento, se utilizó *iperf* para recibir y enviar paquetes, y *Wireshark* en el switch S2 para detectar el tráfico de los mismos. Se utilizó la siguiente topología para comprobar el funcionamiento:

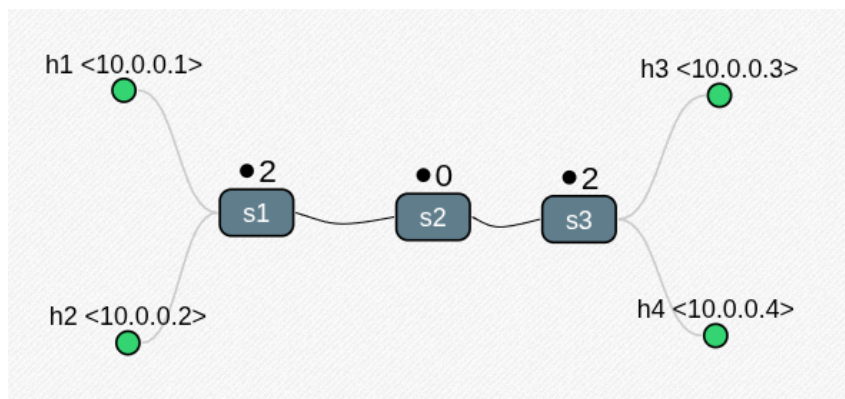


Figura 1: Topología utilizada para comprobar el funcionamiento.

4.1. Funcionamiento normal cuando no matchea con ninguna regla

Para comprobar el funcionamiento cuando no se matchea con ninguna regla, probaremos enviar mensajes desde el host 1 usando TCP al host 4 por el puerto 1000

```

"Node: h1"
root@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN# iperf -c
10.0.0.4 -p 1000
-----
Client connecting to 10.0.0.4, TCP port 1000
TCP window size: 85.3 KByte (default)
-----
[ 1] local 10.0.0.1 port 50268 connected with 10.0.0.4 port 1000
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-10.0026 sec 26.4 GBytes 22.7 Gbits/sec
-----

```

Figura 2: Regla 2 desde el host 1 (TCP)

```

"Node: h4"
root@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN# iperf -s
-p 1000
-----
Server listening on TCP port 1000
TCP window size: 85.3 KByte (default)
-----
[ 1] local 10.0.0.4 port 1000 connected with 10.0.0.1 port 50268
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-9.9987 sec 26.4 GBytes 22.7 Gbits/sec
-----

```

Figura 3: Regla 2 desde el host 4 (TCP)

| No. | Time | Source | Destination | Protocol | Source port | Destination Port | Length | Info |
|-----|----------|----------|-------------|----------|-------------|------------------|--------|------------------------------------|
| 1 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 37668 | 1000 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 2 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 1000 | 37668 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 3 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 37668 | 1000 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 4 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 1000 | 37668 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 5 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 37668 | 1000 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 6 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 1000 | 37668 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 7 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 37668 | 1000 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 8 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 1000 | 37668 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 9 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 37668 | 1000 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 10 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 1000 | 37668 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 11 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 37668 | 1000 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 12 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 1000 | 37668 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 13 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 37668 | 1000 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 14 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 1000 | 37668 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 15 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 37668 | 1000 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |

Figura 4: Wireshark para la regla 2 (TCP)

En este caso, al utilizar TCP y enviar los mensajes al puerto 1000, se observa que el host 1 envi  mensajes y recib  respuestas por parte del host 4.

4.2. Regla 1: No se aceptan mensajes al puerto 80

Se intentar  enviar un mensaje desde el host 1 al host 4 al puerto 80 utilizando TCP.

```

"Node: h1"
root@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN# iperf -c
10.0.0.4 -p 80
^C^Croot@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN#

```

Figura 5: Regla 1 desde el host 1 (TCP)

```

"Node: h4"
root@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN# iperf -s
-p 80

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

```

Figura 6: Regla 1 desde el host 4 (TCP)

| No. | Time | Source | Destination | Protocol | Source port | Destination Port | Length | Info |
|-----|----------|----------|-------------|----------|-------------|------------------|--------|------------------------------------|
| 1 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 58710 | 80 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 2 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 80 | 58710 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 3 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 58710 | 80 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 4 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 80 | 58710 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 5 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 58710 | 80 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 6 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 80 | 58710 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 7 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 58710 | 80 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 8 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 80 | 58710 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 9 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 58710 | 80 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 10 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 80 | 58710 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 11 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 58710 | 80 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 12 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 80 | 58710 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 13 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 58710 | 80 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 14 | 0.000000 | 10.0.0.4 | 10.0.0.1 | TCP | 80 | 58710 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |
| 15 | 0.000000 | 10.0.0.1 | 10.0.0.4 | TCP | 58710 | 80 | 60 | 60 [RST] Seq=154542372 Win=0 Len=0 |

Figura 7: Wireshark para la regla 1 (TCP)

Se observa como desde h1 envia un mensaje SYN, pero desde el h4 no recibe respuesta. A su vez, desde el host 4 no envia ni recibe mensajes al host 1.

Tambi n realizamos la prueba usando UDP.

```

"Node: h1"
root@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN# iperf -c
10.0.0.4 -u -p 80

-----
Client connecting to 10.0.0.4, UDP port 80
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.1 port 58710 connected with 10.0.0.4 port 80
[ ID] Interval      Transfer      Bandwidth
[ 1] 0.0000-10.0152 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 8: Regla 1 desde el host 1 (UDP)

```

"Node: h4"
root@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN# iperf -s
-u -p 80
-----
Server listening on UDP port 80
UDP buffer size: 208 KByte (default)
-----

```

Figura 9: Regla 1 desde el host 4 (UDP)

| No. | Time | Source | Destination | Protocol | Source port | Destination Port | Length | Info |
|-----|-------------|----------|-------------|----------|-------------|------------------|--------|---------------------|
| 1 | 0.000000000 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 2 | 0.011304939 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 3 | 0.011312395 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 4 | 0.022520441 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 5 | 0.033740401 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 6 | 0.044967463 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 7 | 0.056188024 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 8 | 0.067397143 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 9 | 0.078612103 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 10 | 0.089828936 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 11 | 0.101039367 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 12 | 0.112255789 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |
| 13 | 0.123468345 | 10.0.0.1 | 10.0.0.4 | UDP | 58710 | 80 | 1512 | 58710 → 80 Len=1470 |

Figura 10: Wireshark para la regla 1 (UDP)

Se observa como desde el host 1 se envían los mensajes, pero no recibe ningún ACK por parte del host 4. Desde la perspectiva del host 4, no se envían ni se reciben mensajes al host 1.

4.3. Regla 2: Se descartan los mensajes del host 1 con puerto de destino 5001, usando UDP

En esta prueba, se descartan todos los mensajes provenientes del host 1 con puerto destino 5001 y que se este usando UDP.

```

"Node: h1"
root@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN# iperf -c
10.0.0.4 -u -p 5001
-----
Client connecting to 10.0.0.4, 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 60584 connected with 10.0.0.4 port 5001
[ 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.

```

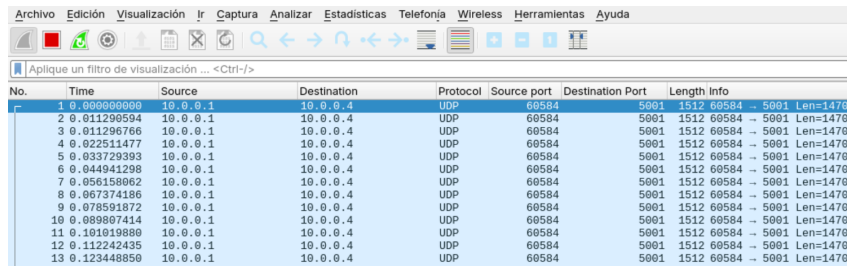
Figura 11: Regla 2 desde el host 1 (UDP)

```

"Node: h4"
root@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN# iperf -s
-u -p 5001
-----
Server listening on UDP port 5001
UDP buffer size: 208 KByte (default)
-----

```

Figura 12: Regla 2 desde el host 4 (UDP)



| No. | Time | Source | Destination | Protocol | Source port | Destination Port | Length | Info |
|-----|-------------|----------|-------------|----------|-------------|------------------|--------|-----------------------|
| 1 | 0.000000000 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 2 | 0.011290594 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 3 | 0.011296766 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 4 | 0.022511477 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 5 | 0.033729393 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 6 | 0.044941298 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 7 | 0.056158062 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 8 | 0.067374186 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 9 | 0.078591872 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 10 | 0.089807414 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 11 | 0.101019880 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 12 | 0.112242435 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |
| 13 | 0.123448850 | 10.0.0.1 | 10.0.0.4 | UDP | 60584 | 5001 | 1512 | 60584 → 5001 Len=1470 |

Figura 13: Wireshark para la regla 2 (UDP)

Podemos observar que desde el host 1 se envían los mensajes al puerto 5001 del host 4, pero no llegan ningún ACK como respuesta por parte del host 4. Desde el lado del host 4, no se realiza ninguna acción.

4.4. Regla 3: No se pueden conectar el host 1 y el host 3

Ahora probamos enviar mensajes desde el host 1 al host 3, lo cual debería ser imposible ya que los mensajes deberían ser droppeados. En este caso, usamos UDP

```

"Node: h1"
root@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN# iperf -c
10.0.0.3 -u -p 1000

-----
Client connecting to 10.0.0.3, UDP port 1000
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 1] local 10.0.0.1 port 44892 connected with 10.0.0.3 port 1000
[ ID] Interval      Transfer      Bandwidth
[ 1] 0.0000-10.0152 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 14: Regla 3 desde el host 1 (UDP)

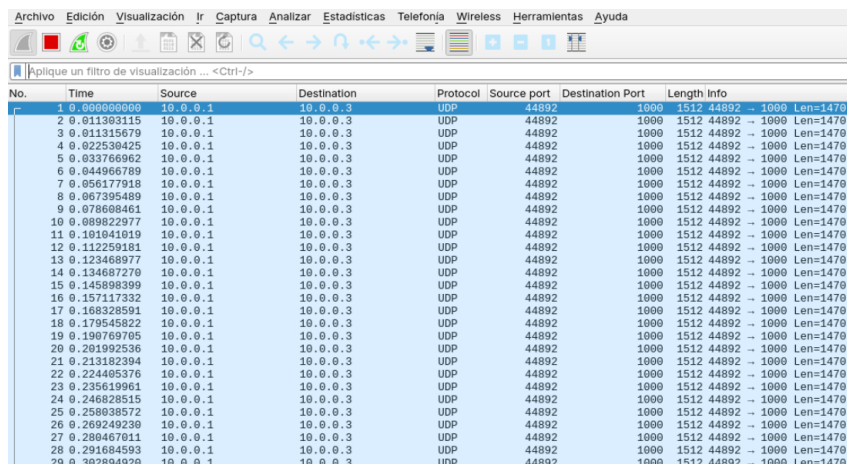
```

"Node: h3"
root@nachoguide-ESCRITORIO:/home/nachoguide/tareas/redes/tp2/TP2-SDN# iperf -s
-u -p 1000

-----
Server listening on UDP port 1000
UDP buffer size: 208 KByte (default)
-----

```

Figura 15: Regla 3 desde el host 3 (UDP)



| No. | Time | Source | Destination | Protocol | Source port | Destination Port | Length | Info |
|-----|-------------|----------|-------------|----------|-------------|------------------|--------|-----------------------|
| 1 | 0.000000000 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 2 | 0.011303115 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 3 | 0.011315679 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 4 | 0.022530425 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 5 | 0.033766962 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 6 | 0.044966789 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 7 | 0.056177918 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 8 | 0.067395489 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 9 | 0.078608461 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 10 | 0.089822977 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 11 | 0.101041019 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 12 | 0.112259181 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 13 | 0.123468977 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 14 | 0.134687270 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 15 | 0.145898399 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 16 | 0.157117332 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 17 | 0.168328591 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 18 | 0.179545822 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 19 | 0.190769705 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 20 | 0.201992536 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 21 | 0.213182394 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 22 | 0.224405376 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 23 | 0.235619961 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 24 | 0.246829515 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 25 | 0.258038572 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 26 | 0.269249230 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 27 | 0.280467811 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 28 | 0.291684593 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |
| 29 | 0.302894920 | 10.0.0.1 | 10.0.0.3 | UDP | 44892 | 1000 | 1512 | 44892 → 1000 Len=1470 |

Figura 16: Wireshark para la regla 3 (UDP)

Como resultado de esta prueba, el host 1 envió mensajes al host 3, pero no le llegó respuestas por parte del host 3.

5. Preguntas

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

La diferencia principal entre un Switch y un Router radica en la capa del modelo de red en la que operan y el tipo de dirección que utilizan para reenviar los datos. Un Switch opera principalmente en la capa de enlace y utiliza direcciones MAC (direcciones de enlace) para el reenvío, mientras que un Router opera en la capa de red y sus decisiones de reenvío se basan en direcciones IP (direcciones de red). Tanto los switches como los routers comparten la función de conmutar paquetes, es decir, trasladar datos de un origen a un destino dentro de una red. Esta similitud funcional implica que ambos dispositivos deben enfrentar problemas similares, como la congestión de red.

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

La diferencia fundamental entre un switch convencional y un switch OpenFlow radica en cómo toman decisiones de reenvío y en su arquitectura de control. Un switch convencional opera de forma autónoma, reenviando tramas según la dirección MAC de destino. Aprende automáticamente estas direcciones observando el tráfico entrante y construye sus tablas de reenvío sin intervención externa. Su funcionalidad está determinada por el hardware y software del fabricante, lo que hace que tenga un comportamiento fijo y poco flexible. En cambio, un switch OpenFlow forma parte de una red basada en SDN, donde el plano de control está separado del plano de datos. Las decisiones de reenvío no las toma el switch por sí solo, sino que son definidas por un controlador SDN remoto que instala reglas en las llamadas tablas de flujo. Estas reglas pueden usar criterios de múltiples capas del protocolo (como direcciones IP, puertos TCP/UDP, tipo de protocolo, etc.), siguiendo un modelo "match plus action". Además de reenviar paquetes, un switch OpenFlow puede realizar acciones más complejas como reescritura de encabezados, bloqueo de tráfico, o redirección a servidores externos. Esta arquitectura permite una red mucho más flexible y programable, y favorece la interoperabilidad entre dispositivos, controladores y aplicaciones de diferentes proveedores.

5.3. ¿Se pueden reemplazar todos los routers de la Internet por switches OpenFlow? Piense en el escenario inter-ASes para elaborar su respuesta.

No, no es viable reemplazar todos los routers de Internet por switches OpenFlow, especialmente si consideramos el escenario inter-AS (entre Sistemas Autónomos). Existen varias razones técnicas y organizativas que lo impiden:

- **Modelo de Control: Distribuido vs. Centralizado**

Los routers tradicionales funcionan bajo un modelo de control distribuido, donde cada uno toma decisiones de enrutamiento de manera autónoma mediante protocolos como BGP. Este enfoque permite adaptarse dinámicamente a cambios en la topología sin depender de una entidad central. En cambio, los switches OpenFlow utilizan un modelo centralizado, en el cual un controlador externo gestiona las decisiones de reenvío. Este enfoque puede ser eficiente en redes administradas por una sola organización, pero resulta difícil de escalar y mantener en un entorno global y descentralizado como Internet.

- **Escalabilidad y Autonomía en el Enrutamiento Inter-AS**

Internet está compuesta por miles de Sistemas Autónomos, como proveedores de servicios (ISPs) o grandes instituciones, que requieren autonomía en sus decisiones de enrutamiento y en su infraestructura. Cada AS puede definir sus propias políticas de enrutamiento y controlar qué información comparte. Un modelo basado en OpenFlow implicaría ceder parte de esta autonomía y compartir información con un controlador externo, lo cual va en contra de los principios de operación del sistema inter-AS.

■ Desafíos de SDN en el Entorno Inter-AS Global

Si bien SDN y OpenFlow aportan flexibilidad, su uso a gran escala (como en todo Internet) presenta grandes desafíos. Para reemplazar todos los routers actuales por switches OpenFlow, sería necesario un sistema de control central o varios controladores que trabajen en perfecta coordinación entre sí. Esto es muy difícil de lograr, ya que Internet está formado por miles de redes independientes (como los distintos ISP) que no siempre comparten información, objetivos o niveles de seguridad.

6. Dificultades encontradas

Algunas dificultades que surgieron en el avance del proyecto fueron:

- Familiarizarse con POX: Instalación y configuración
- Comprensión de conceptos como IP blackholing

7. Conclusion

Se logró desarrollar correctamente y en tiempo y forma un firewall para bloqueo de distintos tipos de paquetes configurable para una topología programable, garantizando la colaboración mutua del grupo y poniendo en práctica todos los conceptos aprendidos en el transcurso de la materia.

El presente trabajo fomentó la investigación del equipo para las dificultades planteadas en la sección previa, materializando de esta manera nuevas herramientas.

Además, resulta importante mencionar que la realización del trabajo fue fundamental para entender el rol que ocupan las SDN hoy y en un futuro, donde el internet ha evolucionado a escalas que no se consideraron en el diseño inicial de los protocolos. Todos estos conceptos y herramientas introdujeron flexibilidad, dinamismo y eficiencia a la hora de distribuir los paquetes, además de interoperabilidad entre dispositivos, controladores y aplicaciones de diferentes proveedores.