Trabajo Práctico 2
Software-Defined Networks - Grupo 9

[75.43] Introducción a los Sistemas Distribuidos
Segundo cuatrimestre de 2025

ALUMNO	PADRON	CORREO
BARTOCCI, Camila	105781	cbartocci@fi.uba.ar
PATIÑO, Franco	105126	fpatino@fi.uba.ar
RETAMOZO, Melina	110065	mretamozo@fi.uba.ar
SAGASTUME, Matias	110530	csagastume@fi.uba.ar
SENDRA, Alejo	107716	asendra@fi.uba.ar

Índice

1. Introducción

El objetivo de este trabajo práctico es profundizar en el estudio de las Redes Definidas por Software (SDN) y el funcionamiento del protocolo OpenFlow.

El desarrollo se centra en la construcción de una arquitectura flexible capaz de soportar una topología de red dinámica, y de un módulo de firewall de capa de enlace para el filtrado de tráfico.

Para la realización de estas tareas se emplea Mininet como entorno de emulación de la infraestructura de red, junto con el controlador POX, el cual opera como el cerebro centralizado responsable de la lógica de reenvío y la aplicación de las políticas de seguridad.

2. Herramientas utilizadas

- **Mininet**: Emulación de la topología de red
- **POX**: Controlador SDN con L2 learning y firewall
- **iperf**: Generación y medición de tráfico TCP/UDP
- **Wireshark**: Captura y análisis de paquetes
- **ping**: Verificación de conectividad ICMP
- **curl/http.server**: Pruebas de tráfico HTTP

3. Implementación

La implementación del proyecto se dividió en tres componentes principales: la topología de red, el controlador SDN y las reglas del firewall. A continuación se detallan los aspectos técnicos de cada uno.

3.1. Arquitectura General

El sistema implementado sigue el paradigma de Software-Defined Networking (SDN), separando el plano de control del plano de datos. La arquitectura consta de:

- **Plano de Datos**: Switches OpenFlow emulados en Mininet, responsables del forwarding de paquetes según las reglas instaladas.

- **Plano de Control**: Controlador POX que implementa la lógica de L2 learning y firewall.
- **Canal de Comunicación**: Protocolo OpenFlow sobre TCP (puerto 6633).

3.2. Conexión Switch-Controlador

La comunicación entre los switches y el controlador se establece mediante el protocolo OpenFlow:

1. El controlador POX inicia un servidor TCP en el puerto 6633 (puerto estándar de OpenFlow).
2. Al crear la topología, Mininet configura cada switch con la dirección del controlador (`127.0.0.1:6633`).
3. Cada switch establece automáticamente una conexión TCP al controlador.
4. Una vez conectado, el controlador recibe un evento `ConnectionUp`, momento en el cual los módulos registrados (L2 learning y firewall) pueden instalar sus reglas en el switch.

Este canal de control se mantiene activo durante toda la simulación.

3.3. Topología Parametrizable

3.3.1. Diseño

Se implementó una topología de cadena (*chain topology*) parametrizable, donde el número de switches (N) puede ser definido por el usuario. La topología cumple con las siguientes características:

- **Switches**: N switches conectados linealmente formando una cadena: $S_1 - S_2 - ... - S_N$
- **Hosts**: 4 hosts totales distribuidos en los extremos
  - h_1 y h_2 conectados al switch S_1 (primer extremo)
  - h_3 y h_4 conectados al switch S_N (último extremo)
- **Direcciones IP**: Asignación secuencial en la red 10.0.0.0/24

- h_1: 10.0.0.1
- h_2: 10.0.0.2
- h_3: 10.0.0.3
- h_4: 10.0.0.4

3.3.2. Caso Especial: $N = 1$

Cuando $N = 1$, todos los hosts se conectan al único switch disponible, formando una topología de estrella simple.

3.3.3. Implementación Técnica

La topología se implementó en Python usando la API de Mininet, definiendo una clase `ChainTopology` que hereda de `Topo`.

La clase maneja dinámicamente la creación de switches y sus enlaces.

3.3.4. Topologías de Ejemplo

A continuación se muestran diagramas de las topologías para diferentes valores de N:

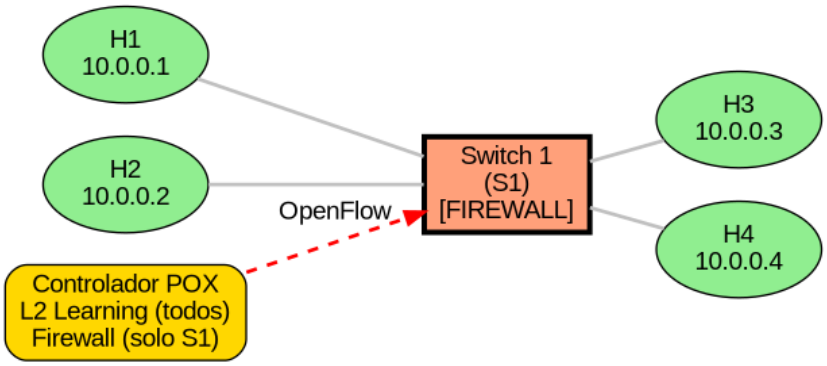

Figura 1: Topología con N=1 switches

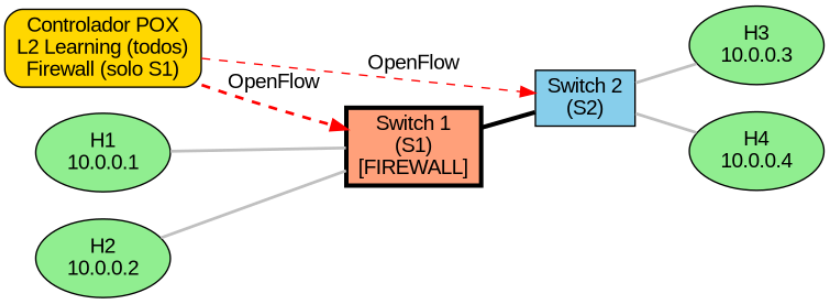

Figura 2: Topología con N=2 switches

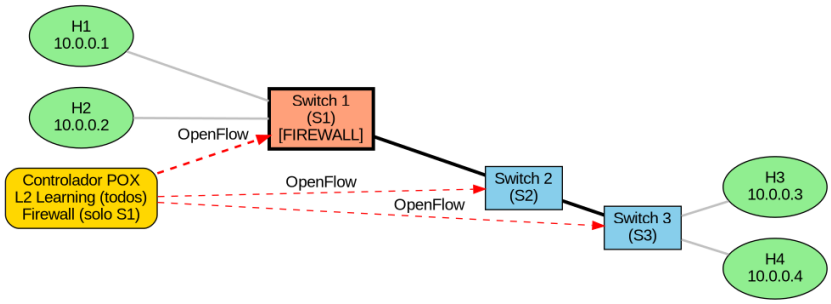

Figura 3: Topología con N=3 switches

3.4. Controlador SDN

El controlador se implementó usando POX, un framework de controladores SDN en Python. La implementación consta de dos módulos principales:

3.4.1. L2 Learning

Se utiliza el módulo `forwarding.l2_learning` de POX, que implementa el algoritmo de aprendizaje de direcciones MAC:

1. Al recibir un *PacketIn*, el switch no sabe cómo reenviar el paquete.
2. El controlador aprende la asociación MAC origen $\leftrightarrow$ puerto.
3. Si conoce la MAC destino, instala una regla de flujo.
4. Si no la conoce, realiza *flooding* en todos los puertos (excepto el de entrada).

Este mecanismo permite que los switches aprendan dinámicamente la topología sin configuración manual.

3.4.2. Firewall

El firewall se implementó como un módulo POX custom en `controller/firewall.py`. Sus características principales son:

Ubicación del Firewall (Parametrizable):
El firewall permite configurar en qué switches se aplican las reglas mediante el campo `switch` en cada regla del archivo JSON. Por defecto, las reglas se aplican en el switch 1 (primer switch de la cadena). Esta decisión de diseño tiene las siguientes implicaciones:

- **Ventaja**: Centraliza el control de seguridad en un único punto, simplificando la gestión y monitoreo.
- **Limitación**: El tráfico que no pasa por S1 (ej: comunicación directa entre h3 y h4 en topologías N>1) no es filtrado si las reglas solo están configuradas para S1.
- **Cobertura**: En la topología implementada (N=2), todo el tráfico inter-extremos (h1/h2 $\leftrightarrow$ h3/h4) pasa por S1, garantizando que las reglas críticas se apliquen correctamente.
- **Flexibilidad**: Se pueden configurar reglas específicas para cada switch según las necesidades de segmentación de la red.

La implementación permite especificar el campo `"switch": N` en cada regla del archivo JSON. Al conectarse un switch, el controlador filtra las reglas correspondientes a su DPID y solo instala aquellas que le corresponden. Si no se especifica el campo, la regla se aplica al switch 1 por defecto.

Instalación Proactiva de Reglas:

- Las reglas se instalan al momento de la conexión del switch (evento `ConnectionUp`).
- Se utiliza el mensaje `ofp_flow_mod` de OpenFlow para crear entradas en la tabla de flujos.

Estructura de Reglas:
Cada regla define un `ofp_match` con los siguientes campos posibles:

- `dl_type`: Tipo de Ethernet (0x0800 para IPv4)
- `nw_src` / `nw_dst`: Direcciones IP origen/destino

- `nw_proto`: Protocolo (TCP=6, UDP=17, ICMP=1)
- `tp_src` / `tp_dst`: Puertos TCP/UDP origen/destino

Las reglas sin acciones asociadas resultan en *DROP* implícito.

Validación de Reglas:
Se implementó un sistema robusto de validación en `controller/utils.py`:

- Validación de formato de direcciones IPv4
- Verificación de protocolos válidos (TCP, UDP, ICMP)
- Validación de rangos de puertos (1-65535)
- Validación del número de switch (entero ≥ 1)
- Verificación de prerequisitos de OpenFlow (ej: puerto requiere protocolo)
- Detección de errores comunes (ej: ICMP no puede tener puertos TCP/UDP)
- Asignación automática de switch por defecto (1) si no se especifica
- Logging detallado de reglas inválidas

Las reglas inválidas son ignoradas automáticamente, permitiendo que el sistema continúe funcionando.

3.5. Reglas del Firewall Implementadas

Las reglas se definen en el archivo `controller/firewall_rules.json`. Se implementaron las siguientes reglas según el enunciado:

3.5.1. Regla 1: Bloqueo de Puerto 80

Descripción: Bloquear todos los mensajes cuyo puerto destino sea 80.
Implementación: Se crearon dos reglas separadas, una para TCP y otra para UDP:

```
{
  "description": "Bloquear todo el tráfico al puerto 80",
  "protocol": "TCP",
  "dst_port": 80,
  "switch": 1
}
```

Esta regla previene el tráfico TCP con destino al puerto 80 en el switch 1.

3.5.2. Regla 2: Bloqueo Específico UDP

Descripción: Bloquear mensajes desde host 1 (10.0.0.1) con puerto destino 5001 y protocolo UDP.

```
{
  "description": "Bloquear UDP desde host 1 con puerto destino 5001",
  "src_ip": "10.0.0.1",
  "protocol": "UDP",
  "dst_port": 5001,
  "switch": 1
}
```

Esta regla es más específica, bloqueando solo el tráfico UDP desde un host particular a un puerto específico en el switch 1.

3.5.3. Regla 3: Bloqueo Bidireccional

Descripción: Impedir comunicación en ambas direcciones entre host 2 (10.0.0.2) y host 3 (10.0.0.3).

Implementación: Se requieren dos reglas para bloquear ambas direcciones:

```
{
  "description": "Bloquear comunicación de host 2 a host 3",
  "src_ip": "10.0.0.2",
  "dst_ip": "10.0.0.3",
  "switch": 1
},
{
  "description": "Bloquear comunicación de host 3 a host 2",
  "src_ip": "10.0.0.3",
  "dst_ip": "10.0.0.2",
  "switch": 1
}
```

Esto asegura el bloqueo total de comunicación entre ambos hosts en el switch 1, sin importar quién inicie la conexión.

3.6. Estructura del Proyecto

El código fuente se organizó de la siguiente manera:

```
proyecto/
|-- controller/              # Módulos SDN custom
|   |-- __init__.py
|   |-- firewall.py         # Implementación del firewall
|   |-- utils.py            # Validación de reglas
|   `-- firewall_rules.json # Reglas del firewall
|-- pox/                    # POX (instalado externamente)
|-- topology.py             # Topología Mininet
|-- run_controller.sh       # Script para ejecutar POX
|-- run_topology.sh         # Script para ejecutar Mininet
|-- install_pox.sh          # Script de instalación
`-- README.md               # Documentación
```

3.7. Decisiones de Diseño

3.7.1. Uso de L2 Learning Existente

Se decidió utilizar el módulo `forwarding.l2_learning` de POX en lugar de implementar uno desde cero. Esta decisión se basó en:

- Cumplimiento de requisitos (el enunciado permite usar módulos de POX)
- Robustez y optimización del módulo oficial
- Enfoque en la implementación del firewall (objetivo principal)
- Reducción de complejidad y tiempo de desarrollo

3.7.2. Validación Extensiva

Se implementó un sistema de validación completo para prevenir errores comunes:

- Evita reglas que generen warnings de OpenFlow
- Proporciona retroalimentación clara sobre errores
- Permite que el sistema continúe funcionando con reglas válidas
- Facilita el debugging y la depuración

3.8. Scripts de Automatización

Se desarrollaron scripts bash para facilitar la ejecución:

- `install_pox.sh`: Clona e instala POX desde GitHub
- `run_controller.sh`: Ejecuta POX con los módulos necesarios
- `run_topology.sh`: Lanza Mininet con la topología parametrizable

Estos scripts incluyen validaciones y mensajes de error informativos.

3.9. Compatibilidad y Modificaciones Técnicas

3.9.1. Fix de Compatibilidad Python 3

Durante la instalación de POX, se aplica un parche al archivo `pox/lib/packet/dns.py` para garantizar compatibilidad con Python 3.x. El archivo original de POX contiene código heredado de Python 2 que utiliza la función `ord()` innecesariamente al indexar bytes.

Modificaciones realizadas:

- Eliminación de llamadas `ord()` redundantes en el parsing de nombres DNS (líneas 378, 382)
- Corrección del método `join()` para trabajar con bytes en lugar de strings (línea 399)

Este fix se aplica automáticamente en el script `install_pox.sh`, copiando el archivo `dns.py` corregido al directorio correspondiente de POX después de su clonación desde GitHub.

4. Pruebas y Validación

Para verificar que el sistema funciona como esperamos, realizamos una serie de pruebas que cubren desde conectividad básica hasta validación específica de cada regla del firewall. También probamos diferentes configuraciones de topología para asegurar que el diseño escala correctamente.

4.1. Configuración del Entorno

El procedimiento para todas las pruebas es el mismo:

1. Arrancar el controlador POX (que carga L2 learning y el firewall):

   ```
   ./run_controller.sh
   ```

2. Levantar la topología de Mininet con el número de switches deseado:

   ```
   ./run_topology.sh 2
   ```

3. Esperar unos segundos a que los switches se conecten y las reglas se instalen.

Es importante arrancar el controlador primero, porque cuando Mininet crea los switches, estos intentan conectarse inmediatamente al controlador. Si no está corriendo, los switches quedan sin controlador y no funcionan.

4.2. Verificación Básica de Conectividad

Antes de probar reglas específicas, verificamos que el sistema funciona correctamente y que el firewall bloquea donde debe.

4.2.1. Prueba 1: PingAll

Para tener una visión general de la conectividad, ejecutamos `pingall` en Mininet:

```
mininet> pingall
```

Los resultados muestran que la mayoría de los hosts se comunican correctamente, excepto h2 y h3 que tienen comunicación bloqueada por la regla del firewall.

```
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:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:================================================================================
INFO:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:Switch 00-00-00-00-00-01 conectado
INFO:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:--------------------------------------------------------------------------------
INFO:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:  [1/5] Bloquear todo el tráfico al puerto 80
INFO:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:  [2/5] Bloquear todo el tráfico al puerto 80
INFO:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:  [3/5] Bloquear UDP desde host 1 con puerto destino 5001
INFO:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:  [4/5] Bloquear comunicación de host 2 a host 3
INFO:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:  [5/5] Bloquear comunicación de host 3 a host 2
INFO:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:--------------------------------------------------------------------------------
INFO:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:Firewall configurado en 00-00-00-00-00-01: 5 reglas instaladas
INFO:.home.matias.Documentos.Redes.Introduccion-a-los-Sistemas-Distribuidos-75.43-TP-N-2-Software-Defined-Networks.controller.firewall:================================================================================
```

Figura 4: Logs del controlador mostrando la instalación de las 4 reglas en el switch S1

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

Figura 5: Resultado del comando pingall. Se observa conectividad completa excepto entre h2 y h3

Para confirmar que el firewall está realmente bloqueando el tráfico, capturamos tráfico en la interfaz que conecta ambos switches (s1-eth1):

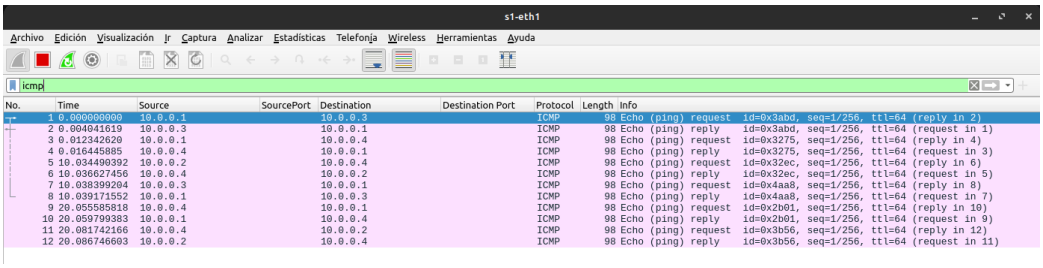

Figura 6: Captura en la interfaz s1-eth1. Los pings entre h2 y h3 no atraviesan el switch, confirmando que el firewall los descarta antes de reenviarlos

4.3. Validación de Reglas Específicas del Firewall

Las siguientes pruebas verifican cada regla configurada en el firewall, asegurando que bloquean el tráfico correcto sin afectar el resto de las comunicaciones permitidas.

4.3.1. Prueba 2: Bloqueo de Puerto 80 (TCP y UDP) - Regla 1

El firewall tiene dos reglas que bloquean todo el tráfico hacia el puerto 80, tanto TCP como UDP. Para verificarlo, probamos con diferentes herramientas y analizamos el tráfico capturado.

Prueba con HTTP (TCP):

```
mininet> h1 python3 -m http.server 80 &
mininet> h4 curl -m 5 http://10.0.0.1:80
```

El comando `curl` falla con timeout, confirmando que no puede establecer conexión.

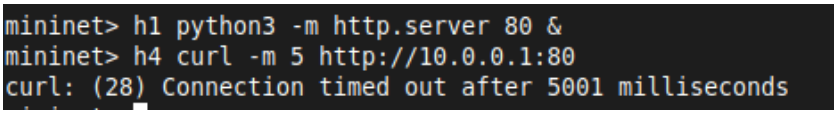

Figura 7: Intento fallido de conexión HTTP desde h4 hacia h1:80

Prueba con iperf (TCP):

```
mininet> h1 iperf -s -p 80 &
mininet> h4 iperf -c 10.0.0.1 -p 80 -t 5
```

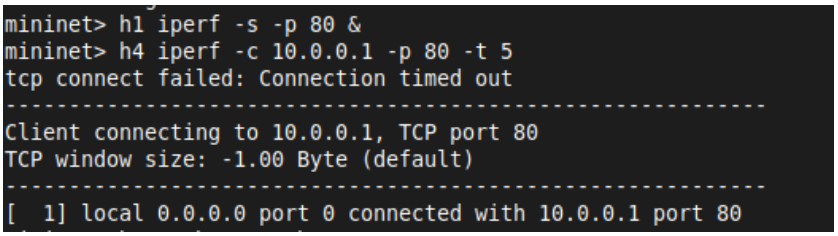

Figura 8: iperf tampoco logra establecer conexión en puerto 80

Para realizar el análisis con Wireshark, capturamos tráfico en distintas interfaces para ver dónde se está bloqueando. En s2-eth3 (salida de h4) vemos los intentos de conexión SYN, pero en s1-eth1 (enlace entre s1 y s2, después del firewall) no aparece nada, lo que confirma que s1 está descartando los paquetes.

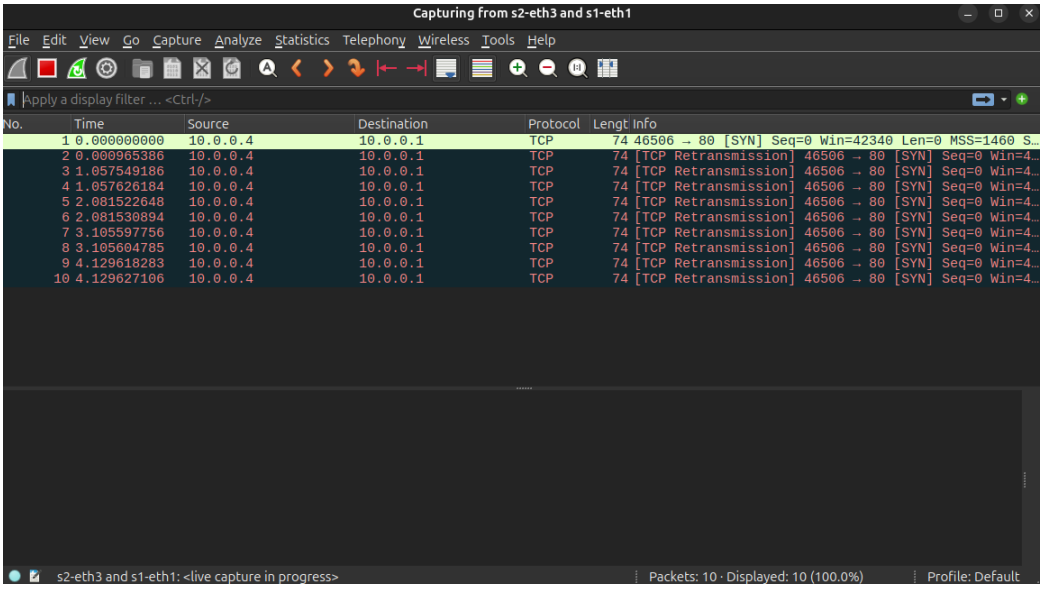

Figura 9: Captura de Wireshark mostrando paquetes SYN retransmitidos sin respuesta. El tráfico no atraviesa s1

Verificación con puerto permitido (8000):

Para confirmar que el problema es específicamente el puerto 80 y no un error de configuración general, probamos con el puerto 8000:

```
mininet> h1 python3 -m http.server 8000 &
mininet> h4 curl http://10.0.0.1:8000
```

Esta vez la conexión funciona perfectamente:

```
mininet> h1 python3 -m http.server 8000 &
mininet> h4 curl http://10.0.0.1:8000
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href=".git/">.git/</a></li>
<li><a href=".gitignore">.gitignore</a></li>
<li><a href=".idea/">.idea/</a></li>
<li><a href="controller/">controller/</a></li>
<li><a href="dns.py">dns.py</a></li>
<li><a href="informe/">informe/</a></li>
<li><a href="install_pox.sh">install_pox.sh</a></li>
<li><a href="pox/">pox/</a></li>
<li><a href="README.md">README.md</a></li>
<li><a href="run_controller.sh">run_controller.sh</a></li>
<li><a href="run_topology.sh">run_topology.sh</a></li>
<li><a href="topology.py">topology.py</a></li>
</ul>
<hr>
</body>
</html>
```

Figura 10: Conexión exitosa en puerto 8000 (no bloqueado)

El análisis de Wireshark muestra el handshake TCP completo y el intercambio HTTP normal:

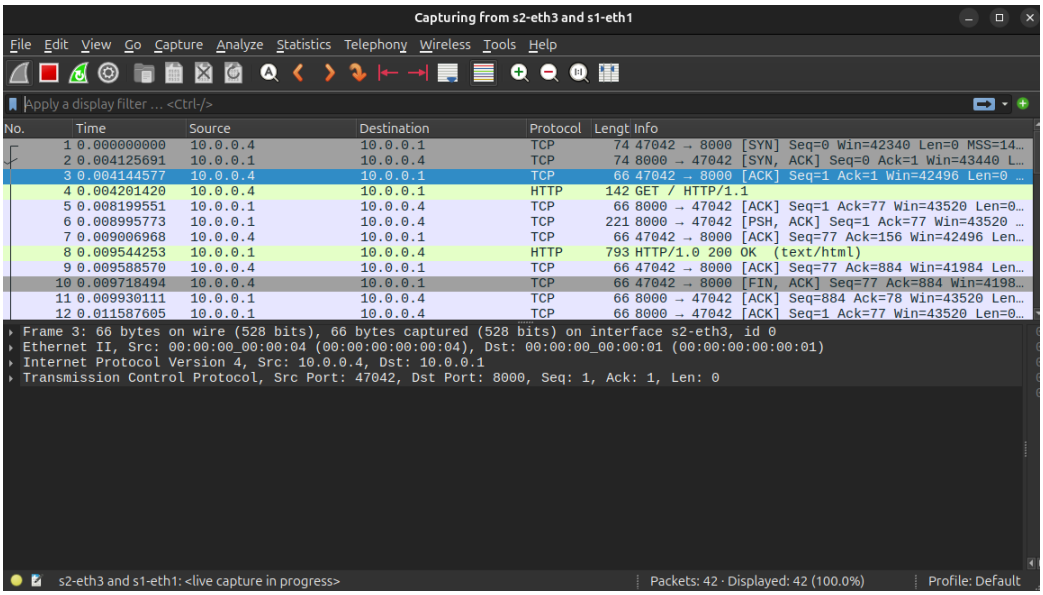

Figura 11: En puerto 8000 el tráfico fluye sin problemas, confirmando que el firewall es selectivo

4.3.2. Prueba 3: Bloqueo UDP específico - Regla 2

Esta regla bloquea específicamente el tráfico UDP desde h1 hacia el puerto 5001, pero permite que otros hosts usen ese mismo puerto. En esta prueba verificamos que el firewall filtra por dirección IP origen.

```
mininet> h4 iperf -s -u -p 5001 &
mininet> h1 iperf -c 10.0.0.4 -u -p 5001 -t 5  # Bloqueado
mininet> h2 iperf -c 10.0.0.4 -u -p 5001 -t 5  # Permitido
```

Los resultados muestran claramente la diferencia: h1 envía paquetes pero el servidor no recibe nada (0 %), mientras que h2 tiene comunicación normal (100 %).

```
mininet> h4 iperf -s -u -p 5001 &
mininet> h1 iperf -c 10.0.0.4 -u -p 5001 -t 5
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
10.0.0.4 - - [21/Nov/2025 21:46:45] "GET / HTTP/1.1" 200 -
------------------------------------------------------------
Client connecting to 10.0.0.4, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 41494 connected with 10.0.0.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-5.0134 sec   645 KBytes  1.05 Mbits/sec
[  1] Sent 450 datagrams
[  3] WARNING: did not receive ack of last datagram after 10 tries.
mininet> h2 iperf -c 10.0.0.4 -u -p 5001 -t 5
------------------------------------------------------------
Client connecting to 10.0.0.4, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.2 port 58549 connected with 10.0.0.4 port 5001
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-5.0134 sec   645 KBytes  1.05 Mbits/sec
[  1] Sent 450 datagrams
[  1] Server Report:
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  1] 0.0000-5.0083 sec   645 KBytes  1.05 Mbits/sec   0.002 ms 0/449 (0%)
```

Figura 12: Comparación de tráfico UDP. h1 está bloqueado, h2 funciona correctamente

Las capturas de Wireshark en s1-eth1 confirman que el tráfico de h1 no atraviesa el switch, mientras que el de h2 sí:

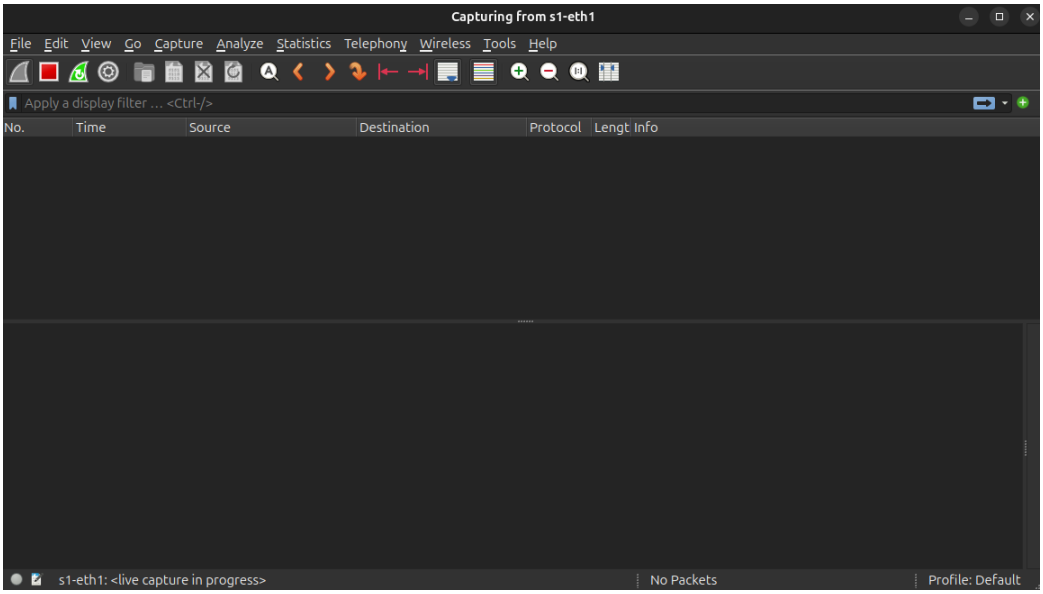

Figura 13: Tráfico UDP de h1 bloqueado: no aparece en la interfaz entre switches

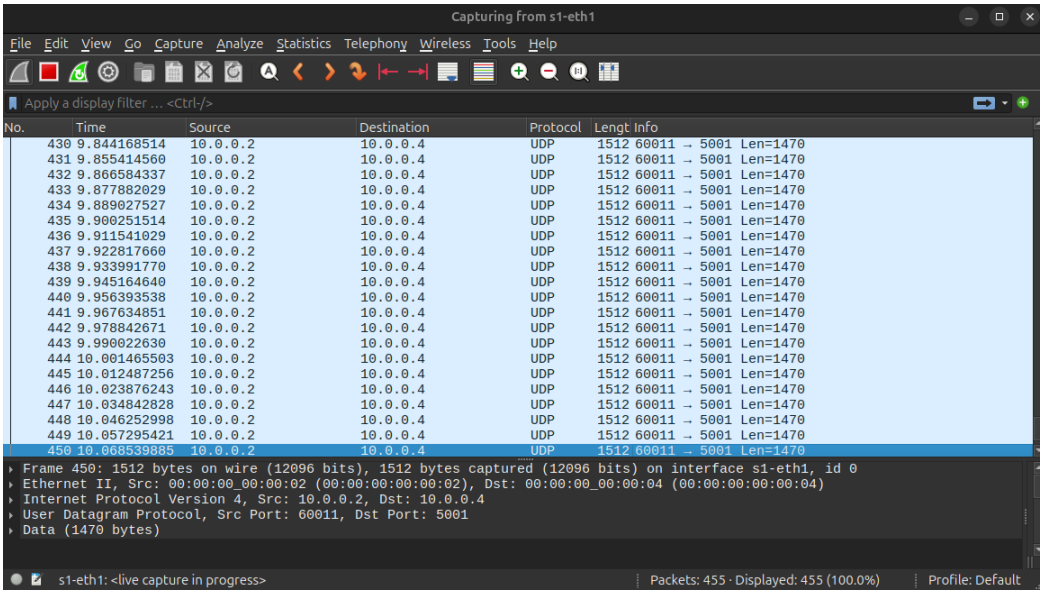

Figura 14: Tráfico UDP de h2 permitido: los datagramas atraviesan S1 sin problemas

4.3.3. Prueba 4: Bloqueo Bidireccional - Regla 3

Esta regla bloquea toda comunicación entre h2 y h3, independientemente del protocolo o puerto. Esto simula un aislamiento completo entre dos hosts.

Prueba con ICMP:

```
mininet> h2 ping -c 4 h3
mininet> h3 ping -c 4 h2
```

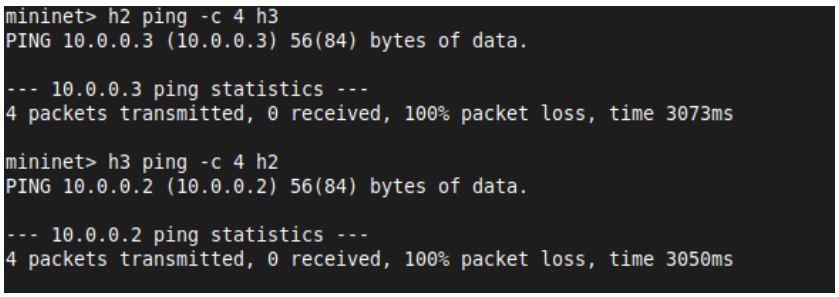

Figura 15: 100 % packet loss en ambas direcciones

Prueba con TCP:

```
mininet> h3 iperf -s &
mininet> h2 iperf -c 10.0.0.3 -t 5
```

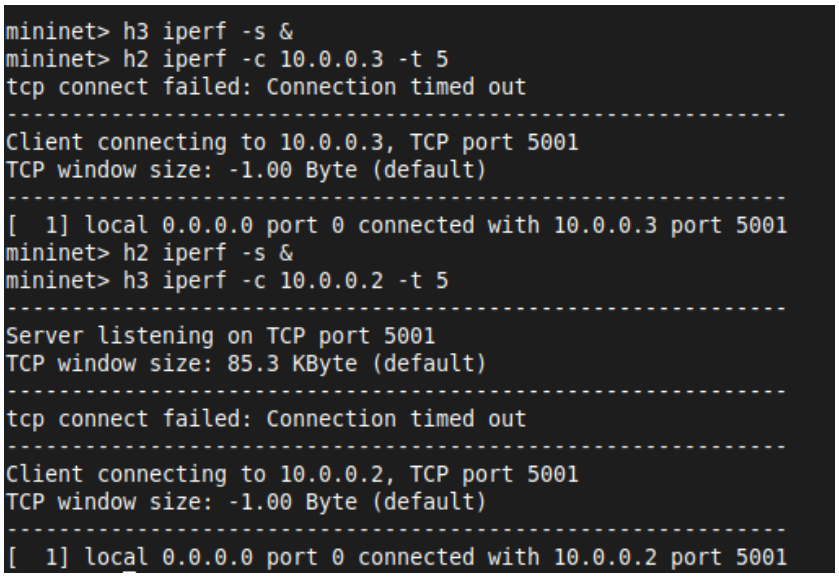

Figura 16: Conexión TCP también bloqueada entre h2 y h3

Para asegurar que el bloqueo es específico y no un problema general de red, verificamos que ambos hosts pueden comunicarse normalmente con otros:

```
mininet> h2 ping -c 4 h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=1.16 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.726 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.078 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.056 ms

--- 10.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3035ms
rtt min/avg/max/mdev = 0.056/0.504/1.158/0.463 ms
```

Figura 17: h2 puede comunicarse sin problemas con h1

```
mininet> h3 ping -c 4 h4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=1.97 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.625 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.072 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=64 time=0.080 ms

--- 10.0.0.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3057ms
rtt min/avg/max/mdev = 0.072/0.686/1.970/0.774 ms
```

Figura 18: h3 puede comunicarse sin problemas con h4

4.4. Escalabilidad: Pruebas con Diferentes Topologías

Una ventaja del diseño parametrizable es que podemos probar el sistema con distintas configuraciones. Ejecutamos las mismas pruebas de firewall en topologías de diferente tamaño para verificar que todo sigue funcionando.

4.4.1. Topología con N=1 (Caso Mínimo)

```
./run_topology.sh 1
```

Con un solo switch, todos los hosts están conectados al mismo equipo (topología estrella). El firewall se instala en S1 y bloquea el tráfico correctamente. Esta configuración es útil para pruebas rápidas y debugging, ya que elimina la complejidad de múltiples saltos.

4.4.2. Topología con N=3 (Cadena Extendida)

```
./run_topology.sh 3
```

Con tres switches en cadena, los hosts extremos (h1 en S1 y h4 en S3) están separados por dos saltos. Esta configuración prueba:

- **L2 learning en múltiples switches**: El controlador debe aprender las MACs en cada switch de la cadena
- **Propagación de reglas**: El firewall solo se instala en S1, pero afecta a todo el tráfico que pasa por ahí
- **Latencia**: Como era de esperar, los pings entre h1 y h4 tienen mayor latencia que en N=1 o N=2

En ambos casos las reglas del firewall funcionan igual, demostrando que el diseño es robusto y escalable.

Nota: En estas pruebas, todas las reglas están configuradas para aplicarse en el switch S1 (mediante el campo `"switch": 1` en el archivo JSON). El firewall permite configurar reglas específicas para cada switch según las necesidades de segmentación de la red.

5. Preguntas a responder

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

Los switches y los routers son dispositivos esenciales para que una red funcione. Ambos reciben paquetes, toman decisiones de reenvío y permiten que distintos equipos se comuniquen.

Sin embargo, presentan diferencias claras:

1. **Capa en la que operan:** Un switch trabaja en la capa de enlace de datos, utiliza direcciones MAC y una tabla de conmutación para decidir por qué puerto enviar cada frame. Un router opera en la capa de red, utiliza direcciones IP y una tabla de enrutamiento para determinar la mejor ruta hacia otras redes.
2. **Tipo de comunicación que habilitan:** El switch conecta dispositivos dentro de la misma red local (LAN). El router conecta diferentes redes entre sí (LAN–LAN, LAN–Internet), permitiendo alcanzar destinos externos.

3. **Reenvío de datos:** El switch aprende qué dirección MAC está asociada a cada puerto y envía el frame únicamente al puerto correspondiente. El router requiere configuración IP y emplea algoritmos de enrutamiento para decidir la ruta óptima. Al procesar información de la capa de red, el reenvío suele ser más costoso computacionalmente.

4. **Funciones de seguridad y control:** El switch ofrece controles básicos a nivel de puerto. El router puede aplicar reglas más avanzadas (como NAT o firewall), gracias a que analiza información de las capas de red y transporte.

En cuanto a sus similitudes:

1. Ambos son dispositivos de conmutación de paquetes (store-and-forward).

2. Ambos toman decisiones de reenvío utilizando una tabla interna (MAC table / routing table).

3. Forman parte crítica de la infraestructura de red.

4. En un entorno SDN, ambos pueden administrarse desde un controlador central que define cómo deben manejar el tráfico.

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

La diferencia principal es que en un switch convencional el plano de control y el plano de datos se ejecutan en el mismo dispositivo. En cambio, un switch OpenFlow separa estos planos: la responsabilidad del reenvío de paquetes reside en el switch, mientras que las decisiones de enrutamiento están centralizadas en un controlador SDN externo que se comunica con el switch a través del protocolo OpenFlow.

La principal diferencia entre un switch convencional y un switch OpenFlow radica en el grado de control y programabilidad que ofrecen. Mientras que un switch tradicional opera de manera autónoma y se limita a aprender direcciones MAC y reenviar tramas siguiendo una lógica fija definida por el fabricante, un switch OpenFlow delega todas las decisiones de forwarding en un controlador central. Esto implica que su comportamiento no está determinado por su firmware, sino por reglas que el controlador instala dinámicamente.

Gracias a esta separación entre el plano de datos y el plano de control, OpenFlow permite modificar políticas y flujos en tiempo real sin intervención

manual en el dispositivo, logrando una flexibilidad mucho mayor que la de un switch convencional. En lugar de "aprender" la red como lo haría un switch tradicional, un switch OpenFlow no actúa por iniciativa propia: cada decisión de reenvío depende de las instrucciones que reciba del controlador, lo que introduce un modelo operativo completamente centralizado.

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

Si bien las redes definidas por software y el protocolo OpenFlow representan un avance significativo en cuanto a flexibilidad y programabilidad dentro de un dominio controlado, no es factible reemplazar todos los routers de Internet por switches OpenFlow, especialmente en el escenario Inter-AS.

La arquitectura de Internet se sostiene sobre la autonomía administrativa de miles de Sistemas Autónomos (ASes), cada uno responsable de sus propias políticas de ruteo, y cuya coordinación global depende del protocolo BGP. A diferencia de los entornos Intra-AS o los data centers, donde SDN funciona con gran éxito gracias a la separación entre el plano de control y el plano de datos y a la centralización lógica del controlador, el ruteo Inter-AS no busca optimizar el camino más corto, sino que respeta políticas comerciales complejas, restricciones contractuales y decisiones estratégicas entre operadores, algo que OpenFlow no está diseñado para manejar.

Además, la escala global del ruteo en Internet hace inviable depender de un controlador SDN que deba computar y distribuir continuamente información de enrutamiento equivalente a las enormes tablas de prefijos que hoy gestionan los routers BGP. Centralizar la lógica de control no solo sería prohibitivo en términos de comunicación y procesamiento, sino que también iría en contra del modelo distribuido que permitió que Internet escale y se mantenga robusta.

A esto se suman cuestiones prácticas: OpenFlow requiere infraestructura adicional, introduce dependencia hacia un controlador remoto y puede generar latencia en la instalación de flujos; mientras que los routers tradicionales, altamente optimizados y probados, ofrecen fiabilidad, compatibilidad con múltiples protocolos y rendimiento predecible incluso bajo condiciones extremas.

6. Conclusión

En este trabajo implementamos una topología de red parametrizable y un controlador con funciones de firewall utilizando el paradigma de Redes

Definidas por Software (SDN) y el protocolo OpenFlow. Esto nos permitió ver en la práctica la separación entre el plano de control y el de datos, delegando la "inteligencia" de la red al controlador POX y dejando a los switches únicamente como dispositivos de reenvío.

Incluimos en nuestro desarrollo un sistema de reglas flexible, ya que implementamos la carga dinámica desde un archivo JSON externo. Esta decisión de diseño nos facilita reconfigurar la seguridad de la red sin necesidad de detener o modificar el controlador, lo que demuestra tambien la versatilidad de este enfoque frente a las configuraciones estáticas tradicionales.

Finalmente, validamos la solución mediante pruebas de conexión con iperf y análisis de tráfico con Wireshark, comprobando que el controlador instalaba correctamente los flujos de descarte y permitía el tráfico según lo esperado