

# Introducción a los Sistemas Distribuidos (75.43)

## TP N° 3: Software-Defined Networks

1° Cuatrimestre, 2022

Apellido y Nombre	Padrón	Email
Gonzalo Sabatino	104609	gsabatino@fi.uba.ar
Mateo Capón Blanquer	104258	mcapon@fi.uba.ar
Santiago Pablo Fernandez Caruso	105267	sfernandezc@fi.uba.ar
Ignacio Iragui	105110	iiragui@fi.uba.ar
Federico Jose Pacheco	104541	fpacheco@fi.uba.ar

## Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Marco teórico</b>	<b>2</b>
2.1. Mininet . . . . .	2
2.2. POX . . . . .	2
<b>3. Preguntas a Responder</b>	<b>3</b>
3.1. ¿Cuál es la diferencia entre un switch y un router? ¿Qué tienen en común? . . . .	3
3.2. ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow? . . . .	3
3.3. ¿Se pueden reemplazar todos los routers de la Intenet por Switches OpenFlow? Piense en el escenario interASes para elaborar su respuesta . . . . .	5
<b>4. Desarrollo del trabajo</b>	<b>6</b>
4.1. Estructura del Código . . . . .	6
4.2. Resultados de las simulaciones . . . . .	10
4.3. Iperf . . . . .	33
<b>5. Conclusión</b>	<b>38</b>
<b>6. Referencias</b>	<b>38</b>

## 1. Introducción

El presente informe reúne la documentación del Trabajo Práctico 3 de la materia Introducción a los Sistemas Distribuidos - Facultad de Ingeniería Universidad de Buenos Aires. El trabajo consiste en simular una red utilizando el protocolo OpenFlow, utilizando distintas herramientas para entender los resultados obtenidos.

## 2. Marco teórico

En esta sección se desarrolla un marco teórico sobre las principales herramientas utilizadas.

### 2.1. Mininet

Es un software que permite emular el data-plane de una topología de red (conjunto de hosts, routers, switches y links) dentro de una misma computadora, utilizando la virtualización de Linux para aislar cada uno de estos componentes.

Para implementar una topología de red propia, se debe instanciar algún controlador (quien implementa el comportamiento del control-plane). Un ejemplo de esto es POX.

### 2.2. POX

Es un software que permite instanciar un controlador SDN utilizando el protocolo OpenFlow, para interactuar con switches OpenFlow (por ejemplo, utilizando mininet).

Lo más importante es que permite definir acciones a partir de los distintos eventos que se disparan en el controlador: Definir una tabla de flujos en el inicio de la conexión con los switches, ver qué paquetes llegan a un switch y no matchean ninguna entrada de la tabla de flujos, definir el tiempo de vida de una entrada en la tabla de flujos para un switch particular, entre otros.

### L2 Learning

Si un determinado switch no tiene un match en su tabla de flujos, envía un mensaje al controlador, `packet_in`, pidiéndole la interfaz de salida. El controlador procesa qué acción debe ejecutar el switch y se la envía.

Estas acciones pueden ser por ejemplo que se realice un forward del paquete a un determinado puerto o que lo deje de lado (drop). En general, cuando se configura l2 learning, el mensaje que le envía el controlador al switch es del tipo “Output to switch port”, agregándole el output port igual al `OFPP_FLOOD = 65531`. El puerto de flood es un puerto reservado en openflow, el cual le indica al switch que debe reenviar el paquete a todas las interfaces de salida con excepción de los puertos que están en estado `OFPPS_BLOCKED` y de la interfaz por la que llegó el paquete.

Por otro lado, si no hay un match en la tabla de flujos del switch, pero sí en la tabla que almacena el controlador para este switch en particular, el controlador agrega el flujo correspondiente a la tabla del switch. Luego, le indica que debe enviar el paquete por el puerto `OFPP_TABLE`. Este puerto reservado representa el inicio del pipeline de openflow. Por lo tanto, lo enviará por el puerto añadido en el mensaje de `packet_out`.

### 3. Preguntas a Responder

#### 3.1. ¿Cuál es la diferencia entre un switch y un router? ¿Qué tienen en común?

##### Switches y routers convencionales

La principal diferencia entre un switch y un router convencionales es la capa sobre la que operan. El switch opera sobre la capa de enlace, y el router sobre la de red. Mientras que el router conoce de *datagramas*, el switch interpreta *frames*. Por lo tanto, los switches reconocen direcciones MAC (también conocidas como direcciones LAN o direcciones físicas), mientras que los routers reconocen direcciones IP.

Las direcciones MAC y las direcciones IP son únicas en el mundo, ambas las distribuye la IEEE. Las direcciones IP, a diferencia de las direcciones MAC, usan direccionamiento jerárquico: cada dirección tiene una parte que indica la dirección de red, y otra que indica la dirección de host. En cambio, las MAC utilizan direccionamiento plano.

Tanto el switch como el router tienen en común que llevan a cabo un guardado temporal de paquetes para luego poder realizar su correspondiente forwarding. La salida del paquete que reciben se decide utilizando una tabla de búsqueda. Para el switch, la switching table; y para el router, la routing table. La estructura jerárquica de las direcciones IP permite definir la regla *Longest Prefix Match*, para decir sobre qué interfaz se debe reenviar el datagrama.

En general las tablas de ruteo tienen una entrada correspondida con el *Default Gateway* (si así lo define quien lo configure) cuya máscara es la menos específica (0.0.0.0/0), por la cuál son enviados aquellos paquetes que no tienen un match más específico. En contraposición, si no hay match en la switching table, se manda el paquete a todas las salidas excepto a la que se corresponde con la interfaz por la que llegó el mismo.

Otra diferencia entre los switches y los routers es el modo en el que se configura su respectiva tabla. Los switches no deben ser configurados para poder entrar en funcionamiento, se dice que son *plug-and-play*. Una vez instalados, comienzan con su switching table vacía, y la misma se agranda a medida que recibe paquetes. En contraste, los routers deben configurarse para ponerlos en funcionamiento. Utilizan algoritmos de ruteo como RIP u OSPF para determinar el camino a seguir.

El hecho de que los switches procesen en la capa de enlace, a diferencia de los routers que se ubican una capa más arriba, hace que el tiempo de procesamiento de los routers sea mayor que al de los switches. Sin embargo, los routers, al utilizar algoritmos que calculan el camino mínimo, generan rutas más óptimas.

##### Ruteo dentro de SDN

El objetivo de SDN es desacoplar el control-plane del data-plane, y que los dispositivos de red sirvan para cumplir las políticas y reglas definidas en el controlador lógicamente centralizado.

Al ser SDN vertical con todas las capas, no se distingue entre switches y routers dentro de SDN.

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

La principal diferencia reside en: Un switch convencional tiene acoplados su control-plane y data-plane en el mismo dispositivo (lo que produce una arquitectura de red mucho más rígida, y difícil de cambiar y mantener). En cambio, los switches de OpenFlow, al ser una posible implementación de SDN, rompen el acoplamiento de estas capas, y separan el control-plane en un controlador lógicamente centralizado (aunque no físicamente, puesto que no escalaría). De esta manera, los dispositivos del data-plane de OpenFlow (y SDN en general) se convierten en dispositi-

tivos que acatan órdenes del controlador, y performan las acciones que se correspondan con dichas órdenes.

### Switches convencionales

Los switches convencionales comienzan con su tabla vacía, y son *plug-and-play*. Al recibir un paquete desde una dirección MAC por alguna de las interfaces presentes, se agrega esta interfaz, un tiempo de vida y la dirección MAC en la Switching Table. De este modo, se irá configurando la tabla. Es por esto que se dice que los switches convencionales son *self-learning*.

Para la estructura de su tabla: se guardan únicamente tres valores por entrada de la tabla: MAC destino, tiempo de vida e interfaz de salida. Cuando arriba un paquete, se busca una coincidencia de MAC destino, y se forwarda el paquete. Si no existe un match para la dirección destino, se realiza un broadcast del paquete a todas las interfaces, con excepción de la interfaz por la cual llegó el paquete.

Los switches convencionales no pueden implementar distintos comportamientos, se realizará la mismas acciones siempre, y si se quiere cambiar el comportamiento se deben recurrir a middleboxes.

Se aprecia que los switches convencionales adhieren al esquema de separación de capas de internet, y son una parte fundamental de él, por ubicarse **solamente** en la capa de enlace.

### Switches de OpenFlow

Los switches de OpenFlow dependen de ser configurados por el controlador.

La tabla de flujos de OpenFlow incluye:

- Un conjunto de campos de cabecera con los que se busca la correspondencia (ya no es solo la MAC destino para buscar el match).
- Contadores que sirven para obtener estadísticas de paquetes enviados, filtrados, inspeccionados, etc.
- Un conjunto de acciones de salida que hay que tomar cuando se da una correspondencia con una entrada: Estas acciones **no son estáticas**, es decir, un paquete entra a un switch y, según matchee una entrada u otra, será reenviado, inspeccionado, filtrado, cambiado ciertos campos en su cabecera.

Al arribar un paquete a un Switch de OpenFlow, se decide qué hacer con ese paquete de acuerdo a si tiene o no correspondencia con su **tabla de flujos** (reenvío generalizado). Un paquete que no hace match con ninguna entrada se envía al controlador SDN para que decida qué hacer con dicho paquete.

De esta manera, si se quieren implementar las mismas acciones que un switch convencional cuando no encuentra un match en su tabla, el controlador SDN debe enviarle al switch una acción del tipo **FLOOD**, indicando que se debe mandar el paquete a todas sus interfaces de salida, excepto la interfaz por la que arribó el paquete.

Los switches de OpenFlow (y cualquier dispositivo del data-plane en general de SDN) tienen comportamiento dinámico, es decir, pueden actuar como firewall (dropeando paquetes), cambiar campos de los paquetes entrantes, simplemente reenviarlos, entre otras acciones.

En OpenFlow, teniendo esta tabla de flujo con los campos de cabecera que corresponden a las tres capas de protocolos, no adhiere el esquema de capas de internet. Dado que el switch de OpenFlow toma una decisión con respecto a un flujo, el mismo opera sobre todas las capas, a diferencia del switch convencional que opera en la capa de enlace. En este sentido, se puede elegir qué hacer con un paquete teniendo en cuenta tanto la direcciones MAC de origen y destino, como también el protocolo de capa de aplicación o de transporte utilizado y las direcciones IP y Puerto origen y destino.

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

Dividiremos el análisis en dos: Analizando qué pasa dentro de un Sistema Autónomo, y analizando qué pasa en las fronteras de los mismos.

#### Escenario intra-ASes

**Dentro de un Sistema Autónomo** se pueden cambiar todos los routers convencionales por Switches OpenFlow.

Se tendrán los mismos comportamientos puesto que un Switch de OpenFlow es dinámico y actúa sobre varias capas. Además, como ventajas, se tendrá que se podrán eliminar las middleboxes que presente el Sistema Autónomo por completo, reduciendo su complejidad de mantenimiento; y será mucho más fácil definir nuevas políticas que los Switches deban aplicar.

El problema radica en el tamaño de dicho Sistema Autónomo, por lo que, si presenta una gran cantidad de dispositivos de red, se deberá contar con más de un controlador de OpenFlow (puesto que el principal problema de SDN es su escalabilidad).

De esta manera, sí se pueden reemplazar los routers convencionales dentro de un Sistema Autónomo, pero definiendo algún protocolo de comunicación entre los controladores de SDN (que no necesita ser el mismo para cada par de controladores, lo importante es que puedan intercambiar paquetes).

### Escenario inter-ASes

El escenario en la frontera de los Sistemas Autónomos es muy distinto: el principal problema radica en la funcionalidad de los routers de frontera (o *border-gateways routers*). Estos routers son una parte fundamental del protocolo BGP, que es el **único protocolo de comunicación entre Sistemas Autónomos**.

Al tener en cuenta que los Switches de OpenFlow necesitan sí o sí un controlador para funcionar, el reemplazar los routers de frontera por Switches de OpenFlow causará un problema crucial: Se deberá modificar el protocolo BGP para funcionar con estos Switches, o se deberá implementar un nuevo protocolo, que lo apliquen **todos los Switches, en el mismo momento**. Para remarcar el problema: Se deberá obligar a absolutamente todos los Sistemas Autónomos de Internet a implementar el mismo estándar de comunicación. Puesto que cada vez es más difícil cambiar un estándar, parece impracticable para el estado actual de internet.

Si no se cuenta con dicha estandarización para la comunicación, o algunos controladores no implementaron dicha funcionalidad en sus Switches que actúen en la frontera, el Sistema Autónomo quedará completamente aislado del resto.

Desde el punto de vista teórico parece posible, pero cabe destacar que (según se define en su paper de SDN: [1]), a día de hoy no existen implementaciones de SDN capaces de estandarizar estas comunicaciones, por lo que probablemente en caso de querer usar switches OpenFlow la mejor opción sería implementar controladores para todos los routers a excepción de los border-gateway, combinando los routers tradicionales con los Switches de OpenFlow.

Para finalizar esta pregunta, podemos concluir:

Se pueden reemplazar todos los routers convencionales de internet por Switches de OpenFlow desde el punto de vista teórico, aunque en la realidad no están dadas las tecnologías ni los estándares para que esto suceda, siendo su problema principal la comunicación entre los Sistemas Autónomos.

## 4. Desarrollo del trabajo

En esta sección se desarrollará cómo se estructuró nuestro trabajo desde el código, y pruebas de su funcionamiento utilizando capturas de terminal, wireshark, iperf y Spear.

### 4.1. Estructura del Código

En la siguiente imagen, mostramos qué clases generamos en nuestro trabajo.

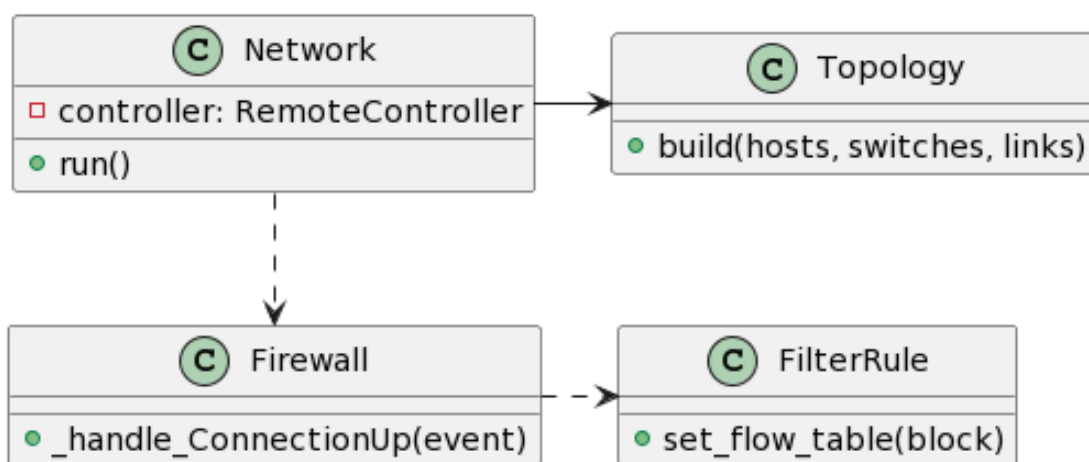


Figura 1: Diagrama de clases del modelo utilizado

Se puede apreciar que Firewall es el controlador de POX, al que la red de mininet se conecta a través de la instancia del *RemoteController* que provee mininet.

### Visualización de la topología con la herramienta Spear by Narmox

Tal como se indica en el enunciado, conectamos dos hosts,  $H_1$  y  $H_2$ , a un switch,  $S_1$ ; otros dos hosts,  $H_3$  y  $H_4$ , a otro switch  $S_2$  y generamos una cadena de cero o más switches  $S_i$  conectados linealmente entre  $S_1$  y  $S_2$ .

Mostramos a continuación dos visualizaciones de estas topologías. La primera, sin switches agregados a la cadena. Y la segunda, con cuatro switches en la cadena, y por lo tanto, seis switches en total.

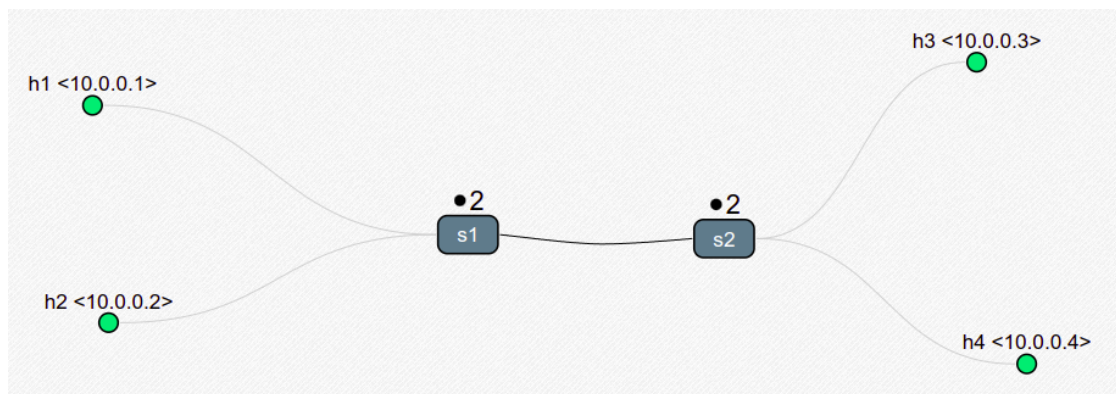


Figura 2: Visualización de la topología sin switches adicionales.

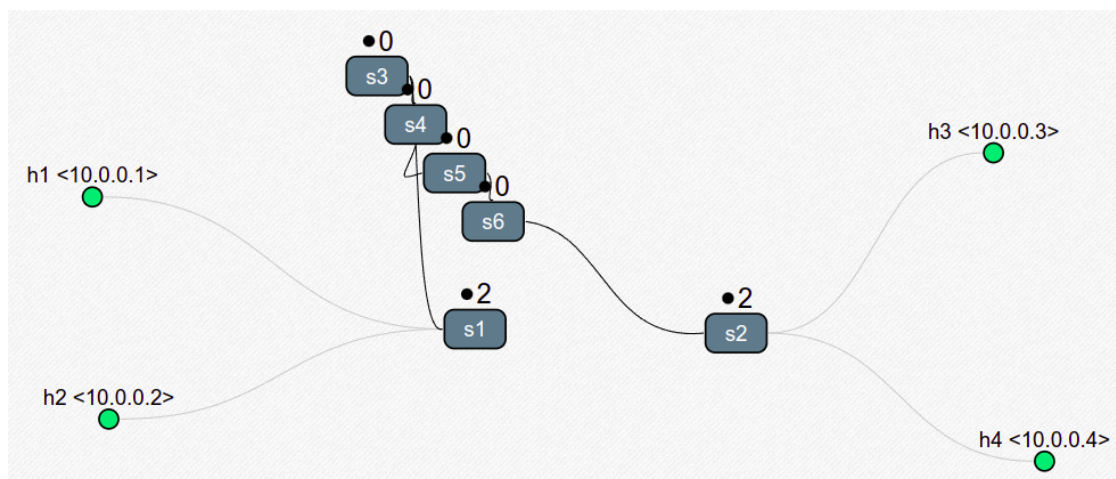


Figura 3: Visualización de la topología con cuatro switches adicionales

### Conectando un switch al controlador de POX

El controlador de POX (en nuestro caso, el Firewall) está constantemente escuchando eventos.

Una vez que se conecta un switch (por medio de una conexión TCP) al controlador de POX, el primer evento que se dispara en el controlador es el `ConnectionUp`. En nuestro Firewall, utilizando `_handle_ConnectionUp`, modificamos directamente la tabla de flujos de cada switch que se conecta con las reglas del enunciado.

De esta manera, tenemos como ventaja:



- Tenemos un lugar centralizado para setear la tabla de flujos de cada switch: Al conectarse cualquiera, el controlador de POX simplemente le manda la tabla de flujos con las reglas que definimos. Por ende, no tenemos que preocuparnos de en qué momento se configuran las entradas en un switch (dejando pasar algunos paquetes que no queríamos).
- Al no setearse un tiempo de vida para cada entrada de la tabla de flujos, la interfaz de POX permite que las entradas permanezcan durante toda la conexión, por ende, no necesitamos preocuparnos de renovar las entradas.
- En general, si se quiere implementar una acción, se debe especificarle a la tabla de flujos qué hacer (por ejemplo, definir algún output port, indicando droppeo del paquete, resend a algún puerto determinado, etc). La interfaz de POX permite que, si no se define una acción por defecto, simplemente filtrar *cada paquete* que matchee con esa entrada en la tabla de flujos.

A continuación, mostramos un diagrama de secuencia de cómo se conectan uno a uno los switches de mininet a los controladores, y los mensajes de inicio del protocolo entre cada switch y el controlador de OpenFlow.

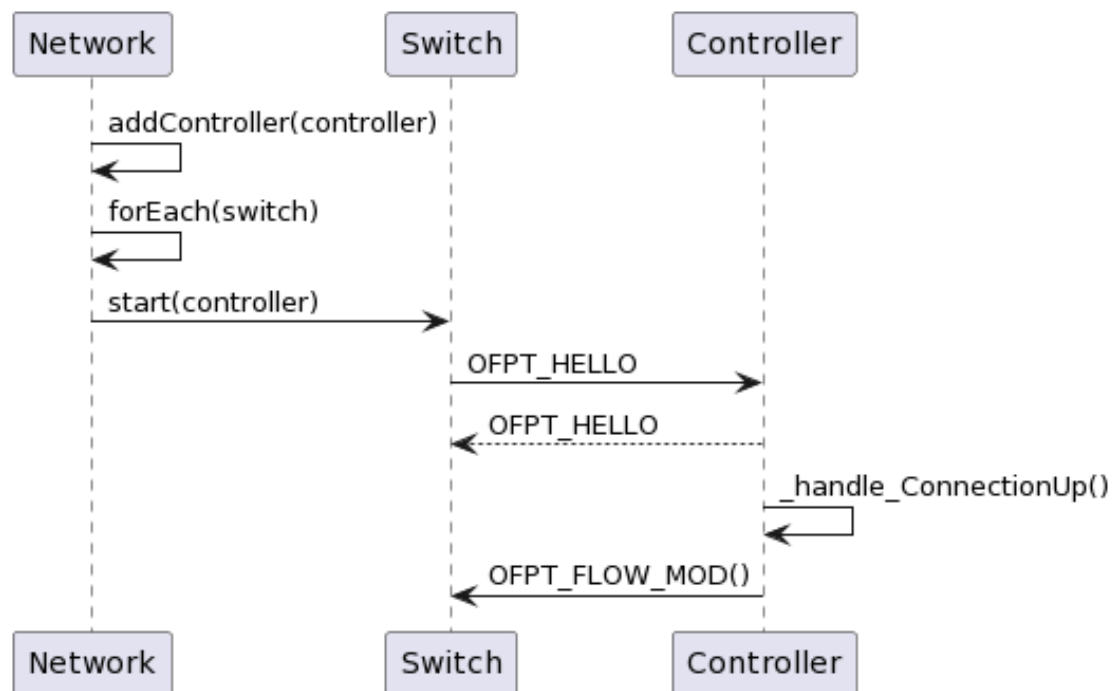


Figura 4: Diagrama de Secuencia: Inicialización de la red junto con el controlador remoto NOX.

### Mensaje entre dos hosts

En el siguiente diagrama de secuencias, mostramos un ping del host  $H_1$  al host  $H_2$ , conectados por un único switch  $S_1$ .

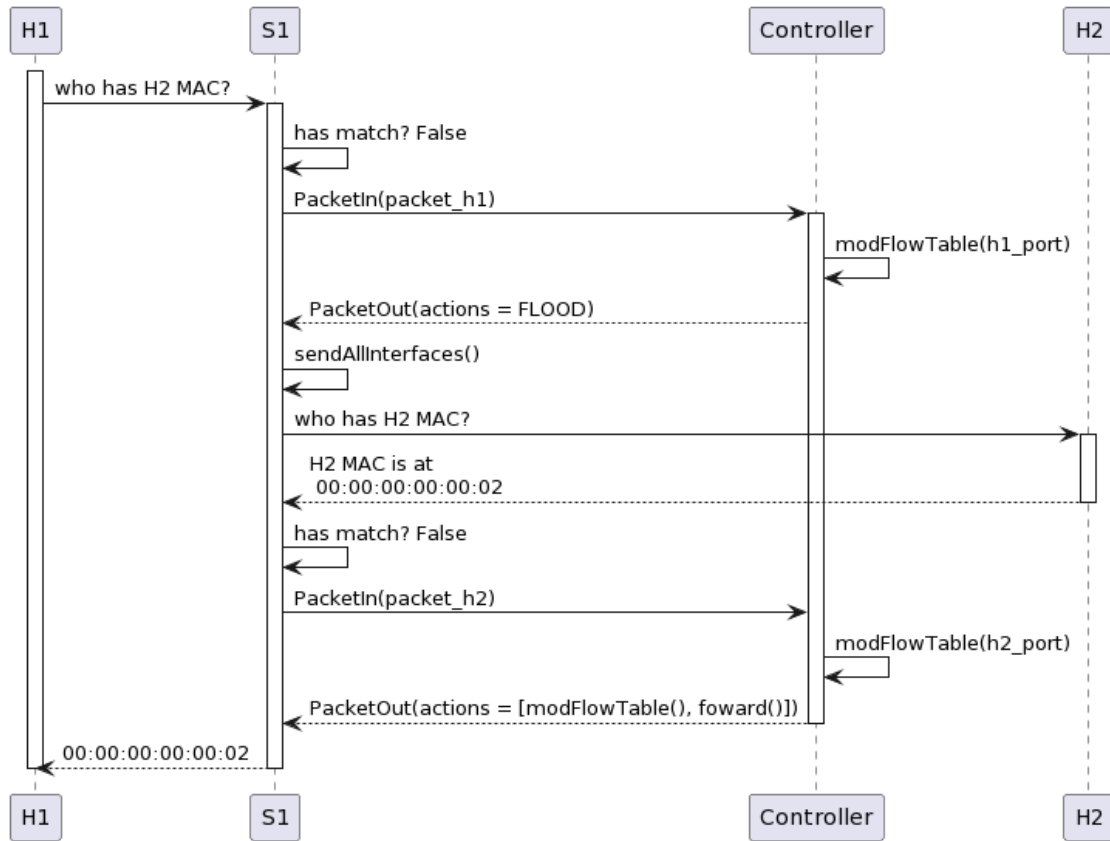


Figura 5: Diagrama de Secuencia: H1 busca la MAC de H2 para poder enviar el mensaje.

En primer lugar,  $H_1$  no encuentra en su tabla ARP la dirección MAC de 10.0.0.2 (ip de  $H_2$ ). Por lo tanto, envía un broadcast a todos sus switches mandando un ARP request, que es recibido solamente por  $S_1$ .

$S_1$  busca un match en su tabla de flujos. Al no encontrarlo dispara un evento al controlador de POX del tipo `PacketIn`. El controlador setea en la tabla de flujos (que se corresponde con este switch en particular) los datos de la dirección junto con su puerto de salida (por utilizar L2 Learning).

El controlador le devuelve al switch una acción del tipo *FLOOD*, indicando que debe mandarle a todas las interfaces (menos la de entrada) el paquete recibido de  $H_1$ .

$H_2$  recibe dicho paquete y le responde con su dirección MAC. Para ello, le manda al switch un mensaje desde  $H_2$  a  $H_1$ . El switch no tiene una entrada que matchee con estos campos en su tabla de flujos, por lo que vuelve a disparar el evento de paquete entrante en el controlador.

El controlador agrega una nueva entrada en la tabla con los datos de la dirección de  $H_2$  y su puerto de salida. Por último, el controlador le retorna al switch dos acciones: Una indicándole que debe agregar en la tabla de flujos (cargada en la memoria del switch) una entrada correspondida al flujo  $H_2 \rightarrow H_1$  y la interfaz de salida de  $H_1$ . La otra, indicando el reenvío del paquete proveniente de  $H_2$  hacia la interfaz de salida que se acaba de añadir en la tabla: la de  $H_2 \rightarrow H_1$ .

Por último,  $S_1$  reenvía el paquete a  $H_1$ , quien guarda en su tabla ARP la dirección MAC de  $H_2$ .

Ahora sí, una vez que el host  $H_1$  obtuvo la dirección MAC de  $H_2$ ,  $H_1$  envía el PING request. Esta secuencia se muestra a continuación.

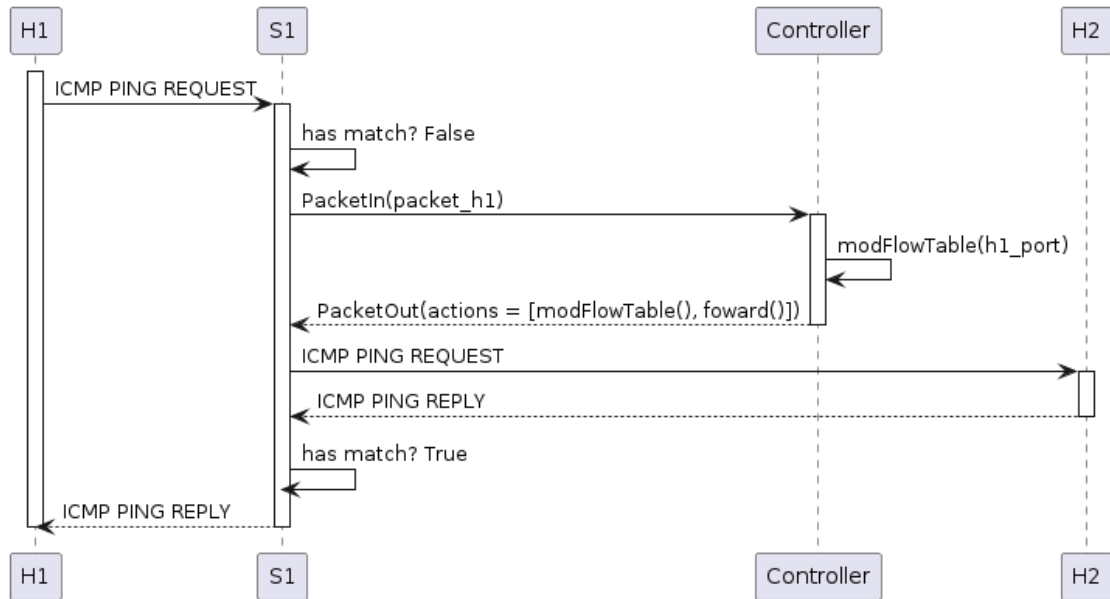


Figura 6: Diagrama de Secuencia: Ping Request de H1 a H2.

Tal como se observa en el diagrama,  $H_1$  envía el PING REQUEST a través de  $S_1$ .  $S_1$  no tiene una entrada en su tabla de flujos para el flujo que se corresponde con  $H_1 \rightarrow H_2$ . Por lo tanto, envía un mensaje **PacketIn** al controlador.

El controlador actualiza su tabla con la interfaz de salida de  $H_1$  nuevamente, pisando la anterior entrada que se correspondía con este host. Luego, al notar que existe una entrada para la salida  $H_2$ , en el mensaje **PacketOut**, se le indica a  $S_1$  que debe agregar el flujo  $H_1 \rightarrow H_2$  junto con el puerto de salida correspondiente a la llegada a  $H_2$ . Además, se le señala que debe reenviar el paquete.

$H_2$  recibe el PING REQUEST y lo contesta con un mensaje del tipo PING REPLY. El switch tiene ya los flujos almacenados y realiza un forward del paquete sin la necesidad de consultar al controlador.

Este último ejemplo se desarrollará en la siguiente subsección de resultados, mostrando capturas de wireshark que se corresponden con los diagramas de secuencia expuestos.

## 4.2. Resultados de las simulaciones

En la presente sección mostraremos tanto capturas de wireshark como distintos logs del controlador (en la terminal), evidenciando el correcto funcionamiento de la red, y mostrando algunos mensajes claves que se envían entre el controlador OpenFlow y los switches.

Vale aclarar que las capturas presentadas se realizaron con distintas corridas de mininet. Por lo tanto, no siempre se conservan los puertos de los switches, que no tiene ninguna incidencia en la lógica esencial del resultado.

### Inicialización de mininet sin el controlador de POX

Inicializando mininet sin un controlador de POX (pero se inicializa con un controlador remoto), los switches se intentan comunicar mediante una conexión TCP, realizando el SYN del three-way-handshake al puerto 6633 (o el puerto donde se vaya a levantar el controlador de POX). Se mandan constantemente paquetes sin respuesta, como se aprecia en la siguiente captura de wireshark.

127.0.0.1	127.0.0.1	TCP	74 46394 → 6633 [SYN] Seq=0 Win=43690 Len=0 M...
127.0.0.1	127.0.0.1	TCP	54 6633 → 46394 [RST, ACK] Seq=1 Ack=1 Win=0 ...
127.0.0.1	127.0.0.1	TCP	74 46396 → 6633 [SYN] Seq=0 Win=43690 Len=0 M...
127.0.0.1	127.0.0.1	TCP	54 6633 → 46396 [RST, ACK] Seq=1 Ack=1 Win=0 ...
127.0.0.1	127.0.0.1	TCP	74 46398 → 6633 [SYN] Seq=0 Win=43690 Len=0 M...
127.0.0.1	127.0.0.1	TCP	54 6633 → 46398 [RST, ACK] Seq=1 Ack=1 Win=0 ...

Figura 7: Los switches intentan conectarse al controlador remoto.

### Conectando el controlador de POX

Una vez que se levanta el controlador (corriendo el script que entregamos junto al código), el cual escucha conexiones en el puerto especificado, se establece una conexión TCP por cada switch existente en la topología. Mostramos a continuación el caso de una topología con dos switches. Por lo tanto, se observan dos conexiones TCP establecidas.

127.0.0.1	127.0.0.1	TCP	74 46400 → 6633 [SYN] Seq=0 Win=43690 Len=0 M...
127.0.0.1	127.0.0.1	TCP	74 6633 → 46400 [SYN, ACK] Seq=0 Ack=1 Win=43...
127.0.0.1	127.0.0.1	TCP	66 46400 → 6633 [ACK] Seq=1 Ack=1 Win=44032 L...
127.0.0.1	127.0.0.1	Open...	74 Type: OFPT_HELLO
127.0.0.1	127.0.0.1	TCP	66 6633 → 46400 [ACK] Seq=1 Ack=9 Win=44032 L...
127.0.0.1	127.0.0.1	TCP	74 46402 → 6633 [SYN] Seq=0 Win=43690 Len=0 M...
127.0.0.1	127.0.0.1	TCP	74 6633 → 46402 [SYN, ACK] Seq=0 Ack=1 Win=43...
127.0.0.1	127.0.0.1	TCP	66 46402 → 6633 [ACK] Seq=1 Ack=1 Win=44032 L...

Figura 8: Se establecen las conexiones TCP entre los switches y el controlador.

### Saludo inicial del protocolo OpenFlow

Una vez que se establece una conexión TCP por cada switch, se inicia el protocolo OpenFlow con un mensaje de handshake (enviado primero del switch al controlador, luego del controlador al switch que se acaba de conectar), enviando mensajes del tipo OFPT HELLO.

OpenFl...	74 Type: OFPT_HELLO
TCP	66 6633 → 57076 [ACK] Seq=1 Ack=9 Win=44032 Len=0 TSval=25304225...
OpenFl...	74 Type: OFPT_HELLO
TCP	66 57076 → 6633 [ACK] Seq=9 Ack=9 Win=44032 Len=0 TSval=25304225...
OpenFl...	86 Type: OFPT_STATS_REQUEST
TCP	66 57076 → 6633 [ACK] Seq=9 Ack=29 Win=44032 Len=0 TSval=2530422...
OpenFl...	290 Type: OFPT_FEATURES_REPLY
TCP	66 6633 → 57076 [ACK] Seq=29 Ack=233 Win=44032 Len=0 TSval=25304...
OpenFl...	1134 Type: OFPT_STATS_REPLY

Figura 9: Inicio del protocolo OpenFlow con mensajes OFPT HELLO.

### Configuración inicial de la tabla de flujos

Al terminar el saludo del protocolo OpenFlow, el controlador de POX empezará a configurar las reglas iniciales mediante el protocolo de OpenFlow, que definimos en nuestra implementación del controlador de POX (que actuará como firewall en nuestro caso).

Estos mensajes son enviados a través de mensajes del tipo OFPT\_FLOW\_MOD. A continuación mostramos algunos de estos mensajes, los cuales permiten que se filtren aquellos paquetes que cumplen con las reglas que imponemos.

```

▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FLOW_MOD (14)
  Length: 72
  Transaction ID: 8
  Wildcards: 1048607
  In port: 0
  Ethernet source address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Ethernet destination address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Input VLAN id: 0
  Input VLAN priority: 0
  Pad: 00
  D1 type: 0
  IP ToS: 0
  IP protocol: 0
  Pad: 0000
  Source Address: 0.0.0.0
  Destination Address: 0.0.0.0
  Source Port: 0
  Destination Port: 0
  Cookie: 0x0000000000000000
  Command: Delete all matching flows (3)
  Idle time-out: 0
  hard time-out: 0
  Priority: 32768
  Buffer Id: 0xffffffff
  Out port: 65535
  Flags: 0

```

Figura 10: Configuración Inicial: eliminar todas las entradas en la tabla de flujos

El primer mensaje de configuración que se envía le indica al switch que deben remover todas las entradas de su tabla de flujos. Esto es así, para que no queden configuraciones antiguas almacenadas.

A continuación, se muestra el seteo de las reglas pedidas en el enunciado. Se resaltan cuáles son los campos más relevantes para cada regla.

```
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FLOW_MOD (14)
  Length: 72
  Transaction ID: 23
  Wildcards: 1048623
  In port: 0
  Ethernet source address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Ethernet destination address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Input VLAN id: 0
  Input VLAN priority: 0
  Pad: 00
  D1 type: 2054
  IP ToS: 0
  IP protocol: 0
  Pad: 0000
  Source Address: 10.0.0.1
  Destination Address: 10.0.0.4
  Source Port: 0
  Destination Port: 0
  Cookie: 0x0000000000000000
  Command: New flow (0)
  Idle time-out: 0
  hard time-out: 0
  Priority: 32768
  Buffer Id: 0xffffffff
  Out port: 65535
  Flags: 0
```

Figura 11: Regla para filtrar por Source Address 10.0.0.1 y Destination Address 10.0.0.4, protocolo ARP

```
▼ OpenFlow 1.0
.000 0001 = Version: 1.0 (0x01)
Type: OFPT_FLOW_MOD (14)
Length: 72
Transaction ID: 24
Wildcards: 1048623
In port: 0
Ethernet source address: 00:00:00_00:00:00 (00:00:00:00:00:00)
Ethernet destination address: 00:00:00_00:00:00 (00:00:00:00:00:00)
Input VLAN id: 0
Input VLAN priority: 0
Pad: 00
Dl type: 2054
IP ToS: 0
IP protocol: 0
Pad: 0000
Source Address: 10.0.0.4
Destination Address: 10.0.0.1
Source Port: 0
Destination Port: 0
Cookie: 0x0000000000000000
Command: New flow (0)
Idle time-out: 0
hard time-out: 0
Priority: 32768
Buffer Id: 0xffffffff
Out port: 65535
Flags: 0
```

Figura 12: Regla para filtrar por Source Address 10.0.0.4 y Destination Address 10.0.0.1, protocolo ARP

Para inhabilitar la comunicación entre  $H_1$  y  $H_4$ , decidimos filtrar los mensajes ARP que se podrían envír para conocer la dirección MAC del respectivo host. De este modo, si no conocen la MAC address del otro, no se podrán comunicar.

Se aprecia la diferencia entre ARP e IP viendo el campo `dl_type` con disintos valores en cada ocasión:

- 2048 en el caso de IP.
- 2054 en el caso de ARP.

```
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FLOW_MOD (14)
  Length: 72
  Transaction ID: 14
  Wildcards: 3145775
  In port: 0
  Ethernet source address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Ethernet destination address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Input VLAN id: 0
  Input VLAN priority: 0
  Pad: 00
  D1 type: 2048
  IP ToS: 0
  IP protocol: 0
  Pad: 0000
  Source Address: 10.0.0.1
  Destination Address: 10.0.0.4
  Source Port: 0
  Destination Port: 0
  Cookie: 0x0000000000000000
  Command: New flow (0)
  Idle time-out: 0
  hard time-out: 0
  Priority: 32768
  Buffer Id: 0xffffffff
  Out port: 65535
  Flags: 0
```

Figura 13: Regla para filtrar por Source Address 10.0.0.1 y Destination Address 10.0.0.4, protocolo IP



```

▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FLOW_MOD (14)
  Length: 72
  Transaction ID: 15
  Wildcards: 3145775
  In port: 0
  Ethernet source address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Ethernet destination address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Input VLAN id: 0
  Input VLAN priority: 0
  Pad: 00
  D1 type: 2048
  IP ToS: 0
  IP protocol: 0
  Pad: 0000
  Source Address: 10.0.0.4
  Destination Address: 10.0.0.1
  Source Port: 0
  Destination Port: 0
  Cookie: 0x0000000000000000
  Command: New flow (0)
  Idle time-out: 0
  hard time-out: 0
  Priority: 32768
  Buffer Id: 0xffffffff
  Out port: 65535
  Flags: 0
  
```

Figura 14: Regla para filtrar por Source Address 10.0.0.4 y Destination Address 10.0.0.1, protocolo IP

Suponiendo un caso en el que uno de los dos,  $H_1$  o  $H_4$ , conoce la dirección MAC del otro, y por lo tanto puede enviarle un mensaje, filtramos por el protocolo IP, incomunicando todos los paquetes entre ambas direcciones IP.

```
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FLOW_MOD (14)
  Length: 72
  Transaction ID: 16
  Wildcards: 3678243
  In port: 0
  Ethernet source address: 00:00:00_00:00:04 (00:00:00:00:00:04)
  Ethernet destination address: 00:00:00_00:00:01 (00:00:00:00:00:01)
  Input VLAN id: 0
  Input VLAN priority: 0
  Pad: 00
  D1 type: 2048
  IP ToS: 0
  IP protocol: 0
  Pad: 0000
  Source Address: 0.0.0.0
  Destination Address: 0.0.0.0
  Source Port: 0
  Destination Port: 0
  Cookie: 0x0000000000000000
  Command: New flow (0)
  Idle time-out: 0
  hard time-out: 0
  Priority: 32768
  Buffer Id: 0xffffffff
  Out port: 65535
  Flags: 0
```

Figura 15: Regla para filtrar por Ethernet Source Address 00:00:00\_00:00:04 y Ethernet Destination Address 00:00:00\_00:00:01

```
OpenFlow 1.0
.000 0001 = Version: 1.0 (0x01)
Type: OFPT_FLOW_MOD (14)
Length: 72
Transaction ID: 17
Wildcards: 3678243
In port: 0
Ethernet source address: 00:00:00_00:00:01 (00:00:00:00:00:01)
Ethernet destination address: 00:00:00_00:00:04 (00:00:00:00:00:04)
Input VLAN id: 0
Input VLAN priority: 0
Pad: 00
Dl type: 2048
IP ToS: 0
IP protocol: 0
Pad: 0000
Source Address: 0.0.0.0
Destination Address: 0.0.0.0
Source Port: 0
Destination Port: 0
Cookie: 0x0000000000000000
Command: New flow (0)
Idle time-out: 0
hard time-out: 0
Priority: 32768
Buffer Id: 0xffffffff
Out port: 65535
Flags: 0
```

Figura 16: Regla para filtrar por Ethernet Source Address 00:00:00\_00:00:01 y Ethernet Destination Address 00:00:00\_00:00:04

A pesar de que en esta simulación las direcciones IP del host 1 y del host 2 no cambiarán, en un caso real, las direcciones IP son dinámicas. Es por esto que decidimos filtrar también las direcciones MAC de ambos hosts, las cuales son estáticas y supuestamente únicas para cada dispositivo.

```

▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FLOW_MOD (14)
  Length: 72
  Transaction ID: 18
  Wildcards: 3670095
  In port: 0
  Ethernet source address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Ethernet destination address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Input VLAN id: 0
  Input VLAN priority: 0
  Pad: 00
  D1 type: 2048
  IP ToS: 0
  IP protocol: 17
  Pad: 0000
  Source Address: 10.0.0.1
  Destination Address: 0.0.0.0
  Source Port: 0
  Destination Port: 5001
  Cookie: 0x0000000000000000
  Command: New flow (0)
  Idle time-out: 0
  hard time-out: 0
  Priority: 32768
  Buffer Id: 0xffffffff
  Out port: 65535
  Flags: 0
  
```

Figura 17: Regla para filtrar por Source Address 10.0.0.1 y Destination Port 5001 con protocolo UDP

Para satisfacer la segunda regla del enunciado, filtramos también por el puerto 5001, junto con la ip de  $H_1$  para el protocolo UDP.

```
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FLOW_MOD (14)
  Length: 72
  Transaction ID: 19
  Wildcards: 3678287
  In port: 0
  Ethernet source address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Ethernet destination address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Input VLAN id: 0
  Input VLAN priority: 0
  Pad: 00
  D1 type: 2048
  IP ToS: 0
  IP protocol: 6
  Pad: 0000
  Source Address: 0.0.0.0
  Destination Address: 0.0.0.0
  Source Port: 0
  Destination Port: 80
  Cookie: 0x0000000000000000
  Command: New flow (0)
  Idle time-out: 0
  hard time-out: 0
  Priority: 32768
  Buffer Id: 0xffffffff
  Out port: 65535
  Flags: 0
```

Figura 18: Regla Para Filtrar Paquetes TCP en puerto destino 80

```
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FLOW_MOD (14)
  Length: 72
  Transaction ID: 20
  Wildcards: 3678287
  In port: 0
  Ethernet source address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Ethernet destination address: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Input VLAN id: 0
  Input VLAN priority: 0
  Pad: 00
  D1 type: 2048
  IP ToS: 0
  IP protocol: 17
  Pad: 0000
  Source Address: 0.0.0.0
  Destination Address: 0.0.0.0
  Source Port: 0
  Destination Port: 80
  Cookie: 0x0000000000000000
  Command: New flow (0)
  Idle time-out: 0
  hard time-out: 0
  Priority: 32768
  Buffer Id: 0xffffffff
  Out port: 65535
  Flags: 0
```

Figura 19: Regla Para Filtrar Paquetes UDP en puerto destino 80

Por último, filtramos para ambos protocolos de transporte, UDP y TCP, el puerto destino 80.

### Finalización de seteo inicial

Una vez que esta primera etapa finaliza, las tablas de flujo de los switches contienen la información suficiente para que si llega un paquete que se debe filtrar por las políticas que implementamos, no se deba consultar con el controlador. Lo podrá descartar automáticamente.

Las capturas de wireshark referidas al seteo inicial se contrastan con los logs del controlador que agregamos en el código, los cuales mostramos en la siguiente imagen.

```

DEBUG:misc.firewall:Installing rules...

DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.1, IP dst 10.0.0.4, protocol=ARP
DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.4, IP dst 10.0.0.1, protocol=ARP
DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.1, IP dst 10.0.0.4, protocol=IP
DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.4, IP dst 10.0.0.1, protocol=IP
DEBUG:misc.firewall:Rule sent: Dropping: MAC src 00:00:00:00:00:04, MAC dst 00:00:00:00:00:01
DEBUG:misc.firewall:Rule sent: Dropping: MAC src 00:00:00:00:00:01, MAC dst 00:00:00:00:00:04
DEBUG:misc.firewall:Rule sent: Dropping: UDP port 5001 for messages sent by host 10.0.0.1
DEBUG:misc.firewall:Rule sent: Dropping: PORT dst 80, protocol=UDP
DEBUG:misc.firewall:Rule sent: Dropping: PORT dst 80, protocol=TCP
DEBUG:misc.firewall:All rules sent.

INFO:openflow.of_01:[00-00-00-00-02 2] connected
DEBUG:forwarding_l2_learning:Connection [00-00-00-00-02 2]
DEBUG:misc.firewall:Installing rules...

DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.1, IP dst 10.0.0.4, protocol=ARP
DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.4, IP dst 10.0.0.1, protocol=ARP
DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.1, IP dst 10.0.0.4, protocol=IP
DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.4, IP dst 10.0.0.1, protocol=IP
DEBUG:misc.firewall:Rule sent: Dropping: MAC src 00:00:00:00:00:04, MAC dst 00:00:00:00:00:01
DEBUG:misc.firewall:Rule sent: Dropping: MAC src 00:00:00:00:00:01, MAC dst 00:00:00:00:00:04
DEBUG:misc.firewall:Rule sent: Dropping: UDP port 5001 for messages sent by host 10.0.0.1
DEBUG:misc.firewall:Rule sent: Dropping: PORT dst 80, protocol=UDP
DEBUG:misc.firewall:Rule sent: Dropping: PORT dst 80, protocol=TCP
DEBUG:misc.firewall:All rules sent.

```

Figura 20: Captura de logs del controlador: Instalación de reglas

Se aprecia que se repite dos veces la instalación puesto que hay solo dos switches.

### Mensajes entre hosts

A continuación, mostraremos cómo se modifica la tabla de flujos (en este caso,  $S_1$ ) a partir del L2 learning, cuando se mandan mensajes entre dos hosts ( $H_1$  y  $H_2$ ). Ejecutamos un ping desde  $H_1$  a  $H_2$ , y se aprecia que los resultados se equivalen con lo mostrado en los respectivos diagramas de secuencia mostrados en la sección anterior.

En este caso,  $H_1$  tiene dirección ip 10.0.0.1, y  $H_2$  tiene dirección ip 10.0.0.2, con  $S_1$  como único switch que los conecta.

A continuación, se muestran los mensajes que aparecen en el switch, visto desde wireshark.

00:00:00_00:00:01	Broadcast	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:01	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT
127.0.0.1	127.0.0.1	TCP	66 57420 → 6633 [ACK] Seq=149 Ack=161 Win=86 Len=0 TSval=2531951...
00:00:00_00:00:02	00:00:00_00:00:01	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	00:00:00_00:00:01	OpenFl...	220 Type: OFPT_PACKET_OUT

Figura 21: Mensajes iniciales entre el Switch y el controlador: H1 hace una consulta ARP.

Los primeros dos mensajes de la captura son el manejo del mensaje de  $H_1$  al switch, que mostramos a continuación.

En el primer caso, el switch le está notificando al controlador que llegó un paquete (y no matcheó con la tabla de flujos, entonces es permitido, levantando el evento de `_handle_PacketIn`). Se aprecia que se quiere mandar el mensaje a una dirección de broadcast, puesto que  $H_1$  no tiene la dirección MAC de 10.0.0.2, entonces, por el protocolo ARP, mandará un broadcast hasta que alguien le conteste con la dirección MAC solicitada.

```
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_PACKET_IN (10)
  Length: 60
  Transaction ID: 0
  Buffer Id: 0xffffffff
  Total length: 42
  In port: 1
  Reason: No matching flow (table-miss flow entry) (0)
  Pad: 00
  ▼ Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
    ▶ Destination: Broadcast (ff:ff:ff:ff:ff:ff)
    ▶ Source: 00:00:00_00:00:01 (00:00:00:00:00:01)
    Type: ARP (0x0806)
  ▼ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: 00:00:00_00:00:01 (00:00:00:00:00:01)
    Sender IP address: 10.0.0.1
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 10.0.0.2
```

Figura 22: Mensaje Packet In de S1 al Controlador, indicando que no tiene entrada para el flujo correspondiente.

En el segundo caso, el controlador le devuelve al switch un mensaje con un output port FLOOD indicándole al switch que debe el paquete por todas sus interfaces, menos la interfaz de entrada.

```
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_PACKET_OUT (13)
  Length: 66
  Transaction ID: 173
  Buffer Id: 0xffffffff
  In port: 1
  Actions length: 8
  Actions type: Output to switch port (0)
  Action length: 8
  Output port: 65531
  Max length: 0
  ▼ Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
    ▶ Destination: Broadcast (ff:ff:ff:ff:ff:ff)
    ▶ Source: 00:00:00_00:00:01 (00:00:00:00:00:01)
    Type: ARP (0x0806)
  ▼ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: 00:00:00_00:00:01 (00:00:00:00:00:01)
    Sender IP address: 10.0.0.1
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 10.0.0.2
```

Figura 23: Mensaje Packet Out indicando que haga FLOOD (Output Port = 65531).

Al mandar el mensaje de broadcast a todas las interfaces de salida, el host  $H_2$  recibe el mensaje y contesta con su dirección MAC. Por ende, al switch  $S_1$  le llega un mensaje proveniente de  $H_2$  con su dirección MAC. Este mensaje se manejará con los eventos *PacketIn* (del switch al controlador) y *PacketOut* (del controlador al switch).

En la siguiente imagen se muestra el evento de paquete entrante (del switch al controlador, sin match en la tabla de flujos), proveniente de la MAC de  $H_2$  con destino a la MAC de  $H_1$ .



```
Transmission Control Protocol, Src Port: 57420, Dst Port: 8080, Seq: 149, Ack: 101, Len: 60
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_PACKET_IN (10)
  Length: 60
  Transaction ID: 0
  Buffer Id: 0xffffffff
  Total length: 42
  In port: 2
  Reason: No matching flow (table-miss flow entry) (0)
  Pad: 00
  ▼ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
    ▶ Destination: 00:00:00_00:00:01 (00:00:00:00:00:01)
    ▶ Source: 00:00:00_00:00:02 (00:00:00:00:00:02)
    Type: ARP (0x0806)
  ▼ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: 00:00:00_00:00:02 (00:00:00:00:00:02)
    Sender IP address: 10.0.0.2
    Target MAC address: 00:00:00_00:00:01 (00:00:00:00:00:01)
    Target IP address: 10.0.0.1
```

Figura 24: Packet In: No hay entrada en la tabla de flujos de S1.

Luego, el controlador le devuelve al switch un mensaje con dos acciones: Una acción de mandar el mensaje a un output port (*Accions Type: "Output to switch port (0)"*), y otra acción de guardar la entrada en la tabla de flujos (*'Command: New Flow (0)'*), con un tiempo de vida.

```
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FLOW_MOD (14)
  Length: 80
  Transaction ID: 131
  Wildcards: 0
  In port: 2
  Ethernet source address: 00:00:00_00:00:02 (00:00:00:00:00:02)
  Ethernet destination address: 00:00:00_00:00:01 (00:00:00:00:00:01)
  Input VLAN id: 65535
  Input VLAN priority: 0
  Pad: 00
  D1 type: 2054
  IP ToS: 0
  IP protocol: 2
  Pad: 0000
  Source Address: 10.0.0.2
  Destination Address: 10.0.0.1
  Source Port: 0
  Destination Port: 0
  Cookie: 0x0000000000000000
  Command: New flow (0)
  Idle time-out: 10
  hard time-out: 30
  Priority: 32768
  Buffer Id: 0xffffffff
  Out port: 65535
  Flags: 0
  ▶ OpenFlow 1.0
  ▼ OpenFlow 1.0
    .000 0001 = Version: 1.0 (0x01)
    Type: OFPT_PACKET_OUT (13)
    Length: 66
    Transaction ID: 133
    Buffer Id: 0xffffffff
    In port: 2
    Actions length: 8
    Actions type: Output to switch port (0)
    Action length: 8
    Output port: 65529
    Max length: 0
  ▼ Ethernet II, Src: 00:00:00_00:00:02 (00:00:00:00:00:02), Dst: 00:00:00_00:00:01 (00:00:00:00:00:01)
    ▶ Destination: 00:00:00_00:00:01 (00:00:00:00:00:01)
```

Figura 25: Packet Out: Acciones de agregar entrada y reenviar paquete.

En la captura no se puede observar el puerto de salida para el nuevo flujo que se agrega a la tabla del switch. Esta es visible si se expande el campo del payload en WireShark.

Por otro lado, para la acción que le indica el reenvío del paquete, se puede observar que el output port es 65529, el cual se corresponde con el puerto reservado de openflow, `OFPP_TABLE`. Esto le indica que debe buscar en el inicio del pipeline, donde se indica la interfaz de salida.

Una vez que le llega el paquete de ARP Reply a  $H_1$ ,  $H_1$  envía el PING REQUEST, tal como se muestra en la siguiente captura.

```
Internet Control Message Protocol
  Type: 8 (Echo (ping) request)
  Code: 0
  Checksum: 0x2806 [correct]
  [Checksum Status: Good]
  Identifier (BE): 22804 (0x5914)
  Identifier (LE): 5209 (0x1459)
  Sequence number (BE): 1 (0x0001)
  Sequence number (LE): 256 (0x0100)
  [Response frame: 17]
  Timestamp from icmp data: Jul 1, 2022 00:32:05.000000000 -03
  [Timestamp from icmp data (relative): 0.611795728 seconds]
  Data (48 bytes)
```

Figura 26: Ping Request de  $H_1$  a  $H_2$

Arriba el Ping Request a  $S_1$ . Dado que el flujo  $H_1 \rightarrow H_2$  no tiene match en la tabla de flujos de  $S_1$ , se manda un `PacketIn` al Controlador.

```
OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_PACKET_IN (10)
  Length: 116
  Transaction ID: 0
  Buffer Id: 0xffffffff
  Total length: 98
  In port: 1
  Reason: No matching flow (table-miss flow entry) (0)
  Pad: 00
  Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
  Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.2
  Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x2806 [correct]
    [Checksum Status: Good]
    Identifier (BE): 22804 (0x5914)
    Identifier (LE): 5209 (0x1459)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
    [No response seen]
    Timestamp from icmp data: Jul 1, 2022 00:32:05.000000000 -03
    [Timestamp from icmp data (relative): 0.612481576 seconds]
    Data (48 bytes)
```

Figura 27: Packet In: No hay match para el flujo  $H_1$  a  $H_2$ .

El controlador sí tiene una interfaz que se corresponde con  $H_2$  para la tabla que él almacena. Por lo tanto, le indica a  $H_1$  que agregue el flujo junto con esta interfaz en la tabla del switch y que forwardee el paquete en esta interfaz.

```

▶ Transmission Control Protocol, Src Port: 6633, Dst Port: 41084, Seq: 1028, Ack: 1010, Len: 210
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FLOW_MOD (14)
  Length: 80
  Transaction ID: 135
  Wildcards: 0
  In port: 1
  Ethernet source address: 00:00:00_00:00:01 (00:00:00:00:00:01)
  Ethernet destination address: 00:00:00_00:00:02 (00:00:00:00:00:02)
  Input VLAN id: 65535
  Input VLAN priority: 0
  Pad: 00
  D1 type: 2048
  IP ToS: 0
  IP protocol: 1
  Pad: 0000
  Source Address: 10.0.0.1
  Destination Address: 10.0.0.2
  Source Port: 8
  Destination Port: 0
  Cookie: 0x0000000000000000
  Command: New flow (0)
  Idle time-out: 10
  hard time-out: 30
  Priority: 32768
  Buffer Id: 0xffffffff
  Out port: 65535
  Flags: 0
▶ OpenFlow 1.0
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_PACKET_OUT (13)
  Length: 122
  Transaction ID: 137
  Buffer Id: 0xffffffff
  In port: 1
  Actions length: 8
  Actions type: Output to switch port (0)
  Action length: 8
  Output port: 65529
  Max length: 0
▶ Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:02 (00:00:00:00:00:02)
▶ Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.2

```

Figura 28: Packet Out: acciones de guardado en tabla de flujo y forwarding del paquete.

Una vez que llega el paquete a  $H_2$ , se crea un paquete PING REPLY, el cual es enviado a  $H_1$  sin la necesidad de consultar con el controlador, dado que el flujo  $H_2 \rightarrow H_1$  tiene una correspondencia en la tabla de flujos. Mostramos a continuación la estructura de este paquete ICMP.

```

Internet Control Message Protocol
Type: 0 (Echo (ping) reply)
Code: 0
Checksum: 0x3006 [correct]
[Checksum Status: Good]
Identifier (BE): 22804 (0x5914)
Identifier (LE): 5209 (0x1459)
Sequence number (BE): 1 (0x0001)
Sequence number (LE): 256 (0x0100)
[Request frame: 16]
[Response time: 6,878 ms]
Timestamp from icmp data: Jul  1, 2022 00:32:05.000000000 -03
[Timestamp from icmp data (relative): 0.618673378 seconds]
▶ Data (48 bytes)

```

Figura 29: Paquete Ping Reply enviado desde H2 a H1.

Las anteriores capturas de wireshark se contrastan con los logs del controlador entre un ping de  $H_1$  y  $H_2$ .

```
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 1: 00:00:00:00:00:01 --> ff:ff:ff:ff:ff:ff
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 2: 00:00:00:00:00:02 --> 00:00:00:00:00:01
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:02.2 -> 00:00:00:00:00:01.1
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-02 interface 3: 00:00:00:00:00:01 --> ff:ff:ff:ff:ff:ff
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 1: 00:00:00:00:00:01 --> 00:00:00:00:00:02
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:01.1 -> 00:00:00:00:00:02.2
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 2: 00:00:00:00:00:02 --> 00:00:00:00:00:01
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:02.2 -> 00:00:00:00:00:01.1
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 2: 00:00:00:00:00:02 --> 00:00:00:00:00:01
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:02.2 -> 00:00:00:00:00:01.1
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 1: 00:00:00:00:00:01 --> 00:00:00:00:00:02
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:01.1 -> 00:00:00:00:00:02.2
INFO:openflow.of_01:[00-00-00-00-00-01 1] closed
INFO:openflow.of_01:[00-00-00-00-00-02 2] closed
```

Figura 30: Captura de logs del controlador: Ping entre  $H_1$  y  $H_2$ .

En esta captura, se puede visualizar que también le llegan paquetes al segundo switch, pese a no estar conectado a  $H_1$  y  $H_2$ . Esto ocurre puesto que  $S_1$  hace un FLOOD del paquete proveniente de  $H_1$  y este le llega tanto a  $H_2$  como a  $S_2$ .

$S_2$ , al no encontrar un match en su tabla de flujos, le enviará el paquete para que resuelva el controlador, y este le indicará que realice un FLOOD del paquete entrante (menos a la interfaz de entrada, que es el link con  $S_1$ ). Nadie contestará este mensaje enviado, puesto que ningún host conectado a  $S_2$  es  $H_2$ .

Además, se visualiza que:  $S_1$  instala en su tabla de flujos la entrada  $H_1 \rightarrow H_2$  y  $H_2 \rightarrow H_1$ . Luego de un tiempo, cuando arriba un paquete, vuelve a consultar al controlador puesto que estas entradas tienen un tiempo de vida pequeño y este se venció.

### Mensajes entre hosts filtrados

Como hosts a filtrar (que no deban comunicarse de ninguna forma) elegimos los hosts  $H_1$  y  $H_4$ .

En este apartado, mostramos una prueba en la terminal de qué ocurre cuando se filtran dichos hosts.

En la siguiente imagen, se evidencia en mininet que se están filtrando ambos hosts (por lo tanto, no puede llegar el ping).

```
mininet> h1 ping h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
```

Figura 31: Filtrado entre dos hosts: Mininet.

A continuación, mostramos qué ocurre con la salida de los logs del controlador.

```
DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.1, IP dst 10.0.0.4, protocol=ARP
DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.4, IP dst 10.0.0.1, protocol=ARP
DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.1, IP dst 10.0.0.4, protocol=IP
DEBUG:misc.firewall:Rule sent: Dropping: IP src 10.0.0.4, IP dst 10.0.0.1, protocol=IP
DEBUG:misc.firewall:Rule sent: Dropping: MAC src 00:00:00:00:00:04, MAC dst 00:00:00:00:00:01
DEBUG:misc.firewall:Rule sent: Dropping: MAC src 00:00:00:00:00:01, MAC dst 00:00:00:00:00:04
DEBUG:misc.firewall:Rule sent: Dropping: UDP port 5001 for messages sent by host 10.0.0.1
DEBUG:misc.firewall:Rule sent: Dropping: PORT dst 80, protocol=UDP
DEBUG:misc.firewall:Rule sent: Dropping: PORT dst 80, protocol=TCP
DEBUG:misc.firewall:All rules sent.

DEBUG:openflow.of_01:1 connection aborted
```

Figura 32: Filtrado entre dos hosts: Logs del controlador.

Se aprecia que, luego de instalar las reglas, no se genera ningún mensaje (ni del tipo ARP), y luego se aborta la conexión cuando finaliza mininet. Por ende, el filtrado resultó como esperábamos.

Para filtrar totalmente ambos hosts, decidimos que  $H_1$  jamás puede saber cuál es la MAC de  $H_4$  y viceversa. Por ende, debemos filtrar los paquetes ARP.

En la siguiente imagen de Wireshark, capturando desde la interfaz que conecta al host  $H_1$  y al switch  $S_1$ , se evidencia que el host realiza un broadcast para encontrar la MAC de  $H_4$ , pero, como está solamente conectado a  $S_1$ , queda completamente descartado el mensaje. Es decir, nunca le llega ninguna respuesta de cuál es la MAC de  $H_4$ .

Source	Destination	Protocol	Length	Info
00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1
00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.4? Tell 10.0.0.1

Figura 33: Filtrado de paquetes: Captura de Wireshark.

### Realizando un pingall en mininet

En este apartado, mostraremos capturas de logs del controlador y de Wireshark, evidenciando el funcionamiento del pingall (utilizando las reglas de filtrado).

En la siguiente imagen, se muestra el resultado corriendo la terminal de mininet el pingall, con el controlador de POX levantado.

```
*** Post configure switches and hosts
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 X
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> X h2 h3
*** Results: 16% dropped (10/12 received)
mininet> exit
```

Se evidencian las reglas de filtrado entre los hosts  $H_1$  y  $H_4$ , puesto que no llega una respuesta para ninguno de los dos.

En las siguientes imagenes, se mostrarán capturas de los logs del controlador de POX.

```
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 1: 00:00:00:00:00:01 --> ff:ff:ff:ff:ff:ff
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 2: 00:00:00:00:00:02 --> 00:00:00:00:00:01
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:02.2 -> 00:00:00:00:00:01.1
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-02 interface 3: 00:00:00:00:00:01 --> ff:ff:ff:ff:ff:ff
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 1: 00:00:00:00:00:01 --> 00:00:00:00:00:02
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:01.1 -> 00:00:00:00:00:02.2
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 2: 00:00:00:00:00:02 --> 00:00:00:00:00:01
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:02.2 -> 00:00:00:00:00:01.1
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 1: 00:00:00:00:00:01 --> ff:ff:ff:ff:ff:ff
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-02 interface 3: 00:00:00:00:00:01 --> ff:ff:ff:ff:ff:ff
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-02 interface 1: 00:00:00:00:00:03 --> 00:00:00:00:00:01
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:03.1 -> 00:00:00:00:00:01.3
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 3: 00:00:00:00:00:03 --> 00:00:00:00:00:01
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:03.3 -> 00:00:00:00:00:01.1
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 1: 00:00:00:00:00:01 --> 00:00:00:00:00:03
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:01.1 -> 00:00:00:00:00:03.3
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-02 interface 3: 00:00:00:00:00:01 --> 00:00:00:00:00:03
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:01.3 -> 00:00:00:00:00:03.1
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-02 interface 1: 00:00:00:00:00:03 --> 00:00:00:00:00:01
DEBUG:forwarding.l2_learning:installing flow for 00:00:00:00:00:03.1 -> 00:00:00:00:00:01.3
DEBUG:misc.firewall:Packet arrived from switch: 00-00-00-00-00-01 interface 3: 00:00:00:00:00:03 --> 00:00:00:00:00:01
```









00:00:00_00:00:01	Broadcast	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:01	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT
00:00:00_00:00:02	00:00:00_00:00:01	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:01	Broadcast	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	00:00:00_00:00:01	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:01	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT
00:00:00_00:00:01	Broadcast	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:01	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT
00:00:00_00:00:01	Broadcast	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:01	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	00:00:00_00:00:01	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:03	00:00:00_00:00:01	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	00:00:00_00:00:01	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:03	00:00:00_00:00:01	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:02	Broadcast	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT
00:00:00_00:00:02	Broadcast	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	00:00:00_00:00:02	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:03	00:00:00_00:00:02	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	00:00:00_00:00:02	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:03	00:00:00_00:00:02	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:02	Broadcast	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT
00:00:00_00:00:02	Broadcast	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT
00:00:00_00:00:04	00:00:00_00:00:02	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:04	00:00:00_00:00:02	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:04	00:00:00_00:00:02	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:04	00:00:00_00:00:02	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	Broadcast	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:03	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT

00:00:00_00:00:04	00:00:00_00:00:03	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:03	Broadcast	OpenFl...	132 Type: OFPT_PACKET_OUT
00:00:00_00:00:04	00:00:00_00:00:03	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	00:00:00_00:00:01	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	00:00:00_00:00:01	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	00:00:00_00:00:01	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:01	00:00:00_00:00:02	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:03	00:00:00_00:00:01	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	00:00:00_00:00:01	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:01	00:00:00_00:00:02	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	00:00:00_00:00:01	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:01	00:00:00_00:00:03	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:01	00:00:00_00:00:03	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:01	00:00:00_00:00:03	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:01	00:00:00_00:00:03	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:04	00:00:00_00:00:03	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:04	00:00:00_00:00:03	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	00:00:00_00:00:04	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:03	00:00:00_00:00:02	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:04	00:00:00_00:00:02	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:03	00:00:00_00:00:04	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	00:00:00_00:00:02	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:03	00:00:00_00:00:02	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:04	00:00:00_00:00:02	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:04	00:00:00_00:00:02	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:03	00:00:00_00:00:02	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:02	00:00:00_00:00:03	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:04	00:00:00_00:00:02	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:02	00:00:00_00:00:04	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	00:00:00_00:00:03	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:02	00:00:00_00:00:03	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	00:00:00_00:00:04	OpenFl...	220 Type: OFPT_PACKET_OUT

00:00:00_00:00:02	00:00:00_00:00:04	OpenFl...	126 Type: OFPT_PACKET_IN
00:00:00_00:00:02	00:00:00_00:00:03	OpenFl...	220 Type: OFPT_PACKET_OUT
00:00:00_00:00:02	00:00:00_00:00:04	OpenFl...	220 Type: OFPT_PACKET_OUT

Luego de mostrar todas las capturas, se puede evidenciar el correcto funcionamiento del pingall (filtrando ciertos pings entre hosts, definidos en las reglas). Las capturas de wireshark demuestran que se contrastan los logs con los mensajes que se están capturando, sin que se produzca ningún error.

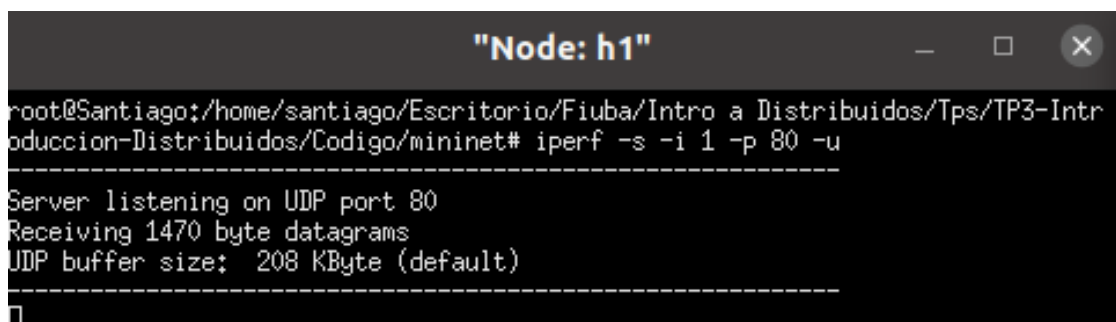
### 4.3. Iperf

Al levantar la network con mininet podemos ejecutar el comando de 'xterm <host>' el cual nos habilita una terminal virtual para dicho host. Desde esta terminal ejecutamos la herramienta iperf, simulando diferentes escenarios, para poder probar el correcto funcionamiento de las reglas del firewall.

#### Caso 1: Se descartan mensajes con puerto destino 80

Para validar este caso levantamos dos servidores UDP, uno ejecutandose en el host H1 en el puerto 80 y otro en el host H2 en el puerto 4000. Desde el host H3 enviaremos mensajes UDP a ambos servidores y el mensaje debería llegar solo a H2 y no a H1.

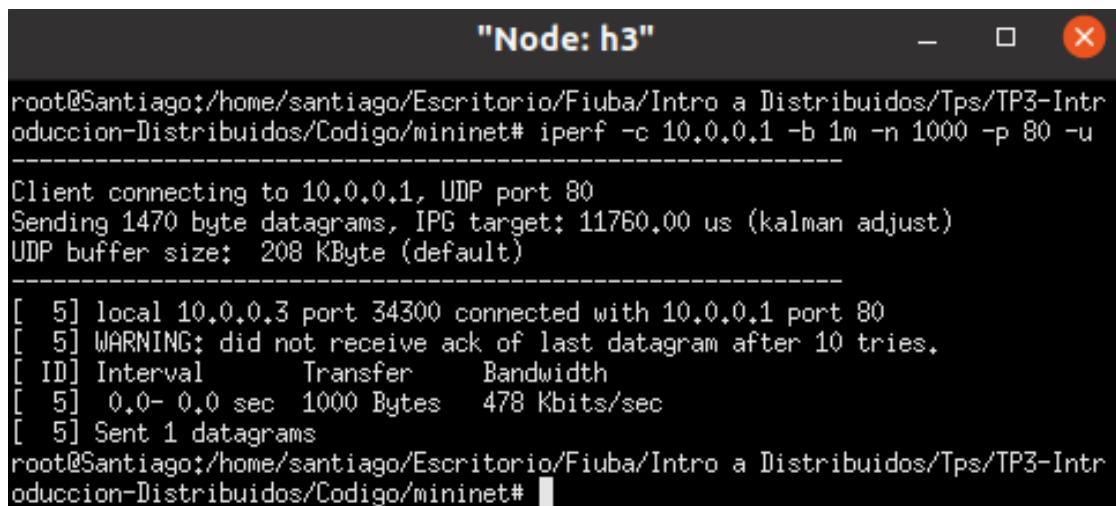
**Host 1:** Iniciamos un servidor UDP en el puerto 80



```
"Node: h1"
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Intro
duccion-Distribuidos/Codigo/mininet# iperf -s -i 1 -p 80 -u
-----
Server listening on UDP port 80
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
█
```

Figura 34: Servidor UDP corriendo en 10.0.0.1:80

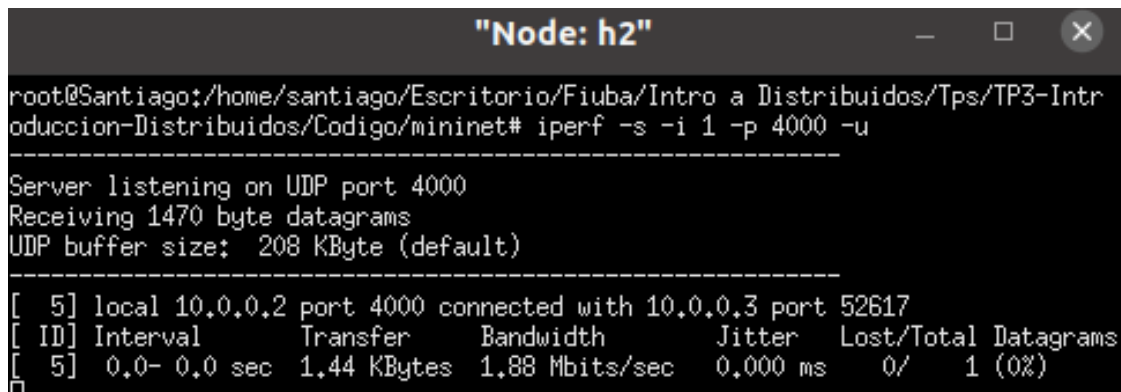
**Host 3:** Enviamos mensaje UDP al server corriendo en 10.0.0.1:80 y vemos como un warning indica que luego de 10 intentos no se recibió un ack. De igual forma, en la terminal de H1 no vemos que el mensaje de H3 llegue.



```
"Node: h3"
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Intro
duccion-Distribuidos/Codigo/mininet# iperf -c 10.0.0.1 -b 1m -n 1000 -p 80 -u
-----
Client connecting to 10.0.0.1, UDP port 80
Sending 1470 byte datagrams, IPG target: 11760.00 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.3 port 34300 connected with 10.0.0.1 port 80
[ 5] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0- 0.0 sec    1000 Bytes  478 Kbits/sec
[ 5] Sent 1 datagrams
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Intro
duccion-Distribuidos/Codigo/mininet# █
```

Figura 35: Cliente UDP en H3 intenta enviar mensaje a 10.0.0.1:80

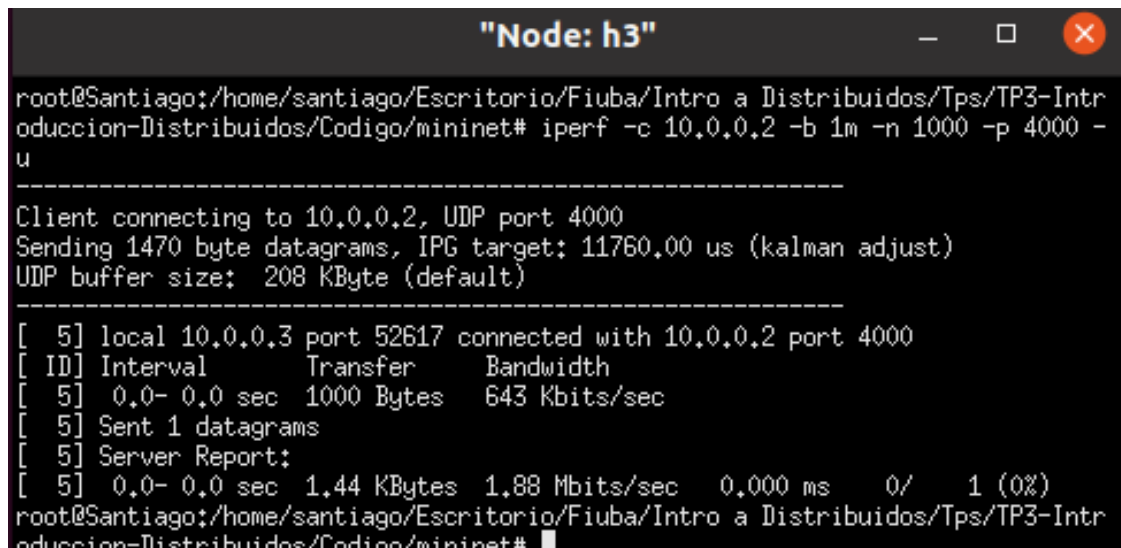
**Host 2:** Realizamos el mismo procedimiento que para el host 1 con la diferencia de que el servidor correrá en el puerto 4000. Como vemos, la captura fue tomada luego de que el cliente en h3 envíe el mensaje y el servidor en H2 efectivamente lo recibe ya que no hay ninguna restricción que lo impida.



```
"Node: h2"
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -s -i 1 -p 4000 -u
-----
Server listening on UDP port 4000
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.2 port 4000 connected with 10.0.0.3 port 52617
[ ID] Interval      Transfer    Bandwidth    Jitter    Lost/Total Datagrams
[ 5] 0.0- 0.0 sec  1.44 KBytes 1.88 Mbits/sec 0.000 ms   0/ 1 (0%)
```

Figura 36: Servidor UDP corriendo en 10.0.0.2:4000

**Host 3:** Enviamos mensaje UDP al server corriendo en 10.0.0.2:4000 y vemos como el mensaje es enviado y recibido en el servidor con exito.



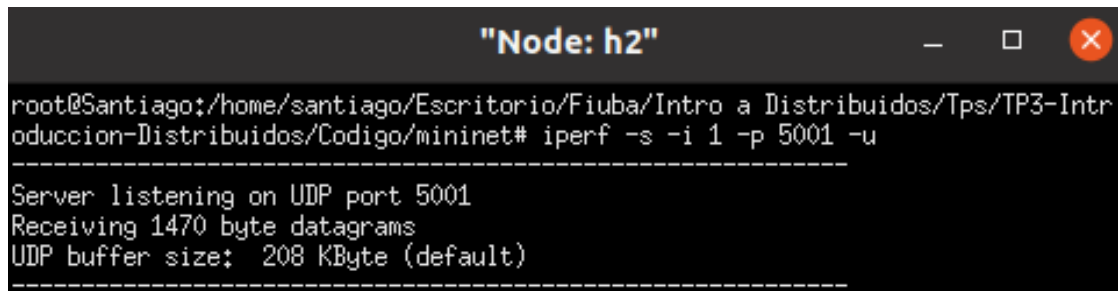
```
"Node: h3"
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -c 10.0.0.2 -b 1m -n 1000 -p 4000 -u
-----
Client connecting to 10.0.0.2, UDP port 4000
Sending 1470 byte datagrams, IPG target: 11760.00 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.3 port 52617 connected with 10.0.0.2 port 4000
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0- 0.0 sec  1000 Bytes  643 Kbits/sec
[ 5] Sent 1 datagrams
[ 5] Server Report:
[ 5] 0.0- 0.0 sec  1.44 KBytes 1.88 Mbits/sec 0.000 ms   0/ 1 (0%)
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Introduccion-Distribuidos/Codigo/mininet#
```

Figura 37: Cliente UDP en H3 enviar mensaje a 10.0.0.2:4000

**Caso 2:** Se descartan mensajes que provienen del host 1, con puerto destino 5001 y utilicen UDP

Para validar este caso levantamos un servidor UDP ejecutandose en el host H2 con puerto 5001 (10.0.0.2:5001). Intentaremos enviar mensajes conectándonos desde el host 1 y desde el host 3, solo deben llegar los mensajes que envia el host 3.

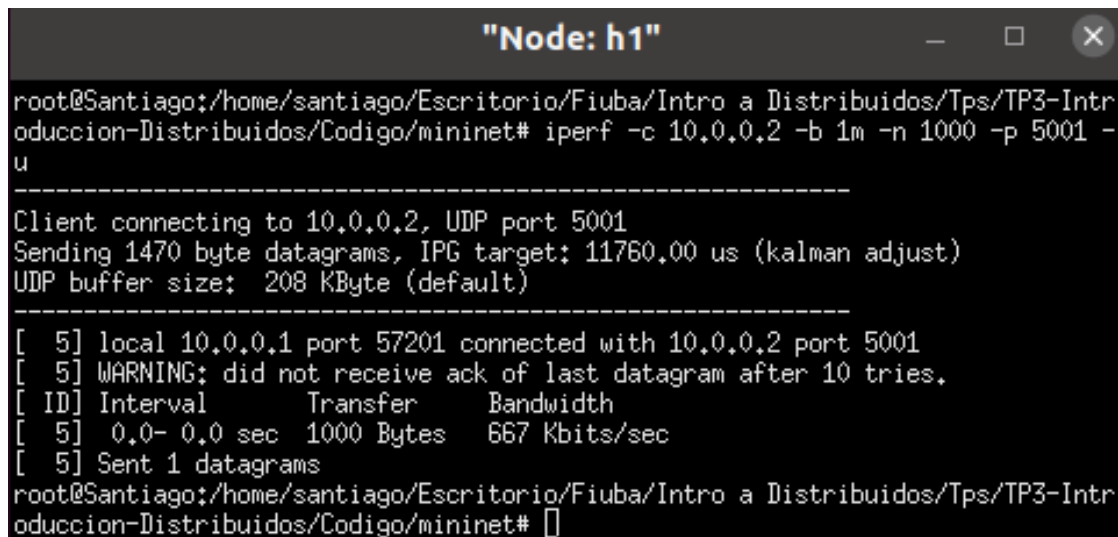
**Host 2:** Iniciamos un servidor UDP en el puerto 5001



```
"Node: h2"
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -s -i 1 -p 5001 -u
-----
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
```

Figura 38: Servidor UDP corriendo en 10.0.0.2:5001

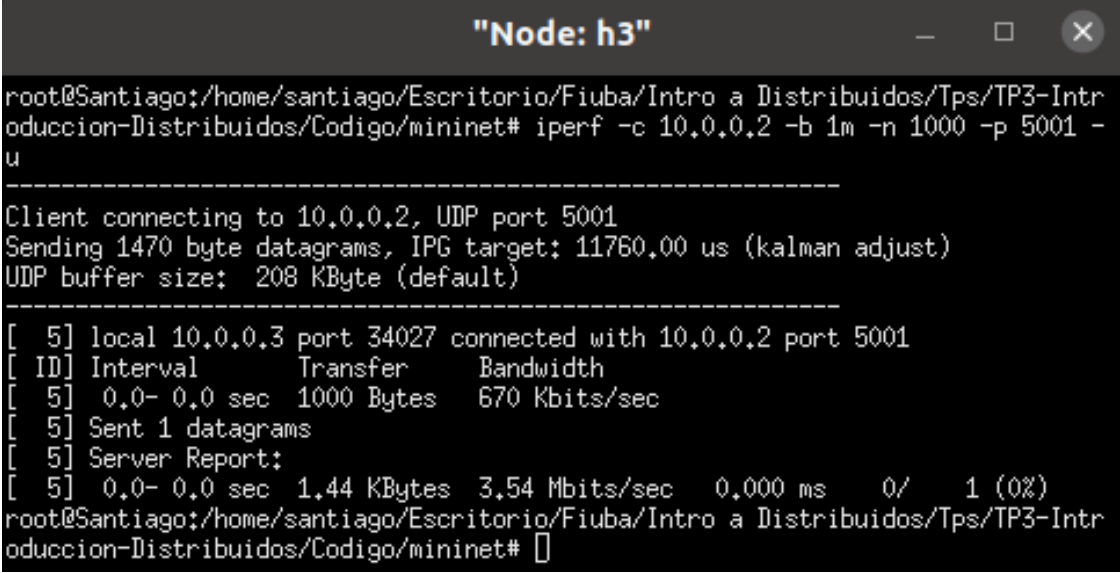
**Host 1:** Enviamos mensaje UDP al server corriendo en 10.0.0.2:5001 y vemos como un warning indica que luego de 10 intentos no se recibió un ack. De igual forma, en la terminal de H2 veremos que el mensaje de H1 no llegó.



```
"Node: h1"
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -c 10.0.0.2 -b 1m -n 1000 -p 5001 -u
-----
Client connecting to 10.0.0.2, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11760.00 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.1 port 57201 connected with 10.0.0.2 port 5001
[ 5] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0- 0.0 sec  1000 Bytes  667 Kbits/sec
[ 5] Sent 1 datagrams
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Introduccion-Distribuidos/Codigo/mininet#
```

Figura 39: Cliente UDP en H1 intenta enviar mensaje a 10.0.0.2:5001

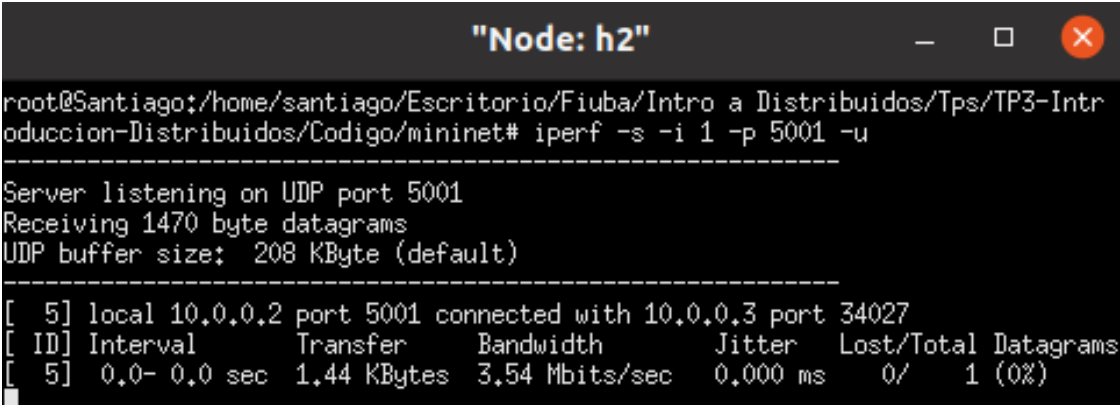
**Host 3:** Enviamos mensaje UDP al server corriendo en 10.0.0.2:5001 y vemos como este se envía de forma correcta a diferencia del caso recién visto en H1.



```
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -c 10.0.0.2 -b 1m -n 1000 -p 5001 -u
-----
Client connecting to 10.0.0.2, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11760.00 us (kalman adjust)
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.3 port 34027 connected with 10.0.0.2 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0- 0.0 sec  1000 Bytes  670 Kbits/sec
[ 5] Sent 1 datagrams
[ 5] Server Report:
[ 5] 0.0- 0.0 sec  1.44 KBytes  3.54 Mbits/sec  0.000 ms  0/ 1 (0%)
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Introduccion-Distribuidos/Codigo/mininet#
```

Figura 40: Cliente UDP en H3 envía mensaje a 10.0.0.2:5001

**Host 2:** Por último podemos ver el estado final del servidor ejecutándose en el host 2 donde podemos verificar que el mensaje enviado desde H3 llegó de forma correcta mientras que el mensaje enviado desde el host 1 no lo hizo.

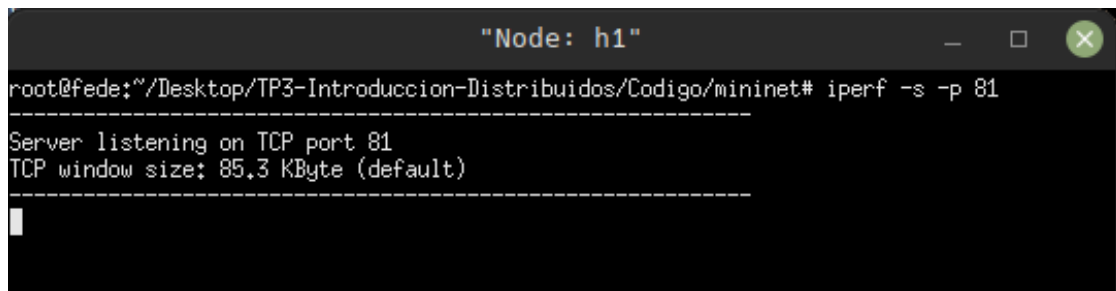


```
root@Santiago:/home/santiago/Escritorio/Fiuba/Intro a Distribuidos/Tps/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -s -i 1 -p 5001 -u
-----
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size: 208 KByte (default)
-----
[ 5] local 10.0.0.2 port 5001 connected with 10.0.0.3 port 34027
[ ID] Interval      Transfer    Bandwidth    Jitter  Lost/Total Datagrams
[ 5] 0.0- 0.0 sec  1.44 KBytes  3.54 Mbits/sec  0.000 ms  0/ 1 (0%)
```

Figura 41: Estado final del servidor UDP ejecutado en 10.0.0.2:5001

### Caso 3: Dos host no pueden comunicarse de ninguna forma (Host 1 y Host 4)

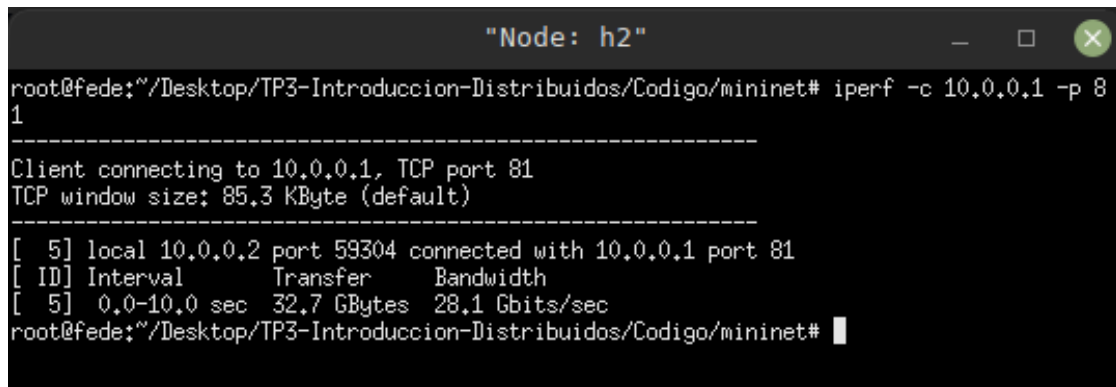
Dado que el servidor está levantado en el host 1 y sobre un puerto que no tiene restricciones, la única conexión que debería fallar es la del host H4 ya que hay una regla en el firewall que prohíbe toda conexión entre el host H1 y H4.



```
"Node: h1"
root@fede:~/Desktop/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -s -p 81
-----
Server listening on TCP port 81
TCP window size: 85.3 KByte (default)
-----
```

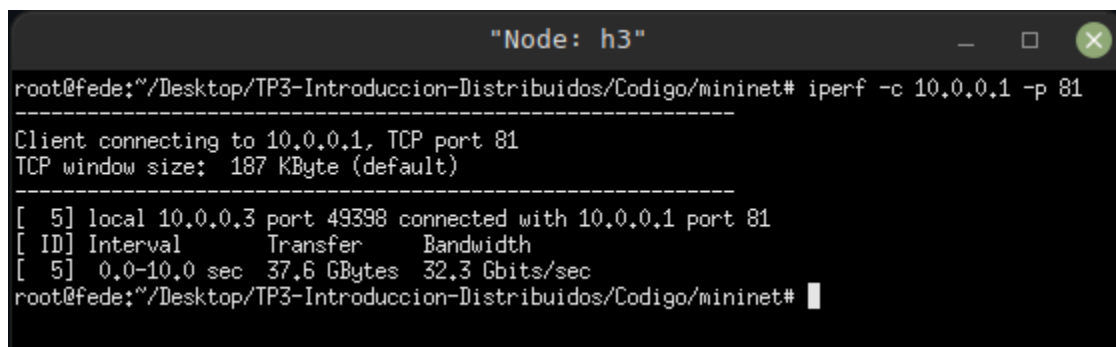
Figura 42: Servidor TCP - Host 1 - 10.0.0.1 - Puerto 81

**Host 2 y Host 3:** En las siguientes capturas, se ingresó a las terminales de los hosts 2 y 3 (con IP's 10.0.0.2 y 10.0.0.3 respectivamente) y se envió tráfico al servidor en el host 1 previamente levantado.



```
"Node: h2"
root@fede:~/Desktop/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -c 10.0.0.1 -p 81
-----
Client connecting to 10.0.0.1, TCP port 81
TCP window size: 85.3 KByte (default)
-----
[ 5] local 10.0.0.2 port 59304 connected with 10.0.0.1 port 81
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0-10.0 sec  32.7 GBytes 28.1 Gbits/sec
root@fede:~/Desktop/TP3-Introduccion-Distribuidos/Codigo/mininet#
```

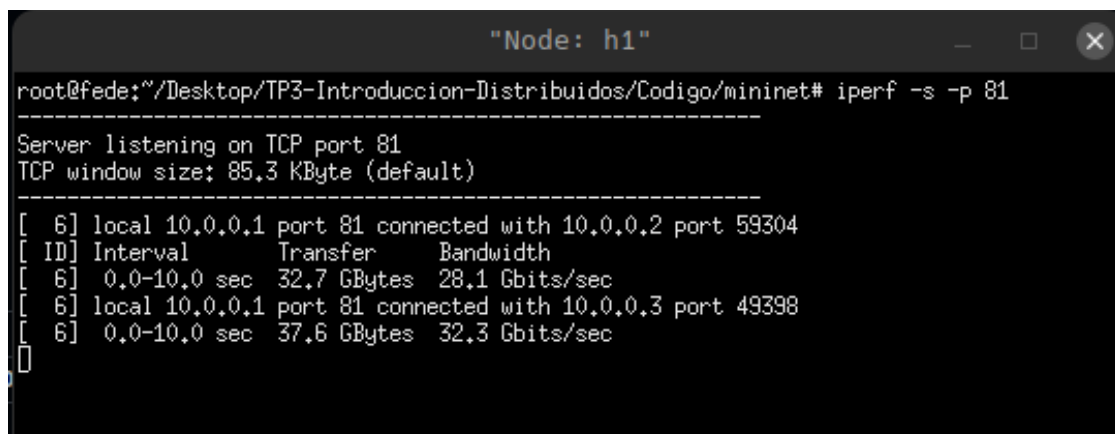
Figura 43: Cliente Host 2 - 10.0.0.2 enviado trafico a Host 1



```
"Node: h3"
root@fede:~/Desktop/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -c 10.0.0.1 -p 81
-----
Client connecting to 10.0.0.1, TCP port 81
TCP window size: 187 KByte (default)
-----
[ 5] local 10.0.0.3 port 49398 connected with 10.0.0.1 port 81
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0-10.0 sec  37.6 GBytes 32.3 Gbits/sec
root@fede:~/Desktop/TP3-Introduccion-Distribuidos/Codigo/mininet#
```

Figura 44: Cliente Host 3 - 10.0.0.3 enviado trafico a Host 1

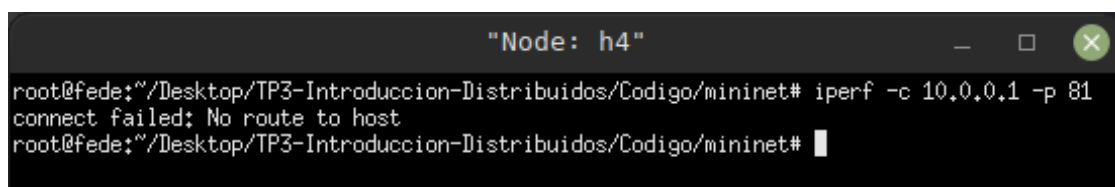
Como no hay ninguna regla que actúe frente a estas direcciones IPs (ni al puerto ni al protocolo), el tráfico de los hosts se manda completamente al host 1 servidor. Esto también se valida en la terminal del servidor:



```
"Node: h1"
root@fede:~/Desktop/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -s -p 81
-----
Server listening on TCP port 81
TCP window size: 85.3 KByte (default)
-----
[ 6] local 10.0.0.1 port 81 connected with 10.0.0.2 port 59304
[ ID] Interval      Transfer    Bandwidth
[ 6]  0.0-10.0 sec  32.7 GBytes 28.1 Gbits/sec
[ 6] local 10.0.0.1 port 81 connected with 10.0.0.3 port 49398
[ 6]  0.0-10.0 sec  37.6 GBytes 32.3 Gbits/sec
[]
```

Figura 45: Servidor Host 1 recibiendo trafico

**Host 4:** Se puede validar la regla en la que no se pueden comunicar dos host cualquiera (en este caso, host 1 y host 4). Al intentar enviar tráfico desde el cliente corriendo en el host 4 al servidor corriendo en el host 1 se puede observar que no es posible establecer la conexión entre los mismos.



```
"Node: h4"
root@fede:~/Desktop/TP3-Introduccion-Distribuidos/Codigo/mininet# iperf -c 10.0.0.1 -p 81
connect failed: No route to host
root@fede:~/Desktop/TP3-Introduccion-Distribuidos/Codigo/mininet#
```

Figura 46: Cliente Host 4 (10.0.0.4) imposible conectarse con Server en Host 1

## 5. Conclusión

El presente trabajo nos sirvió para:

- **Entender cómo es la interacción entre los Switches OpenFlow y su controlador.** Fue importante poder entender la interacción en cada componente entre el data-plane y el controlador que implementamos (además del controlador que implementa el L2 learning), para comprobar el correcto funcionamiento de la red.
- **Introducción a nuevas herramientas:** Iperf, Spear by Namox, xterm.
- **Profundizar la utilización wireshark:** Fundamentalmente, hicimos capturas desde distintas interfaces, capturando las interfaces de los Switches para entender qué mensajes se estaban mandando, y capturando en localhost (filtrando por el puerto 6633) para comprender los mensajes entre el controlador y cada Switch.

De esta manera, tenemos un mejor panorama sobre la **utilización de SDN en una red real**, entendiendo: La interacción entre el controlador y cada Switch dentro de una red.

## 6. Referencias

- [1] BD. KREUTZ, F. M. V. RAMOS, P. E. VERÍSSIMO, C. E. ROTHENBERG, S. AZODOLMOLKY y S. UHLIG, "*Software-Defined Networking: A Comprehensive Survey*", in Proceedings of the IEEE, vol. 103, no. 1, pp. 14-76, Jan. 2015, doi: 10.1109/JPROC.2014.2371999.

- [2] Documentación oficial de POX: <https://noxrepo.github.io/pox-doc/html/>.
- [3] Documentación oficial de Mininet: <http://mininet.org/walkthrough/>.
- [4] Documentación oficial de Iperf: <https://iperf.fr/iperf-doc.php>.
- [5] Computer Networking : A Top-Down Approach with Access. Kurose. Ross.