



Trabajo Práctico II — Software Defined Networks

[75.43] Introducción a los Sistemas Distribuidos
Segundo cuatrimestre de 2023

Alumno	Número de padrón	Email
Masri, Micaela	108814	noahmasri19@gmail.com
Ayala, Camila	107440	cayala@fi.uba.ar
Pol, Juan Manuel	108448	jpol@fi.uba.ar
Shih, Ian	108349	ishih@fi.uba.ar
Sanchez, Manuel	107951	msanchezf@fi.uba.ar

Índice

1. Introducción	2
2. Preguntas teóricas	2
2.1. ¿Cuál es la diferencia entre un Switch y un router? ¿Qué tienen en común?	2
2.2. ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow?	3
2.3. ¿Se pueden reemplazar todos los routers de la Internet por Switches OpenFlow? .	3
3. Implementación	4
3.1. Topología	4
3.2. Firewall con POX	4
3.2.1. Explicación de la implementación	4
3.2.2. Supuestos de las reglas	7
4. Pruebas	7
4.1. Testeo comando ping sin firewall	7
4.2. Testeo comando ping con firewall	9
4.3. Testeo de firewall con las reglas de la consigna	10
4.3.1. Paquetes cuyo host de destino es el 80	11
4.3.2. Paquetes cuyo host de origen es el 1, tienen como protocolo de transporte UDP y tienen puerto de destino 5001	15
4.3.3. Paquetes entre los hosts 2 y 3	17
4.3.4. Comunicación entre hosts que no deben droppearse	21
5. Conclusiones	22
6. Referencias	22

1. Introducción

En este trabajo practico desarrollamos un Software Designed Network (SDN). Para esto creamos una configuración de switches y hosts usando mininet. Un SDN se caracteriza por tener centralizado el plano de control y este se implementa por software, para lo cual creamos un controlador usando POX y usamos el algoritmo proveído por este para configurar el algoritmo a usar para que se conozcan entre sí, elegimos l2-learning. . En este puede instalarse un firewall, cuyas reglas pueden configurarse dentro de un archivo JSON.

2. Preguntas teóricas

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

Los routers son *store-and-forward packet switches*, es decir, dispositivos que almacenan paquetes y deciden a donde enviarlos luego, que utilizan las direcciones de red para hacerlo. Los switches, si bien tienen la misma funcionalidad, difieren en que utilizan las direcciones MAC. Ambos reciben un paquete y en base al destino que se encuentra en el header, compara el campo que le representa esto con las entradas de una tabla (lookup) donde se indica a que interfaz redireccionar el paquete que haga match con esa entrada. Debido a esto, los routers son switches de 3 capas, mientras que los switches son solo de 2 capas.

Los switches, a diferencia de los routers, son plug-and-play. Esto quiere decir que no es necesario configurarle tablas, sino que al instalarla las ira creando de forma automática, conociendo al resto de los dispositivos, usando el protocolo ARP (Address Resolution Protocol) a medida que los necesita. Esto los hace tentadores para los administradores de red, ya que no hay tablas que configurar. Además, tienen tasas de filtrado y enviado altas, ya que solo procesan hasta la capa 2, mientras que los routers deben procesar los paquetes hasta la capa de red.

Como no hay jerarquías entre direcciones MAC, por cada dirección MAC se requiere una entrada en la tabla. Esto hace que no sea muy escalable, ya que requerirían memorias inmensas para guardar todas las MAC addresses de la red. Además, al usar ARP para conocer las direcciones MAC, si se quisieran conocer todas las MACs de la red y crearle una entrada para cada uno se generaría un trafico inmenso. Es por esto que su uso se restringe a redes pequeñas de algunos cientos de hosts que tienen pocos segmentos LAN. Los routers, por el otro lado, pueden saber a donde mandar un paquete en toda la red y son altamente instalables.

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

Un switch convencional utiliza la MAC de destino para realizar el forwarding, e implementa en el mismo dispositivo tanto el plano de control como el plano de datos. Por el otro lado, en un switch OpenFlow se hace uso de un dispositivo llamado controlador, que implementa el plano de control, mientras que en el switch se implementa únicamente el plano de datos, y se comunican entre si usando el protocolo OpenFlow. Esto permite crear una identificación de flujos, haciendo uso de la información de las capas superiores para realizar un forwardo mas eficiente.

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

OpenFlow suele utilizarse para la comunicación dentro de un mismo *Autonomous System* (intra AS). Si se reemplazan todos los routers de la Internet por Switches OpenFlow, este protocolo tendría que soportar también el transporte de paquetes a través de distintos AS, lo que sería muy complejo por dos principales motivos:

1. **Escalabilidad:** OpenFlow y las arquitecturas de SDNs son más fáciles de implementar dentro de las AS por tener un tamaño limitado de hosts, y por ser un este un protocolo basado en control centralizado de estos hosts. Un controlador debe armar y configurar las tablas de enrutamiento en cada uno de los routers, la cual de implementarse en toda la internet tendría una cantidad de entradas imposible de computar.
2. **Traslado inter AS:** en el traslado de estos paquetes, se manejan distintas políticas de enrutamiento y tráfico de red dependiendo de la AS en la que se esta, y la coordinación y gestión de estas diversas políticas por parte de uno o varios controladores se volvería demasiado dificultoso. Internet opera con una diversidad de tecnologías y protocolos, y cambiar todo a OpenFlow podría no ser práctico ni eficiente.
3. **Migración** Internet se basa en una variedad de protocolos, normas y tecnologías muy diversas. La transición a switches OpenFlow requeriría una interoperabilidad que puede ser muy costosa e ineficiente.

Es por estos motivos que no creemos que sea óptimo implementar toda la lógica de la red con switches OpenFlow.

3. Implementación

3.1. Topología

Para poder generar la topología pedida, basándonos un poco en el ejemplo otorgado en la sección 5.1.1 en el enunciado, se hizo una función similar, pero que ahora recibía por parámetro la cantidad de switches a inicializar. A estos se los crea conectando a cada uno con el anterior (excepto el primero) y con el siguiente (excepto el ultimo), para así generar una fila de switches, tal como se puede observar en la figura 1. Luego, se creaba un numero fijo de hosts, 4, y se conectaba a los primeros dos (h1 y h2) con el primer switch (s1), y a los últimos dos (h3 y h4) con el ultimo switch (sn).

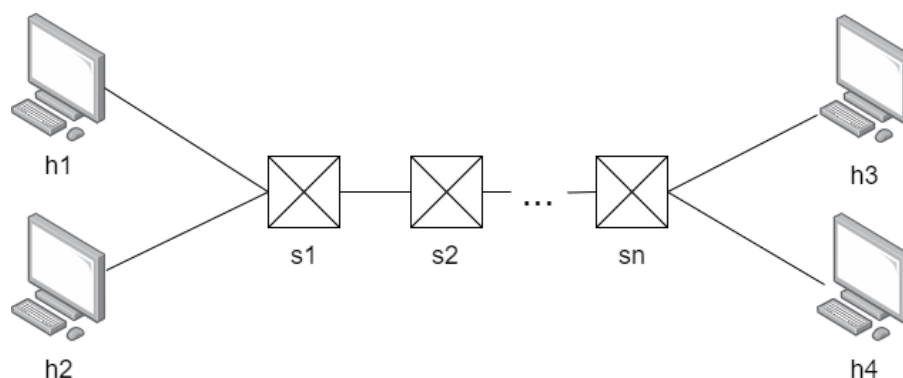


Figura 1: Topología de la red planteada

Este archivo puede encontrarse bajo el nombre *topologia.py*, usando el lenguaje Python3.

3.2. Firewall con POX

3.2.1. Explicación de la implementación

Este se puede encontrar dentro de la carpeta ext de la carpeta pox, siguiendo con las [instrucciones de desarrollo](#). A la hora de crear el firewall, debimos investigar en primer lugar como funcionaban las reglas, como se las creaba, y en segundo lugar, como podríamos hallar el switch al que queríamos aplicarle estas reglas. Se esperaba que todos los switches aprendieran la topología con l2 learning, pero que a solo uno se le agregasen ciertas condiciones extras para cumplir con su rol de firewall. La documentación de POX es altamente detallada y clara, por lo que cumplir con el aprendizaje automatizado fue muy sencillo. Al correr el controlador, simplemente debíamos correr también el modulo l2_learning que se puede encontrar dentro de la carpeta forwarding ([referencia de l2 learning](#)).

Hallamos luego el evento que se disparaba cuando se creaba una conexión con un switch ([evento connection up](#)), y decidimos aprovechar ese momento para aplicarle al switch que haría de firewall las reglas planteadas por el usuario. Notamos que cada evento tenía un atributo “dpid”, datapath ID, que nos permitía identificar que switch había ocasionado el evento, y de esta manera ver si se debía convertir al switch en un firewall o no. Los dpids notábamos que iban desde 1 hasta n, siendo n la cantidad de switches en la subnet. Tras algunas pruebas mas, notamos que la numeración era igual a la observada en la figura 1 previamente mencionada, por lo que sabíamos si había que aplicarle las reglas al switch o no basándonos en si su dpid coincidía con el switch que el usuario deseaba que fuese el firewall.

Se halló a su vez como instalar una entrada en la tabla, y eso fue lo que se usó para aplicar reglas. Se notó que, o bien se puede especificar una acción para los mensajes de cierto tipo, como podría ser redirigir la salida a cierto puerto específico, también se podía no especificar ninguna y que se interprete esto como que se debe descartar el paquete.

También existía otro evento como el PacketIn ([evento PacketIn](#)) sin embargo nos dimos cuenta que era poco eficiente, ya que por cada paquete el switch tenía que consultar con el controlador si necesitaba descartar el paquete, esta consulta termina siendo muy costosa ya que la comunicación entre el switch y controlador tarda mucho mas que la verificación local del paquete en el switch con las reglas aplicadas.

Attribute	Meaning
in_port	Switch port number the packet arrived on
dl_src	Ethernet source address
dl_dst	Ethernet destination address
dl_vlan	VLAN ID
dl_vlan_pcp	VLAN priority
dl_type	Ethertype / length (e.g. 0x0800 = IPv4)
nw_tos	IP TOS/DS bits
nw_proto	IP protocol (e.g., 6 = TCP) or lower 8 bits of ARP opcode
nw_src	IP source address
nw_dst	IP destination address
tp_src	TCP/UDP source port
tp_dst	TCP/UDP destination port

Figura 2: Significado de cada campo de un match

Para crear una regla, se debía usar una estructura del tipo match, y enviársela al switch. Los campos con los que cuenta esta estructura se pueden ver en la figura 2. Dado que los nombres no

parecen ser de lo mas claro, se creo una estructura propia muy similar llamada Rule.

```
1 class Rule():
2     def __init__(self, rule_options):
3         self.dest_port = rule_options.get('dest_port', None)
4         self.origin_port = rule_options.get('origin_port', None)
5         self.dest_ip = (IPAddr(rule_options['dest_ip'][0]),
6                         rule_options['dest_ip'][1]) if 'dest_ip' in rule_options else
7                         None
8         self.src_ip = (IPAddr(rule_options['src_ip'][0]),
9                        rule_options['src_ip'][1]) if 'src_ip' in rule_options else
10                        None
11         self.dest_mac = EthAddr(rule_options['dest_mac']) if 'dest_mac'
12                             in rule_options else None
13         self.src_mac = EthAddr(rule_options['src_mac']) if 'src_mac' in
14                             rule_options else None
15         self.trans_protocol =
16             prot_from_string(rule_options.get('trans_protocol', None))
17 }
```

El usuario debe ingresar la ruta a un archivo de reglas en formato json, que pueden ser compuestas (paquetes de TCP que provengan del puerto 80 deben descartarse), como también podría ser simple (paquetes de ICMP deben descartarse). Se debe incluir un vector de reglas, que cuente con diversos diccionarios. Este acepta en cada uno de los diccionarios las siguientes opciones:

- *src_port*: (int) numero de puerto de origen
- *dest_port*: (int) numero de puerto de destino.
- *src_ip*: [(string)ip, (int)mascara] IP de origen y mascara. Si se quiere bloquear una unica IP, en mascara se debe ingresar 32.
- *dest_ip*: [(string)ip, (int)mascara] IP de destino y mascara. Si se quiere bloquear una unica IP, en mascara se debe ingresar 32.
- *dest_mac*: (string) MAC de destino.
- *src_mac*: (string) MAC de origen.
- *trans_protocol*: (string) Protocolo de transporte. Se aceptan las opciones “ICMP” (aunque no sea puramente transporte), “UDP” y “TCP”

A tener en cuenta: la dirección IP del host “y” sera siempre la dirección “10.0.0.y”.

Si no se incluye ninguna regla en el json, o no se ingresa la ruta a un archivo de reglas, se ejecutara una red sin ningún firewall, donde los mensajes deberían llegar siempre. En la carpeta `pox/ext` se encuentra un archivo llamado `ejemplo.json`, en donde hay un ejemplo de como se deben settear las reglas. No es necesario que el usuario ingrese los campos que no quiere que se tengan en cuenta, sino que al no ingresarlos se los toma como nulos, y el switch al recibir un `None` en el match nota que no debe tenerlo en cuenta.

3.2.2. Supuestos de las reglas

Para la simplificación del parseo de las reglas, se tomaron algunos supuestos respecto de la información que deberá ingresar el usuario:

- Si se desea bloquear un par de hosts, se deben ingresar dos reglas separadas, en donde se alterne el host de destino y el host de origen, y así bloquear ambos sentidos de la comunicación.
- Si se desea bloquear un puerto, se debe especificar el protocolo de transporte a utilizar. Si se quiere bloquear un puerto tanto para UDP como para TCP, se deben agregar reglas separadas.

4. Pruebas

4.1. Testeo comando ping sin firewall

Por empezar, se brindan capturas (tanto de mininet como de Wireshark) para ver, una vez levantada la topología parametrizable, el correcto funcionamiento de la red a través del comando *pingall*.


```

manuel@manuel-Lenovo-V130-15IKB:~/Escritorio/distribuidos/trabajo2/tp2$ ./pox/pox.py log-level -DEBUG log-color forwarding_l2_learning
POX 0.3.0 (dart) / copyright 2011-2014 James McCauley, et al.
DEBUG:core:POX 0.3.0 (dart) going up...
DEBUG:core:Running on CPython (2.7.18/Jul 1 2022 12:27:04)
DEBUG:core:Platform is Linux-5.15.0-88-generic-x86_64-with-Ubuntu-20.04-focal
INFO:core:POX 0.3.0 (dart) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
DEBUG:forwarding_l2_learning:Connection [00-00-00-00-00-01 2]
INFO:openflow.of_01:[00-00-00-00-00-02 5] connected
DEBUG:forwarding_l2_learning:Connection [00-00-00-00-00-02 5]
INFO:openflow.of_01:[00-00-00-00-00-03 4] connected
DEBUG:forwarding_l2_learning:Connection [00-00-00-00-00-03 4]
INFO:openflow.of_01:[00-00-00-00-00-04 3] connected
DEBUG:forwarding_l2_learning:Connection [00-00-00-00-00-04 3]
DEBUG:openflow.of_01:1 connection aborted
DEBUG:forwarding_l2_learning:installing flow for ea:bf:db:32:fd:6f.3 -> 72:87:53:e2:74:00.2
87:53:e2:74:00.2
DEBUG:forwarding_l2_learning:installing flow for 72:87:53:e2:74:00.2 -> ea:bf:db:32:fd:6f.3
87:53:e2:74:00.2
DEBUG:forwarding_l2_learning:installing flow for ea:bf:db:32:fd:6f.3 -> 72:87:53:e2:74:00.2
87:53:e2:74:00.2
DEBUG:forwarding_l2_learning:installing flow for aa:dd:7f:05:a7:f1.2 -> 72:87:53:e2:74:00.2
87:53:e2:74:00.2
DEBUG:forwarding_l2_learning:installing flow for aa:dd:7f:05:a7:f1.2 -> 72:87:53:e2:74:00.1
87:53:e2:74:00.1
DEBUG:forwarding_l2_learning:installing flow for aa:dd:7f:05:a7:f1.2 -> 72:87:53:e2:74:00.1
87:53:e2:74:00.1
DEBUG:forwarding_l2_learning:installing flow for aa:dd:7f:05:a7:f1.1 -> 72:87:53:e2:74:00.2
87:53:e2:74:00.2
DEBUG:forwarding_l2_learning:installing flow for 72:87:53:e2:74:00.2 -> aa:dd:7f:05:a7:f1.1
87:53:e2:74:00.1
DEBUG:forwarding_l2_learning:installing flow for 72:87:53:e2:74:00.1 -> aa:dd:7f:05:a7:f1.1
87:53:e2:74:00.1

```

Figura 3: Pingall desde la interfaz de mininet

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	7e8b:e9bf:dbff:fe32:fd6f	ff02::2	ICMPv6	76	Router Solicitation from ea:bf:db:32:fd:6f
2	0.171151487	72:87:53:e2:74:00	Broadcast	ARP	42	Who has 10.0.0.2? Tell 10.0.0.1
3	0.190306866	72:87:53:e2:74:00	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
4	0.205673519	aa:dd:7f:05:a7:f1	72:87:53:e2:74:00	ARP	42	10.0.0.3 is at aa:dd:7f:05:a7:f1
5	0.211388616	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=0x2caa, seq=1/256, ttl=64 (reply in 6)
6	0.230415550	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=0x2caa, seq=1/256, ttl=64 (request in 5)
7	0.239351433	72:87:53:e2:74:00	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
8	0.260909654	e6:14:9e:9a:6e:9b	72:87:53:e2:74:00	ARP	42	10.0.0.4 is at e6:14:9e:9a:6e:9b
9	0.265974919	10.0.0.1	10.0.0.4	ICMP	98	Echo (ping) request id=0x2cab, seq=1/256, ttl=64 (reply in 10)
10	0.286469134	10.0.0.4	10.0.0.1	ICMP	98	Echo (ping) reply id=0x2cab, seq=1/256, ttl=64 (request in 9)
11	0.30285191	ea:bf:db:32:fd:6f	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.2
12	0.314227917	aa:dd:7f:05:a7:f1	ea:bf:db:32:fd:6f	ARP	42	10.0.0.3 is at aa:dd:7f:05:a7:f1
13	0.318676894	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) request id=0x2cad, seq=1/256, ttl=64 (reply in 14)
14	0.337045025	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) reply id=0x2cad, seq=1/256, ttl=64 (request in 13)
15	0.345792447	ea:bf:db:32:fd:6f	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.2
16	0.360390711	e6:14:9e:9a:6e:9b	ea:bf:db:32:fd:6f	ARP	42	10.0.0.4 is at e6:14:9e:9a:6e:9b
17	0.368270761	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) request id=0x2cae, seq=1/256, ttl=64 (reply in 18)
18	0.412508668	10.0.0.4	10.0.0.2	ICMP	98	Echo (ping) reply id=0x2cae, seq=1/256, ttl=64 (request in 17)
19	0.444255875	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) request id=0x2caf, seq=1/256, ttl=64 (reply in 20)
20	0.451219857	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) reply id=0x2caf, seq=1/256, ttl=64 (request in 19)
21	0.474723404	10.0.0.3	10.0.0.2	ICMP	98	Echo (ping) request id=0x2cb0, seq=1/256, ttl=64 (reply in 22)
22	0.481710318	10.0.0.2	10.0.0.3	ICMP	98	Echo (ping) reply id=0x2cb0, seq=1/256, ttl=64 (request in 21)
23	0.503816495	aa:dd:7f:05:a7:f1	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.3
24	0.516663060	10.0.0.4	10.0.0.1	ICMP	98	Echo (ping) request id=0x2cb2, seq=1/256, ttl=64 (reply in 25)
25	0.521743668	10.0.0.1	10.0.0.4	ICMP	98	Echo (ping) reply id=0x2cb2, seq=1/256, ttl=64 (request in 24)
26	0.543511089	10.0.0.4	10.0.0.2	ICMP	98	Echo (ping) request id=0x2cb3, seq=1/256, ttl=64 (reply in 27)
27	0.564871352	10.0.0.2	10.0.0.4	ICMP	98	Echo (ping) reply id=0x2cb3, seq=1/256, ttl=64 (request in 26)
28	5.418787157	aa:dd:7f:05:a7:f1	ea:bf:db:32:fd:6f	ARP	42	Who has 10.0.0.2? Tell 10.0.0.3
29	5.420905064	aa:dd:7f:05:a7:f1	72:87:53:e2:74:00	ARP	42	Who has 10.0.0.1? Tell 10.0.0.3
30	5.420858694	e6:14:9e:9a:6e:9b	72:87:53:e2:74:00	ARP	42	Who has 10.0.0.1? Tell 10.0.0.4
31	5.420862613	ea:bf:db:32:fd:6f	aa:dd:7f:05:a7:f1	ARP	42	10.0.0.2 is at ea:bf:db:32:fd:6f

Figura 4: Pingall desde wireshark

Se puede ver en la 4 como cada *Echo(ping) request* tiene su correspondiente *Echo(ping) reply*, a su vez que la 3 muestra como cada *host* pudo comunicarse con el resto.

Vale aclarar que el comando fue ejecutado sobre la topología con la que se trabajó, pero que no se usó el controlador que funciona como *firewall*, justamente para verificar la comunicación entre todos los *hosts* sin que se *droppee* alguno de los datagramas ICMP en el camino. Además,

se capturaron los paquetes que pasan por la interfaz del *switch* 2, por lo que los *pings* que tienen como extremo a los pares $(h1, h2)$, $(h3, h4)$ -e invertidos los ordenes- no se verán por pantalla por no ser capturados por el *switch*. Igualmente, puede verse por la interfaz de mininet que todos los *pings* llegan a destino.

4.2. Testeo comando ping con firewall

A continuación, se probó configurando una topología con 4 switches, en la cual se estableció el switch 1 como firewall. En este caso, se aplica una única regla que establece que se debe filtrar todos los paquetes cuyo protocolo de transporte (o algo así) sea ICMP, la cual se puede observar en la figura 5.

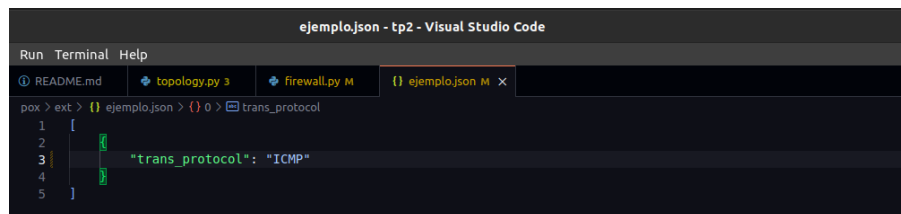


Figura 5: Regla que bloquea paquetes ICMP

Luego de establecida la topología y su firewall, se corrió el comando pingall. Esta corrida se puede observar en la figura 6. Notamos que los únicos pings que logran llegar son los correspondientes a la interacción entre el host 3 y host 4. Esto se puede explicar observando la figura 7, donde se observa como para enviar paquetes desde el host 3 al 4 y viceversa, no se necesita atravesar el host 1 por lo que sus paquetes no se droppean, mientras que para todo el resto de las interacciones si es necesario.

```

manuelpol@manuelpol-Lenovo-V130-15IKB:~/Escritorio/distribuido
manuelpol@manuelpol-Lenovo-V130-15IKB:~/Escritorio/distribuidos/trabajo2/tp2$ s
udo mn --custom topology.py --topo customTopo,4 --controller=remote,ip=127.0.0.1,port=6633
s1
s2
s3
s4
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(s1, h1) (s1, h2) (s1, s2) (s2, s3) (s3, s4) (s4, h3) (s4, h4)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X h4
h4 -> X X h3
*** Results: 83% dropped (2/12 received)
mininet>

```

Figura 6: Testeo del comando pingall

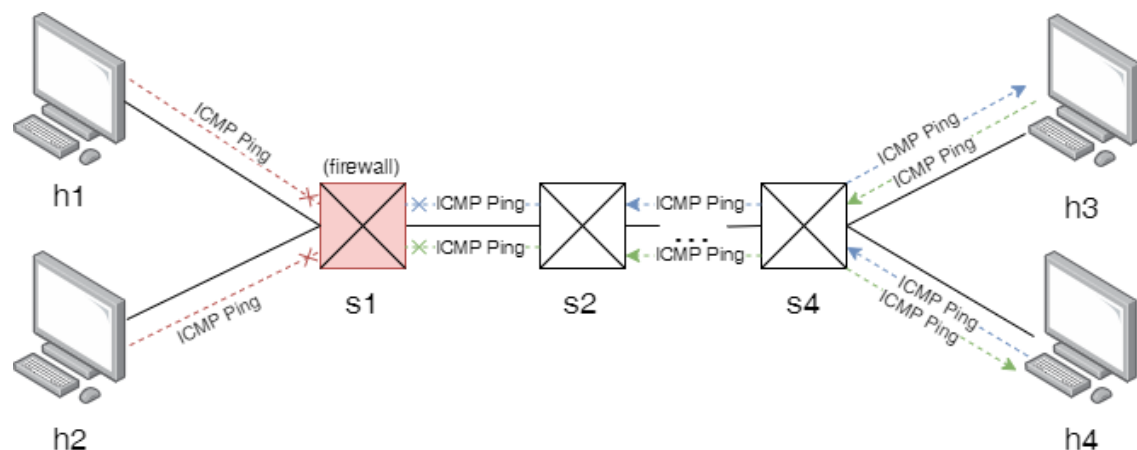


Figura 7: Topología con firewall bloqueando ICMP

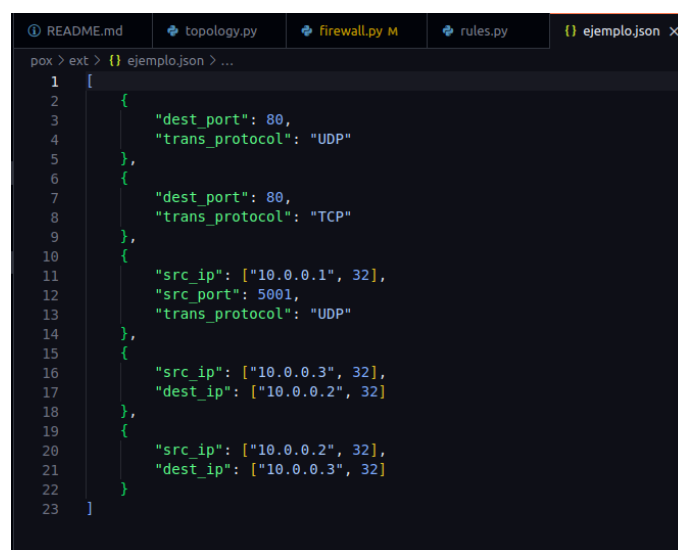
4.3. Testeo de firewall con las reglas de la consigna

Se utilizó un archivo configurable de firewall, en donde se establecen las reglas propuestas en la consigna del trabajo:

- Se deben descartar todos los paquetes cuyo puerto de destino sea el 80.
- Se deben descartar todos los mensajes que provengan del host 1, tengan como puerto destino el 5001, y estén utilizando el protocolo UDP.

- Se debe elegir dos hosts cualquiera (en este caso, h2 y h3), y los mismos no deben poder comunicarse de ninguna forma.

Se estableció a su vez el firewall en el switch 2. Sobre esta configuración, se realizaron diversas pruebas para testear que se estén droppeando los paquetes que cumplen con las reglas, mientras que el resto pueda ser enviado. Estas reglas establecidas se observan en la figura 8, donde las primeras dos corresponden al droppeo de los paquetes del puerto de destino 80, la tercera corresponde al droppeo de los provenientes del host 1 en el puerto 5001, utilizando UDP, y las ultimas dos corresponden a la incomunicación entre los hosts 2 y 3.



```
1  [
2    {
3      "dest_port": 80,
4      "trans_protocol": "UDP"
5    },
6    {
7      "dest_port": 80,
8      "trans_protocol": "TCP"
9    },
10   {
11     "src_ip": ["10.0.0.1", 32],
12     "src_port": 5001,
13     "trans_protocol": "UDP"
14   },
15   {
16     "src_ip": ["10.0.0.3", 32],
17     "dest_ip": ["10.0.0.2", 32]
18   },
19   {
20     "src_ip": ["10.0.0.2", 32],
21     "dest_ip": ["10.0.0.3", 32]
22   }
23 ]
```

Figura 8: Comunicación entre host 1 (servidor) y host 4 (cliente) (TCP puerto 80)

4.3.1. Paquetes cuyo host de destino es el 80

En primer lugar, se realizaron pruebas correspondientes al droppeo de los paquetes del puerto 80. Haciendo uso de xterm con iperf, observamos en las figuras 9 y 10 el cumplimiento de esta regla. En ambas se tiene al host h1 como servidor escuchando en el puerto 80. En la figura 9, el host 4 intenta mandar un paquete TCP (por default los paquetes son TCP) al host 1, y notamos que del lado del host 4 se nota que la conexión falla, mientras que del lado del host 1 no se recibe absolutamente nada. En la figura 10 se realiza una prueba similar pero usando UDP (flag -u), y desde el lado del host 1 sucede lo mismo que en la anterior, mientras que del lado del host 4, si bien el output es distinto, se especifica que no se pudo enviar los datagramas.

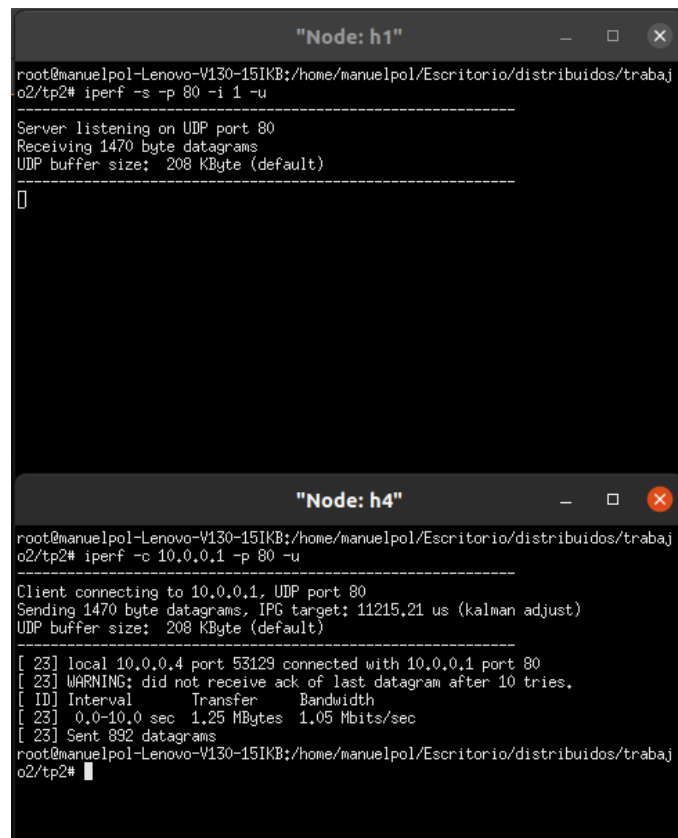


The image shows two terminal windows. The top window, titled "Node: h1", displays the command `root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabajo2/tp2# iperf -s -p 80 -i 1` followed by the output `Server listening on TCP port 80` and `TCP window size: 85,3 KByte (default)`. The bottom window, titled "Node: h4", displays the command `root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabajo2/tp2# iperf -c 10.0.0.1 -p 80` followed by the output `connect failed: Operation now in progress`.

```
"Node: h1"
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabajo2/tp2# iperf -s -p 80 -i 1
-----
Server listening on TCP port 80
TCP window size: 85,3 KByte (default)
-----
[]

"Node: h4"
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabajo2/tp2# iperf -c 10.0.0.1 -p 80
connect failed: Operation now in progress
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabajo2/tp2#
```

Figura 9: Comunicación entre host 1 (servidor) y host 4 (cliente) (TCP puerto 80)



```
"Node: h1"
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Esitorio/distribuidos/trabaj
o2/tp2# iperf -s -p 80 -i 1 -u
-----
Server listening on UDP port 80
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[]

"Node: h4"
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Esitorio/distribuidos/trabaj
o2/tp2# iperf -c 10.0.0.1 -p 80 -u
-----
Client connecting to 10.0.0.1, UDP port 80
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 23] local 10.0.0.4 port 53129 connected with 10.0.0.1 port 80
[ 23] WARNING: did not receive ack of last datagram after 10 tries.
[ 10] Interval      Transfer      Bandwidth
[ 23] 0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 23] Sent 892 datagrams
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Esitorio/distribuidos/trabaj
o2/tp2#
```

Figura 10: Comunicación entre host 1 (servidor) y host 4 (cliente) (UDP puerto 80)

En wireshark para el envío de TCP (figura 11), se pueden ver el paquete inicial de SYN, el paquete para iniciar el handshake con el servidor, pero este se pierde por el firewall, resultando en retransmisión de parte del cliente. Para el envío de UDP (figura 12), se puede ver que nunca se reenvía un ack (correspondiente al protocolo de iperf, no por UDP) a los paquetes que manda el cliente.

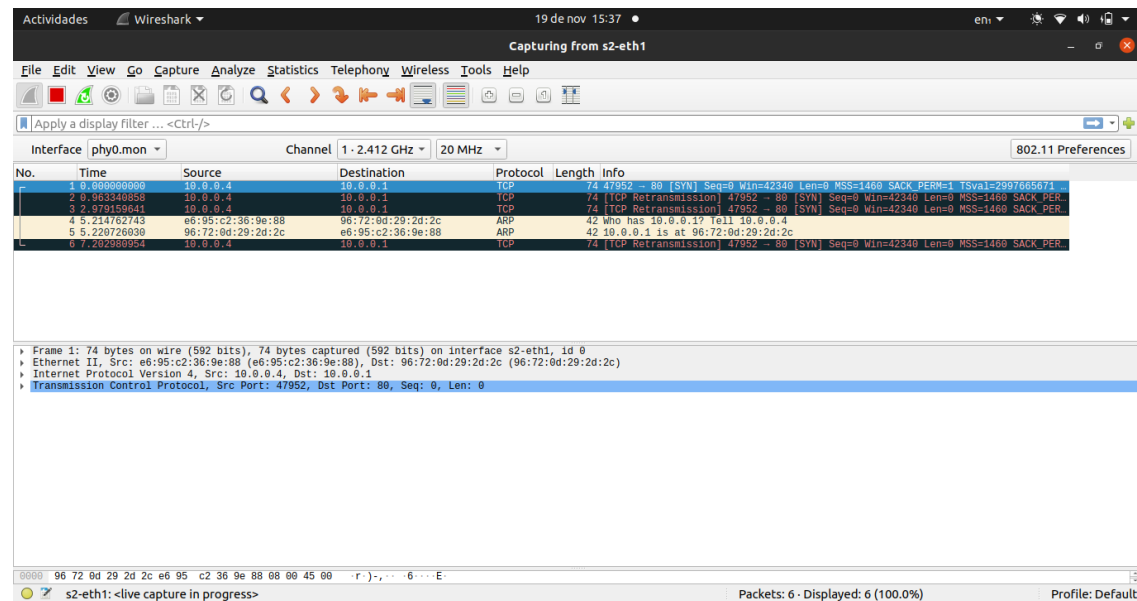


Figura 11: Vista de wireshark de la comunicación con TCP puerto 80

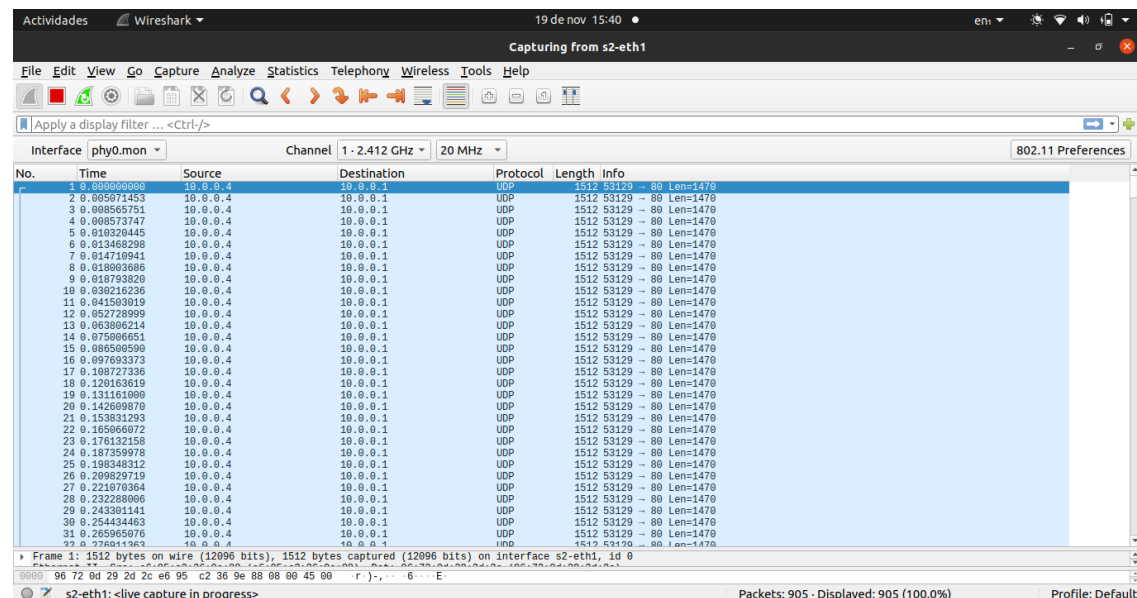


Figura 12: Vista de wireshark de la comunicación con UDP puerto 80

Se observa lo que sucede en el switch 2, el cual tiene el firewall. Notamos que los paquetes ingresan a este, pero no se tiene una salida.

4.3.2. Paquetes cuyo host de origen es el 1, tienen como protocolo de transporte UDP y tienen puerto de destino 5001

Finalmente se probó la regla que descarta los paquetes provenientes del host 1, con UDP y su puerto de destino es el 5001. Para probarlo se levanto en el host 4 un servidor que escucha en el puerto 5001, el cual recibirá distintos paquetes UDP. Se envían paquetes UDP a este servidor desde los hosts 1 y 2. Como se puede observar en la figura 13, notamos que los paquetes del host 2 llegan a destino, mientras que para el host 1 se nos informa que no se recibió un ack para el datagrama.

```

"Node: h1"
root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro# iperf
-c 10.0.0.4 -u -p 5001
root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro# iperf
-c 10.0.0.4 -u -p 5001 -n 10

-----
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)
-----
[ 23] local 10.0.0.1 port 49626 connected with 10.0.0.4 port 5001
[ 23] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval      Transfer      Bandwidth
[ 23] 0.0- 0.0 sec  10.0 Bytes  5.14 Kbits/sec
[ 23] Sent 1 datagrams
root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro#

"Node: h2"
root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro# iperf
-c 10.0.0.4 -u -p 5001
root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro# iperf
-c 10.0.0.4 -u -p 5001 -n 10

-----
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)
-----
[ 23] local 10.0.0.2 port 55176 connected with 10.0.0.4 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 23] 0.0- 0.0 sec  10.0 Bytes  6.93 Kbits/sec
[ 23] Sent 1 datagrams
[ 23] Server Report:
[ 23] 0.0- 0.0 sec  1.44 KBytes  51.1 Mbits/sec  0.000 ms  0/ 1 (0%)
root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro#

"Node: h4"
root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro# iperf
-s -u -p 5001
root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro# iperf
-s -u -p 5001

-----
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 23] local 10.0.0.4 port 5001 connected with 10.0.0.2 port 55176
[ ID] Interval      Transfer      Bandwidth      Jitter  Lost/Total Datagrams
[ 23] 0.0- 0.0 sec  1.44 KBytes  51.1 Mbits/sec  0.000 ms  0/ 1 (0%)

```

Figura 13: Comunicación con el host 4 servidor

Se puede observar en wireshark en la figura 14 las dos interfaces del switch 2. Del lado izquierdo,

podemos observar los paquetes que ingresan al switch proviniendo de h1 y h2 hacia h4 pasando por este switch, y los que salen de este switch proviniendo de h4 y yendo hacia h1 y h2, mientras que del lado derecho aparecen los paquetes que ingresan al switch desde h4, y los que salen proviniendo de h1 y h2 hacia h4. Notamos que los primeros 3, que corresponden a la interacción entre los hosts 2 y 4, entran y salen, mientras que las siguientes corresponden al envío de paquetes desde el host 1 al 4 nunca logran salir, por lo que no reciben una respuesta y notamos que estan siendo droppeados. Esto se corresponde con las reglas establecidas, en donde el switch 2 debería bloquear a los paquetes de h1 que se dirigen al puerto 5001 y usan UDP.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.2	10.0.0.4	UDP	52	
2	0.000484615	10.0.0.2	10.0.0.4	UDP	1512	
3	0.009569763	10.0.0.4	10.0.0.2	UDP	1512	
4	3.248638130	10.0.0.1	10.0.0.4	UDP	52	
5	3.249478629	10.0.0.1	10.0.0.4	UDP	1512	
6	3.471317940	10.0.0.1	10.0.0.4	UDP	1512	
7	3.721867247	10.0.0.1	10.0.0.4	UDP	1512	
8	3.974593721	10.0.0.1	10.0.0.4	UDP	1512	
9	4.228683993	10.0.0.1	10.0.0.4	UDP	1512	
10	4.481479993	10.0.0.1	10.0.0.4	UDP	1512	
11	4.734487821	10.0.0.1	10.0.0.4	UDP	1512	
12	4.985427356	10.0.0.1	10.0.0.4	UDP	1512	
17	5.236740243	10.0.0.1	10.0.0.4	UDP	1512	
18	5.487190358	10.0.0.1	10.0.0.4	UDP	1512	

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.2	10.0.0.4	UDP	52	
2	0.001695317	10.0.0.2	10.0.0.4	UDP	1512	
3	0.034274485	10.0.0.4	10.0.0.2	UDP	1512	

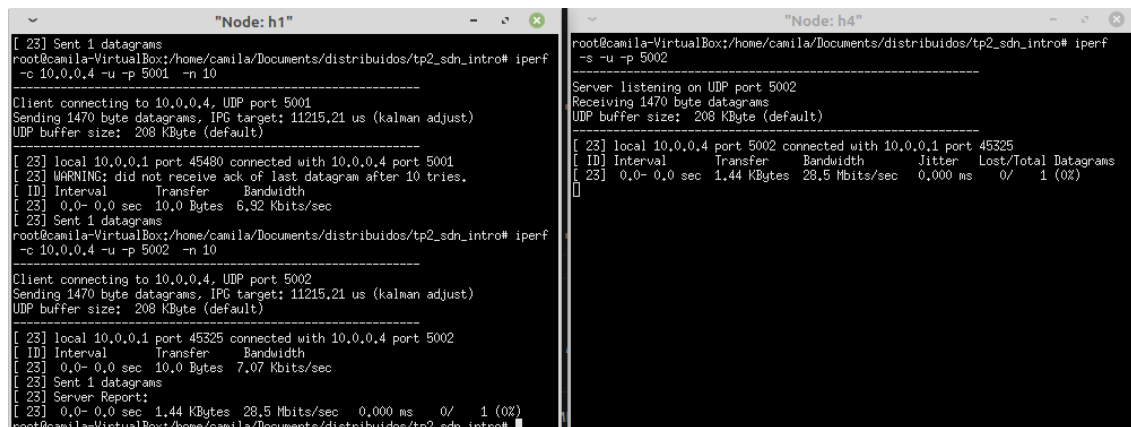
Figura 14: Vista de wireshark de interfaces de s2

Se verifican también los casos en donde se varia el protocolo y el puerto, y se ve que en estos casos si puede establecerse la conexión. En la figura 15 se observa como al comunicarse el host 1 con el host 4 con TCP, los paquetes si pueden viajar, y se recibe una respuesta.

Node: h1	Node: h4
<pre> root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro# iperf -c 10.0.0.4 -p 5001 -n 10 Client connecting to 10.0.0.4, TCP port 5001 TCP window size: 85.3 kByte (default) [23] local 10.0.0.1 port 38354 connected with 10.0.0.4 port 5001 [10] Interval Transfer Bandwidth [23] 0.0- 0.0 sec 10.0 Bytes 473 Kbits/sec root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro# </pre>	<pre> root@camila-VirtualBox:/home/camila/Documents/distribuidos/tp2_sdn_intro# iperf -s -p 5001 Server listening on TCP port 5001 TCP window size: 85.3 kByte (default) [24] local 10.0.0.4 port 5001 connected with 10.0.0.1 port 38354 [10] Interval Transfer Bandwidth [24] 0.0- 0.0 sec 10.0 Bytes 7.47 Kbits/sec </pre>

Figura 15: Comunicación de h1 y h4 por TCP en puerto 5001

Por ultimo, se prueba el caso en el que se establece el servidor en h4 en el puerto 5002. Vemos en la figura 16 que en este caso también se da una comunicación exitosa.



```
"Node: h1"
root@camila-VirtualBox: /home/camila/Documents/distribuidos/tp2_sdn_intro# iperf
-c 10.0.0.4 -u -p 5001 -n 10

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)

[ 23] local 10.0.0.1 port 45480 connected with 10.0.0.4 port 5001
[ 23] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval      Transfer    Bandwidth
[ 23] 0.0- 0.0 sec  10.0 Bytes  6.92 Kbits/sec
[ 23] Sent 1 datagrams

root@camila-VirtualBox: /home/camila/Documents/distribuidos/tp2_sdn_intro# iperf
-c 10.0.0.4 -u -p 5002 -n 10

Client connecting to 10.0.0.4, UDP port 5002
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)

[ 23] local 10.0.0.1 port 45325 connected with 10.0.0.4 port 5002
[ ID] Interval      Transfer    Bandwidth
[ 23] 0.0- 0.0 sec  10.0 Bytes  7.07 Kbits/sec
[ 23] Sent 1 datagrams

[ 23] Server Report:
[ 23] 0.0- 0.0 sec  1.44 KBytes 28.5 Mbits/sec  0.000 ms  0/ 1 (0%)

root@camila-VirtualBox: /home/camila/Documents/distribuidos/tp2_sdn_intro#

"Node: h4"
root@camila-VirtualBox: /home/camila/Documents/distribuidos/tp2_sdn_intro# iperf
-s -u -p 5002

Server listening on UDP port 5002
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)

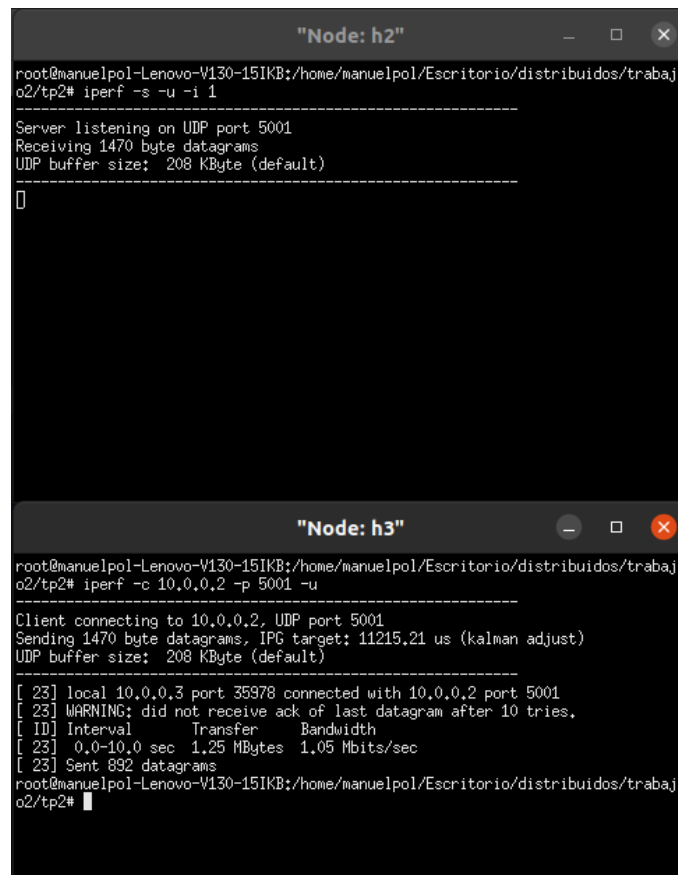
[ 23] local 10.0.0.4 port 5002 connected with 10.0.0.1 port 45325
[ ID] Interval      Transfer    Bandwidth      Jitter  Lost/Total Datagrams
[ 23] 0.0- 0.0 sec  1.44 KBytes 28.5 Mbits/sec  0.000 ms  0/ 1 (0%)

root@camila-VirtualBox: /home/camila/Documents/distribuidos/tp2_sdn_intro#
```

Figura 16: Comunicación de h1 y h4 por UDP en puerto 5002

4.3.3. Paquetes entre los hosts 2 y 3

Para realizar esta prueba, se debió realizar la comunicación bidireccional, y ver que falle de ambos lados. En primer lugar, se estableció al host 3 como cliente y al host 2 como servidor. Se observa en la figura 17, como no llegan los paquetes del host 3 al host 2. Esto cumple con la quinta regla definida en el ejemplo.json (se descartan paquetes con origen en el host 3 y destino en el host 2), y es parte de la tercera regla mencionada en la consigna.



The image shows two terminal windows. The top window, titled "Node: h2", shows a server listening on UDP port 5001 and receiving 1470 byte datagrams. The bottom window, titled "Node: h3", shows a client connecting to 10.0.0.2 on UDP port 5001, sending 1470 byte datagrams, and displaying a summary of the transfer: 0.0-10.0 sec, 1.25 MBytes, 1.05 Mbits/sec, with 892 datagrams sent.

```
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabaj
o2/tp2# iperf -s -u -i 1
-----
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[]

root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabaj
o2/tp2# iperf -c 10.0.0.2 -p 5001 -u
-----
Client connecting to 10.0.0.2, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 23] local 10.0.0.3 port 35978 connected with 10.0.0.2 port 5001
[ 23] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval      Transfer    Bandwidth
[ 23] 0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 23] Sent 892 datagrams
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabaj
o2/tp2#
```

Figura 17: Comunicación entre host 3 (cliente) y host 2 (servidor)

En la figura 18, se observa que los paquetes enviados por h3 llegan al switch 2, pero nunca salen. A la izquierda se ve la interfaz 1, por donde hubiesen salido los paquetes hacia el host 2 y a la derecha vemos que le llegan pero son dropados.

Del lado izquierdo, podemos observar los paquetes que ingresan al switch proviniendo de h2 hacia h3 pasando por este switch, y los que salen de este switch proviniendo de h3 y yendo hacia h2, mientras que del lado derecho aparecen los paquetes que ingresan al switch desde h3, y los que salen proviniendo de h2 hacia h3. Notamos que del lado izquierdo, mas que los paquetes ARP para conocer a los distintos hosts no hay nada, por lo que decimos que el host 2 nunca envía nada al host 3. Dado que del lado derecho entran paquetes pero del lado izquierdo nunca se pueden observar, notamos que estos están siendo dropados.

No.	Time	Source	Destination	Protocol	Length
1	0.000000000	d6:85:5c:5d:a7:e0	Broadcast	ARP	4
2	0.011194145	0a:46:ab:46:f0:fe	d6:85:5c:5d:a7:e0	ARP	4

Time	Source	Destination	Protocol	Length
1	0.000000000	d6:85:5c:5d:a7:e0	Broadcast	ARP
2	0.029763540	0a:46:ab:46:f0:fe	d6:85:5c:5d:a7:e0	ARP
3	0.055186958	10.0.0.3	10.0.0.2	UDP
4	0.057889387	10.0.0.3	10.0.0.2	UDP
5	0.059180117	10.0.0.3	10.0.0.2	UDP
6	0.063383488	10.0.0.3	10.0.0.2	UDP
7	0.064216361	10.0.0.3	10.0.0.2	UDP
8	0.066689132	10.0.0.3	10.0.0.2	UDP
9	0.067572880	10.0.0.3	10.0.0.2	UDP
10	0.069685745	10.0.0.3	10.0.0.2	UDP
11	0.070329121	10.0.0.3	10.0.0.2	UDP
12	0.071986310	10.0.0.3	10.0.0.2	UDP
13	0.073197008	10.0.0.3	10.0.0.2	UDP
14	0.082858991	10.0.0.3	10.0.0.2	UDP
15	0.092692135	10.0.0.3	10.0.0.2	UDP
16	0.103933683	10.0.0.3	10.0.0.2	UDP
17	0.115124209	10.0.0.3	10.0.0.2	UDP
18	0.126539391	10.0.0.3	10.0.0.2	UDP
19	0.137595691	10.0.0.3	10.0.0.2	UDP
20	0.148969852	10.0.0.3	10.0.0.2	UDP
21	0.169827789	10.0.0.3	10.0.0.2	UDP
22	0.171287747	10.0.0.3	10.0.0.2	UDP
23	0.182622331	10.0.0.3	10.0.0.2	UDP
24	0.193824797	10.0.0.3	10.0.0.2	UDP
25	0.204916151	10.0.0.3	10.0.0.2	UDP
26	0.216142345	10.0.0.3	10.0.0.2	UDP
27	0.227807084	10.0.0.3	10.0.0.2	UDP
28	0.238537079	10.0.0.3	10.0.0.2	UDP
29	0.249783211	10.0.0.3	10.0.0.2	UDP
30	0.260987628	10.0.0.3	10.0.0.2	UDP
31	0.272283277	10.0.0.3	10.0.0.2	UDP

Figura 18: Captura de wireshark desde el switch 2

Luego se estableció al host 2 como cliente y al host 3 como servidor. Se observa en la figura 19, como no llegan los paquetes del host 2 al host 3, cumpliendo con la quinta regla definida en el ejemplo.json (se descartan paquetes con origen en el host 2 y destino en el host 3), y es parte de la tercera regla mencionada en la consigna.

```

"Node: h2"
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabaj
o2/tp2# iperf -c 10.0.0.3 -p 5001 -u

Client connecting to 10.0.0.3, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size: 208 KByte (default)

[ 23] local 10.0.0.2 port 40551 connected with 10.0.0.3 port 5001
[ 23] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval      Transfer      Bandwidth
[ 23] 0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 23] Sent 892 datagrams
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabaj
o2/tp2#

"Node: h3"
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/trabaj
o2/tp2# iperf -s -u -i 1

Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)

```

Figura 19: Comunicación entre host 2 (cliente) y host 3 (servidor)

Asimismo, sucede algo similar con la captura de paquetes. En este caso, vemos como los paquetes de h2 llegan al switch s2 por la interfaz 1 pero no son redirigidos a la interfaz 2, por lo que notamos que están siendo droppeados.

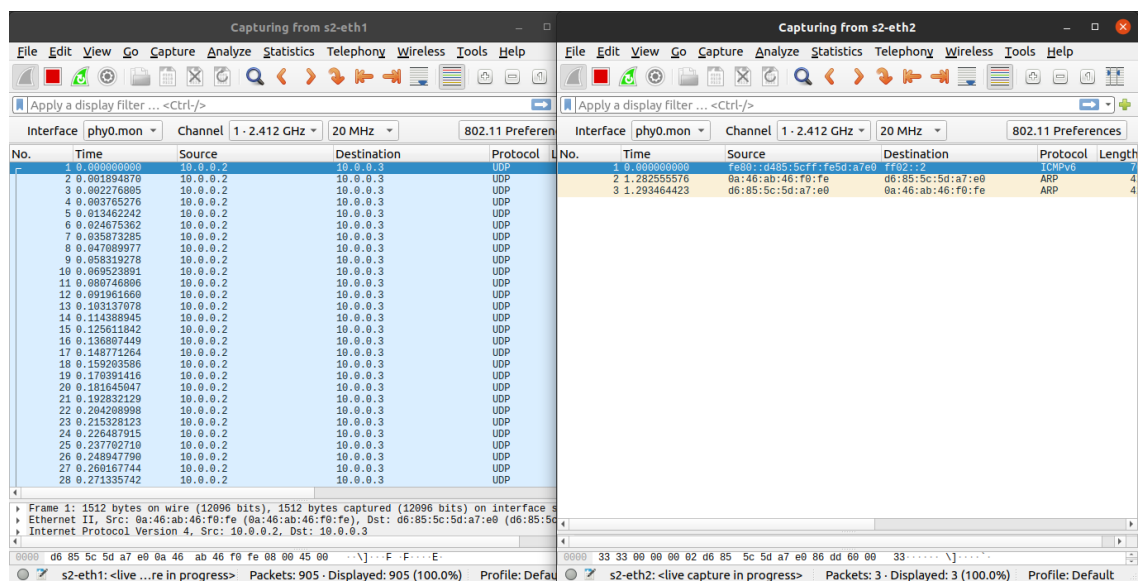


Figura 20: Captura de wireshark del switch 2

4.3.4. Comunicación entre hosts que no deben droppearse

Para finalizar, se agrego un ultimo caso en donde se debería observar comunicación funcional entre dos hosts de nuestra red. Esto se puede notar en la figura 21. En este caso se configuran al host 1 en el puerto como servidor, y al host 4 como cliente. El host 4 envía paquetes TCP al host 1, quien se encuentra conectado en el puerto 2000, y recibe la respuesta de todos los paquetes enviados, ya que no cumple con ninguna regla en su totalidad. En la figura 22 observamos que hay paquetes yendo desde y viniendo de ambos lados en el switch 2. Esto nos deja ver que este switch, quien tiene el firewall, no esta droppeando estos paquetes.

```
"Node: h1"
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/tp2# i
perf -s -p 2000 -i 1

-----
Server listening on TCP port 2000
TCP window size: 85.3 KByte (default)
-----
[ 24] local 10.0.0.1 port 2000 connected with 10.0.0.4 port 38966
[ ID] Interval      Transfer    Bandwidth
[ 24] 0.0- 1.0 sec  2.28 GBytes 19.6 Gbits/sec
[ 24] 1.0- 2.0 sec  2.52 GBytes 21.6 Gbits/sec
[ 24] 2.0- 3.0 sec  2.69 GBytes 23.1 Gbits/sec
[ 24] 3.0- 4.0 sec  3.35 GBytes 28.8 Gbits/sec
[ 24] 4.0- 5.0 sec  3.44 GBytes 29.6 Gbits/sec
[ 24] 5.0- 6.0 sec  2.96 GBytes 25.4 Gbits/sec
[ 24] 6.0- 7.0 sec  3.78 GBytes 32.5 Gbits/sec
[ 24] 7.0- 8.0 sec  3.57 GBytes 30.7 Gbits/sec
[ 24] 8.0- 9.0 sec  2.95 GBytes 25.4 Gbits/sec
[ 24] 9.0-10.0 sec  3.76 GBytes 32.3 Gbits/sec
[ 24] 0.0-10.0 sec 31.3 GBytes 26.9 Gbits/sec
[]

"Node: h4"
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/tp2# i
perf -c 10.0.0.1 -p 2000

-----
Client connecting to 10.0.0.1, TCP port 2000
TCP window size: 2.50 MByte (default)
-----
[ 23] local 10.0.0.4 port 38966 connected with 10.0.0.1 port 2000
[ ID] Interval      Transfer    Bandwidth
[ 23] 0.0-10.0 sec  31.3 GBytes 26.9 Gbits/sec
root@manuelpol-Lenovo-V130-151KB:/home/manuelpol/Escritorio/distribuidos/tp2#
```

Figura 21: Comunicación entre host 1 (servidor) y host 4 (cliente)

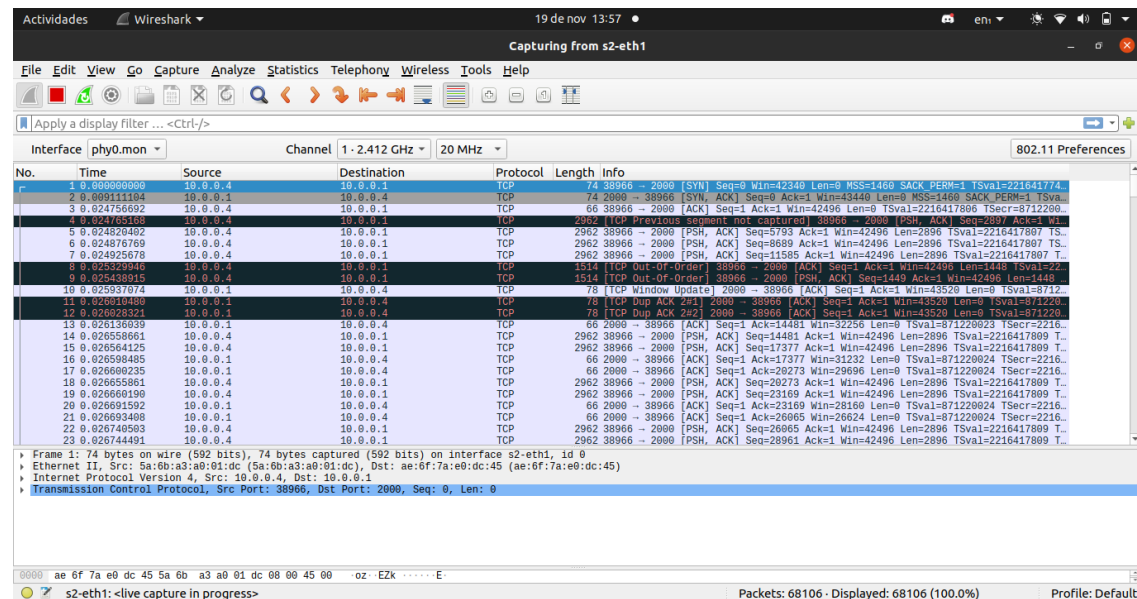


Figura 22: Comunicación entre host 1 (servidor) y host 4 (cliente) por TCP en Wireshark

5. Conclusiones

Gracias a este trabajo practico comprendimos muchos conceptos relacionados a las redes. Entre ellos, aprendimos que son los firewalls, que tipos de reglas tienen, como funcionan y como implementarlos dentro de una SDN. Asimismo, mediante la implementación del firewall pudimos entender con mayor profundidad los principios detrás de un OpenFlow y como una SDN usa esta información. No solo tuvimos que aprender como funcionaba POX, sino también decidir en que momento configurar el switch con el firewall. Comprendimos a su vez como levantar una red local utilizando mininet y generar topologías personalizadas, logrando comunicaciones entre hosts en distintas direcciones. Además, profundizamos nuestros conocimientos de Wireshark, al hacer uso del mismo con el fin de revisar el correcto funcionamiento de nuestro firewall y los paquetes que llegaban y salían por las distintas interfaces.

6. Referencias

- [1] Kurose, J. F., & Ross, K. W. (2007). Computer networking: A top-down approach edition. Addison Wesley.
- [2] POX: <https://noxrepo.github.io/pox-doc/html/>