

# 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
PATÍÑO, 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

<b>1. Introducción</b>	<b>2</b>
<b>2. Herramientas utilizadas</b>	<b>2</b>
<b>3. Implementación</b>	<b>3</b>
3.1. Arquitectura General . . . . .	3
3.2. Conexión Switch-Controlador . . . . .	3
3.3. Topología Parametrizable . . . . .	4
3.3.1. Diseño . . . . .	4
3.3.2. Caso Especial: $N = 1$ . . . . .	4
3.3.3. Implementación Técnica . . . . .	4
3.3.4. Topologías de Ejemplo . . . . .	4
3.4. Controlador SDN . . . . .	5
3.4.1. L2 Learning . . . . .	6
3.4.2. Firewall . . . . .	6
3.5. Reglas del Firewall Implementadas . . . . .	7
3.5.1. Regla 1: Bloqueo de Puerto 80 . . . . .	8
3.5.2. Regla 2: Bloqueo Específico UDP . . . . .	8
3.5.3. Regla 3: Bloqueo Bidireccional . . . . .	8
3.6. Estructura del Proyecto . . . . .	9
3.7. Decisiones de Diseño . . . . .	9
3.7.1. Uso de L2 Learning Existente . . . . .	9
3.7.2. Validación Extensiva . . . . .	10
3.8. Scripts de Automatización . . . . .	10
3.9. Compatibilidad y Modificaciones Técnicas . . . . .	10
3.9.1. Fix de Compatibilidad Python 3 . . . . .	10
<b>4. Pruebas y Validación</b>	<b>11</b>
4.1. Configuración del Entorno de Pruebas . . . . .	11
4.2. Verificación Básica: Conectividad . . . . .	11
4.2.1. Prueba 1: PingAll . . . . .	11
4.3. Pruebas de Firewall . . . . .	13
4.3.1. Prueba 2: Bloqueo de Puerto 80 (HTTP) - Regla 1 . . . . .	13
4.3.2. Prueba 3: Bloqueo UDP Específico - Regla 2 . . . . .	16
4.3.3. Prueba 4: Bloqueo Bidireccional - Regla 3 . . . . .	18
4.4. Pruebas con Diferentes Topologías . . . . .	20
4.4.1. Topología con $N=1$ . . . . .	20
4.4.2. Topología con $N=3$ . . . . .	20

<b>5. Preguntas a responder</b>	<b>21</b>
5.1. ¿Cuál es la diferencia entre un switch y un router? ¿Qué tienen en común? . . . . .	21
5.2. ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow? . . . . .	22
5.3. ¿Se pueden reemplazar todos los routers de la Internet por Switches OpenFlow? . . . . .	22
<b>6. Conclusión</b>	<b>23</b>

## 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:  
 $S_1 - S_2 - \dots - S_N$
- **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  $N$ :

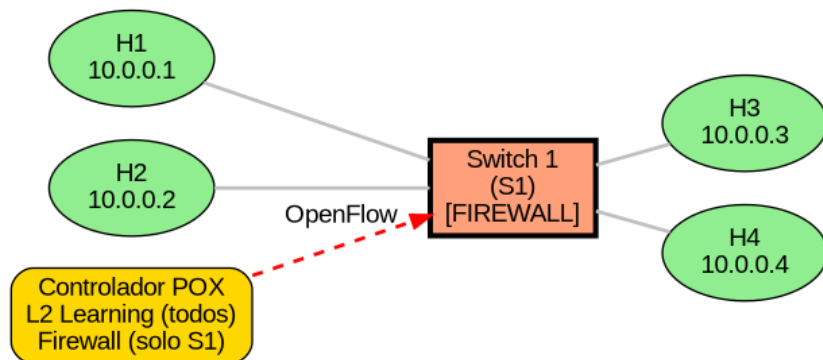


Figura 1: Topología con N=1 switches

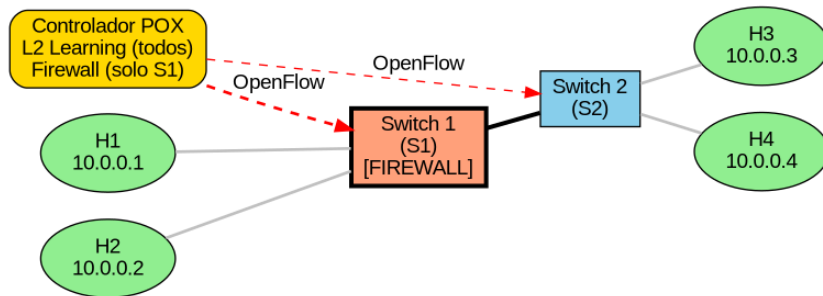


Figura 2: Topología con N=2 switches

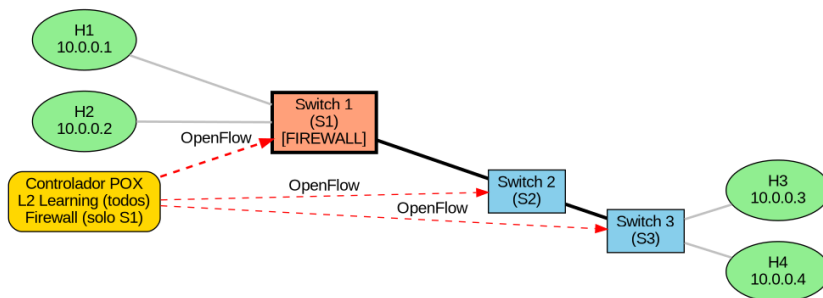


Figura 3: Topología con N=3 switches

### 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:

Las reglas del firewall se aplican **únicamente en el switch S1** (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 por el firewall.
- **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.

La implementación define una lista `controlled_switches = [1]` que especifica qué switches deben ejecutar el firewall. Al conectarse un switch, el controlador verifica su DPID y solo instala reglas si coincide con esta lista.

#### 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)
- Verificación de prerequisites de OpenFlow (ej: puerto requiere protocolo)
- Detección de errores comunes (ej: ICMP no puede tener puertos TCP/UDP)
- 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  
}
```

Esta regla previene el acceso HTTP en cualquier dirección.

### 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  
}
```

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

### 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"  
},  
{  
  "description": "Bloquear comunicación de host 3 a host 2",
```

```
"src_ip": "10.0.0.3",  
"dst_ip": "10.0.0.2"  
}
```

Esto asegura el bloqueo total de comunicación entre ambos hosts, 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

Esta sección describe las pruebas realizadas para verificar el correcto funcionamiento de la topología, el controlador SDN y las reglas del firewall.

### 4.1. Configuración del Entorno de Pruebas

Todas las pruebas se realizaron siguiendo el siguiente procedimiento:

1. **Iniciar el controlador** (Terminal 1):

```
./run_controller.sh
```

2. **Iniciar la topología** (Terminal 2):

```
./run_topology.sh 2    # Para N=2 switches
```

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

**Importante:** El controlador debe iniciarse *antes* que la topología para que los switches puedan conectarse al momento de su creación.

### 4.2. Verificación Básica: Conectividad

#### 4.2.1. Prueba 1: PingAll

**Objetivo:** Verificar que todos los hosts pueden comunicarse entre sí (excepto donde el firewall lo impide).

**Comando:**

```
mininet> pingall
```

**Resultado Esperado:**

- $h_1 \leftrightarrow h_2$ : Conectividad exitosa
- $h_1 \leftrightarrow h_3$ : Conectividad exitosa
- $h_1 \leftrightarrow h_4$ : Conectividad exitosa
- $h_2 \leftrightarrow h_3$ : Bloqueado (Regla 3)

- $h_2 \leftrightarrow h_4$ : Conectividad exitosa

- $h_3 \leftrightarrow h_4$ : Conectividad exitosa

### Logs del Controlador:

```
INFO: opening of 81 [00-00-00-00-01-2] connected
INFO: forwarding 12 learning Connection [00-00-00-00-01-2]
INFO: none.matis.Documentos.Redes.Introduccion a los Sistemas Distribuidos 75 43 TP-M-2-Software-Defined-Networks.controller.firewall: Switch 00-00-00-00-01 conectado
INFO: none.matis.Documentos.Redes.Introduccion a los Sistemas Distribuidos 75 43 TP-M-2-Software-Defined-Networks.controller.firewall: (1/5) Bloquear todo el tráfico al puerto 88
INFO: none.matis.Documentos.Redes.Introduccion a los Sistemas Distribuidos 75 43 TP-M-2-Software-Defined-Networks.controller.firewall: (2/5) Bloquear todo el tráfico al puerto 88
INFO: none.matis.Documentos.Redes.Introduccion a los Sistemas Distribuidos 75 43 TP-M-2-Software-Defined-Networks.controller.firewall: (3/5) Bloquear UDP desde host 1 con puerto destino 5801
INFO: none.matis.Documentos.Redes.Introduccion a los Sistemas Distribuidos 75 43 TP-M-2-Software-Defined-Networks.controller.firewall: (4/5) Bloquear comunicación de host 2 a host 3
INFO: none.matis.Documentos.Redes.Introduccion a los Sistemas Distribuidos 75 43 TP-M-2-Software-Defined-Networks.controller.firewall: (5/5) Bloquear comunicación de host 3 a host 2
INFO: none.matis.Documentos.Redes.Introduccion a los Sistemas Distribuidos 75 43 TP-M-2-Software-Defined-Networks.controller.firewall: Firewall configurado en 00-00-00-00-01: 5 reglas instaladas
```

Figura 4: Carga de las reglas al switch

Desde los logs del controlador podemos observar que las reglas fueron instaladas con éxito en el switch 1.

### Captura de Mininet:

```
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 de pingall

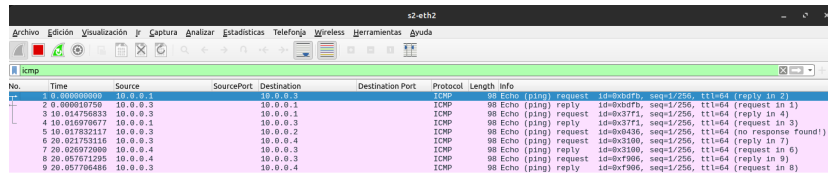
**Capturas de Wireshark:** Captura de interfaz de s1 que conecta con h2:

No.	Time	Source	SourcePort	Destination	Destination Port	Protocol	Length	Info
1	0.000000	10.0.0.1	10.0.0.2	10.0.0.1	10.0.0.1	ICMP	28	8 Echo (ping) request id=0x1271, seq=1/256, ttl=64 (request in 3)
2	0.00044585	10.0.0.2	10.0.0.1	10.0.0.1	10.0.0.1	ICMP	28	8 Echo (ping) reply id=0x1271, seq=1/256, ttl=64 (reply in 3)
3	0.002874187	10.0.0.2	10.0.0.1	10.0.0.1	10.0.0.1	ICMP	28	8 Echo (ping) request id=0x256, seq=1/256, ttl=64 (request in 4)
4	0.006850833	10.0.0.1	10.0.0.2	10.0.0.2	10.0.0.2	ICMP	28	8 Echo (ping) reply id=0x256, seq=1/256, ttl=64 (reply in 4)
5	0.040789658	10.0.0.2	10.0.0.3	10.0.0.3	10.0.0.3	ICMP	28	8 Echo (ping) request id=0x429c, seq=1/256, ttl=64 (request in 5)
6	0.042026389	10.0.0.2	10.0.0.4	10.0.0.4	10.0.0.4	ICMP	28	8 Echo (ping) request id=0x32ec, seq=1/256, ttl=64 (request in 7)
7	0.040938255	10.0.0.4	10.0.0.2	10.0.0.2	10.0.0.2	ICMP	28	8 Echo (ping) reply id=0x32ec, seq=1/256, ttl=64 (reply in 6)
8	0.040432359	10.0.0.4	10.0.0.2	10.0.0.2	10.0.0.2	ICMP	28	8 Echo (ping) request id=0x3b56, seq=1/256, ttl=64 (request in 9)
9	0.040483866	10.0.0.2	10.0.0.4	10.0.0.4	10.0.0.4	ICMP	28	8 Echo (ping) reply id=0x3b56, seq=1/256, ttl=64 (reply in 8)

Figura 6: Resultado de pingall para h2

Se puede observar que se envía un mensaje de ping a h3 pero no se recibe respuesta. Tampoco se recibe un ping de h3.

Captura de interfaz de s2 que conecta con h3:

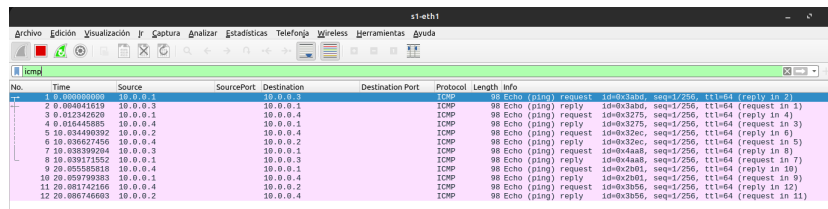


No.	Time	Source	SourcePort	Destination	Destination Port	Protocol	Length	Info
2	0.00000000	10.0.0.1		10.0.0.3		ICMP	60	Echo (ping) request id=0xb0df0, seq=1/256, ttl=64 (reply in 2)
3	0.000010759	10.0.0.3		10.0.0.1		ICMP	60	Echo (ping) reply id=0xb0df0, seq=1/256, ttl=64 (request in 2)
4	0.014756833	10.0.0.3		10.0.0.1		ICMP	60	Echo (ping) request id=0x37f1, seq=1/256, ttl=64 (request in 4)
5	0.016078677	10.0.0.1		10.0.0.3		ICMP	60	Echo (ping) reply id=0x37f1, seq=1/256, ttl=64 (request in 4)
6	0.017832117	10.0.0.3		10.0.0.2		ICMP	60	Echo (ping) request id=0x8436, seq=1/256, ttl=64 (no response found)
7	0.020723009	10.0.0.3		10.0.0.4		ICMP	60	Echo (ping) request id=0x3108, seq=1/256, ttl=64 (request in 7)
8	0.020772395	10.0.0.4		10.0.0.3		ICMP	60	Echo (ping) reply id=0x3108, seq=1/256, ttl=64 (request in 7)
9	0.057706486	10.0.0.3		10.0.0.4		ICMP	60	Echo (ping) request id=0xf906, seq=1/256, ttl=64 (request in 9)
10	0.057706486	10.0.0.3		10.0.0.4		ICMP	60	Echo (ping) reply id=0xf906, seq=1/256, ttl=64 (request in 9)

Figura 7: Resultado de pingall para h3

En este caso no se recibe respuesta del ping a h2 ni se recibe un ping de este ultimo.

Captura de interfaz de s1 que conecta con s2:



No.	Time	Source	SourcePort	Destination	Destination Port	Protocol	Length	Info
1	0.00000000	10.0.0.1		10.0.0.3		ICMP	60	Echo (ping) request id=0x33bd, seq=1/256, ttl=64 (reply in 2)
2	0.000041059	10.0.0.3		10.0.0.1		ICMP	60	Echo (ping) reply id=0x33bd, seq=1/256, ttl=64 (request in 1)
3	0.012342620	10.0.0.1		10.0.0.4		ICMP	60	Echo (ping) request id=0x3275, seq=1/256, ttl=64 (reply in 4)
4	0.014450885	10.0.0.4		10.0.0.1		ICMP	60	Echo (ping) reply id=0x3275, seq=1/256, ttl=64 (request in 3)
5	0.034980392	10.0.0.2		10.0.0.4		ICMP	60	Echo (ping) request id=0x32ec, seq=1/256, ttl=64 (reply in 6)
6	0.036627456	10.0.0.4		10.0.0.2		ICMP	60	Echo (ping) reply id=0x32ec, seq=1/256, ttl=64 (request in 5)
7	0.038390204	10.0.0.3		10.0.0.1		ICMP	60	Echo (ping) request id=0x44a8, seq=1/256, ttl=64 (reply in 8)
8	0.039171552	10.0.0.1		10.0.0.3		ICMP	60	Echo (ping) reply id=0x44a8, seq=1/256, ttl=64 (request in 7)
9	0.055850818	10.0.0.4		10.0.0.1		ICMP	60	Echo (ping) request id=0x2b01, seq=1/256, ttl=64 (reply in 10)
10	0.059799383	10.0.0.1		10.0.0.4		ICMP	60	Echo (ping) reply id=0x2b01, seq=1/256, ttl=64 (request in 9)
11	0.081742206	10.0.0.4		10.0.0.2		ICMP	60	Echo (ping) request id=0x3b56, seq=1/256, ttl=64 (reply in 12)
12	0.080746083	10.0.0.2		10.0.0.4		ICMP	60	Echo (ping) reply id=0x3b56, seq=1/256, ttl=64 (request in 11)

Figura 8: Resultado de pingall en los switches

Se observa que no se envian los pings de h2 a h3 y viceversa ya que estos son descartados.

## 4.3. Pruebas de Firewall

### 4.3.1. Prueba 2: Bloqueo de Puerto 80 (HTTP) - Regla 1

**Objetivo:** Verificar que todo el tráfico al puerto 80 es bloqueado.

**Configuración:**

# Terminal h1 - Iniciar servidor HTTP en puerto 80

```
mininet> h1 python3 -m http.server 80 &
```

# Terminal h4 - Intentar acceder desde otro host

```
mininet> h4 curl -m 5 http://10.0.0.1:80
```

**Resultado Esperado:** Timeout (conexión bloqueada por firewall)

**Resultado Obtenido:**

```
mininet> h1 python3 -m http.server 80 &  
mininet> h4 curl -m 5 http://10.0.0.1:80  
curl: (28) Connection timed out after 5001 milliseconds
```

Figura 9: Resultado de curl de h4 a h1

### Verificación con iperf:

# Servidor en h1

```
mininet> h1 iperf -s -p 80 &
```

# Cliente en h4

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

**Resultado:** No se establece conexión.

```
mininet> h1 iperf -s -p 80 &  
mininet> h4 iperf -c 10.0.0.1 -p 80 -t 5  
tcp connect failed: Connection timed out  
-----  
Client connecting to 10.0.0.1, TCP port 80  
TCP window size: -1.00 Byte (default)  
-----  
[ 1] local 0.0.0.0 port 0 connected with 10.0.0.1 port 80
```

Figura 10: Resultado de iperf de h4 a h1 en puerto 80

### Verificación con Wireshark

Para confirmar el bloqueo, se capturó tráfico en las interfaces relevantes del recorrido  $h4 \rightarrow h1$ . En s2-eth3 (salida de h4) se observan múltiples SYN retransmitidos, señal de que el cliente intenta iniciar la conexión sin obtener respuesta. En s1-eth1 (enlace entre s1 y s2) no aparece ningún paquete dirigido al puerto 80, lo que confirma que el firewall descarta el tráfico antes de que alcance el destino.

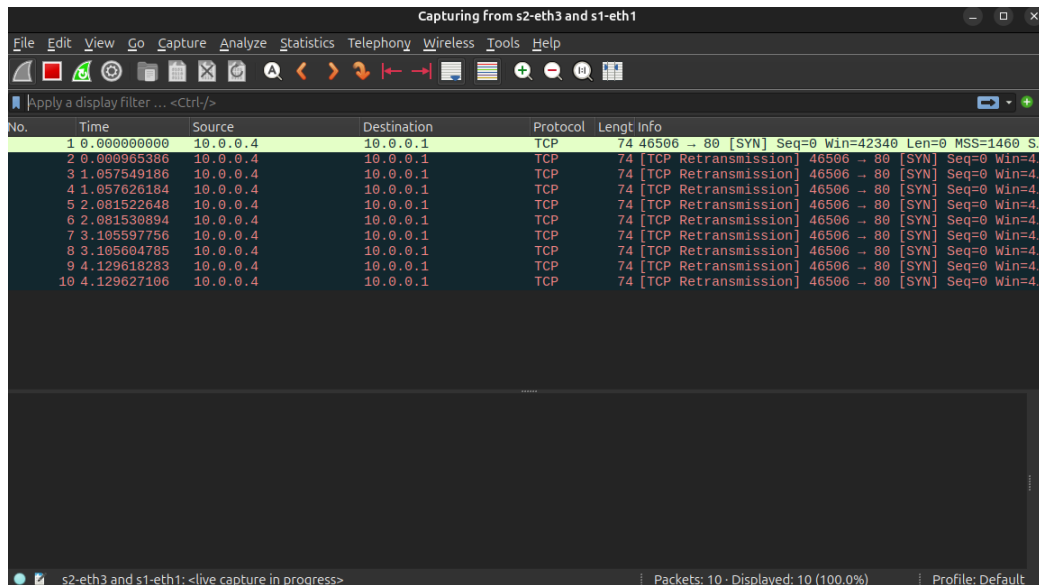


Figura 11: Captura de Wireshark: bloqueo de h4 a h1 en puerto 80

**Prueba de Control (puerto no bloqueado):**

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

**Resultado:** Conexión exitosa (el puerto 8000 no está bloqueado).

```
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 12: Resultado de curl de h4 a h1 en puerto 8000



### Verificación con Wireshark

Para este caso se capturó tráfico en las mismas interfaces que en la prueba anterior. En s2-eth3 y s1-eth1 se observa el three-way handshake completo (SYN, SYN/ACK, ACK), seguido por el intercambio normal de paquetes HTTP. La presencia del GET y la respuesta 200 OK confirma que el tráfico atraviesa ambos switches sin ser filtrado por el firewall.

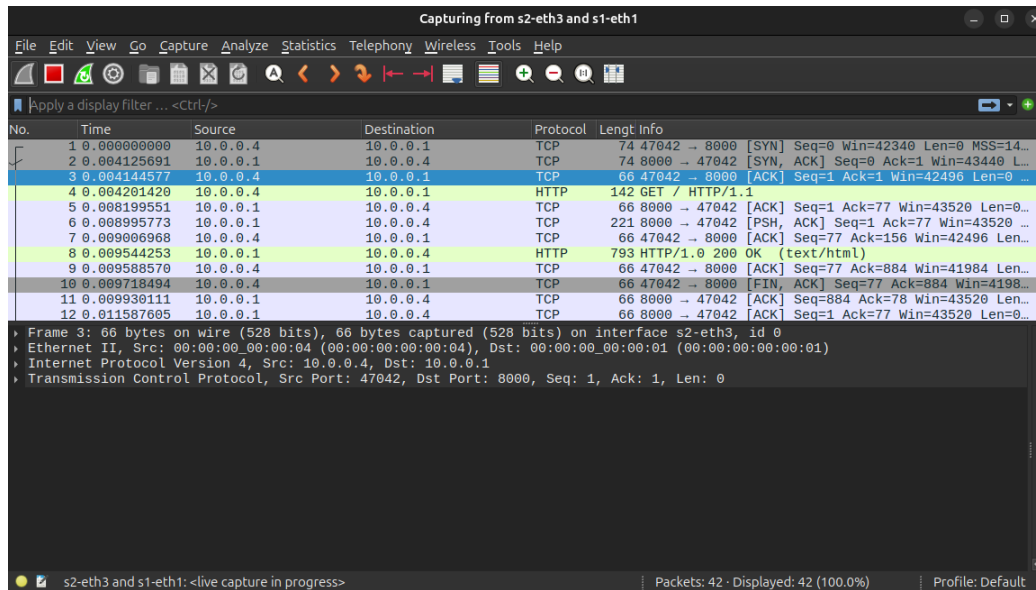


Figura 13: Captura de Wireshark: tráfico de h4 a h1 en puerto 8000

#### 4.3.2. Prueba 3: Bloqueo UDP Específico - Regla 2

**Objetivo:** Verificar que el tráfico UDP desde  $h_1$  al puerto 5001 es bloqueado.

**Configuración:**

```
# Servidor UDP en h4
mininet> h4 iperf -s -u -p 5001 &

# Cliente desde h1 (bloqueado)
mininet> h1 iperf -c 10.0.0.4 -u -p 5001 -t 5

# Cliente desde h2 (permitido)
mininet> h2 iperf -c 10.0.0.4 -u -p 5001 -t 5
```

**Resultados Esperados:**

- Desde  $h_1$ : 0 % de paquetes recibidos (bloqueado)
- Desde  $h_2$ : 100 % de paquetes recibidos (permitido)

### Resultados Obtenidos:

```

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 14: Resultado de iperfs udp de h1 y h2 a h4 en puerto 5001

### Verificación con Wireshark

Se realizó la captura en la interfaz s1-eth1, que conecta ambos switches. El tráfico UDP enviado desde h1 hacia el puerto 5001 no aparece en la captura, indicando que fue bloqueado antes de abandonar s1. En cambio, los datagramas provenientes de h2 sí se observan atravesando s1-eth1, lo que confirma que este flujo fue permitido.

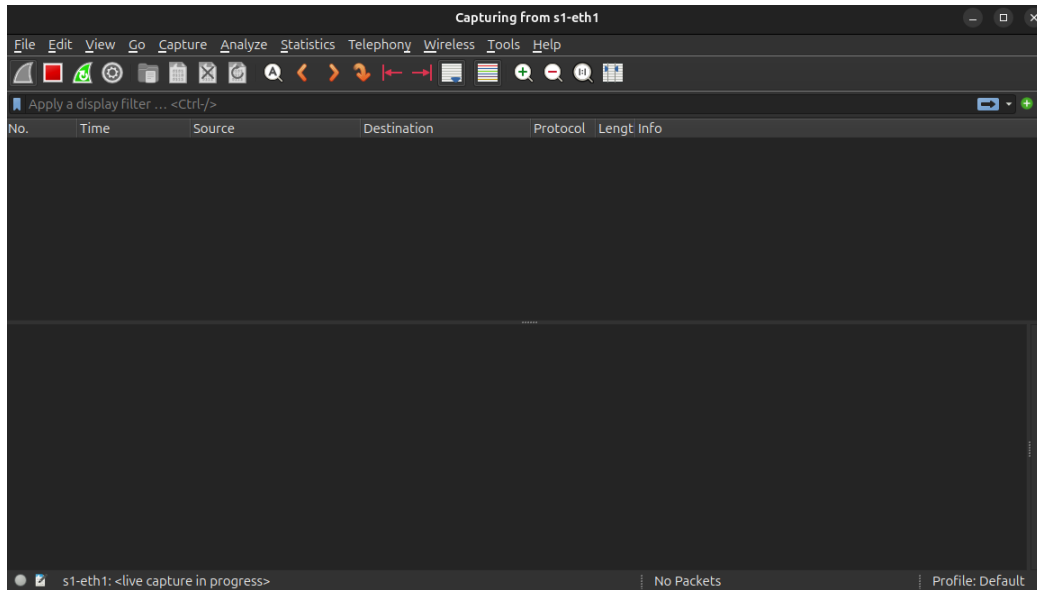


Figura 15: Captura de Wireshark: tráfico UDP de h1 a puerto 5001 bloqueado

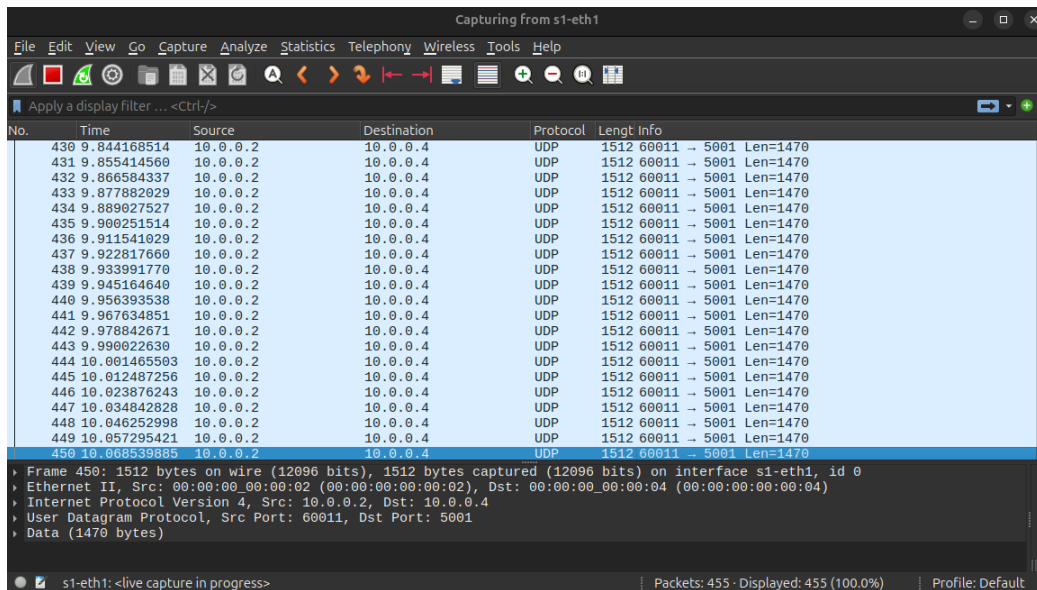


Figura 16: Captura de Wireshark: tráfico UDP de h2 a puerto 5001 permitido

### 4.3.3. Prueba 4: Bloqueo Bidireccional - Regla 3

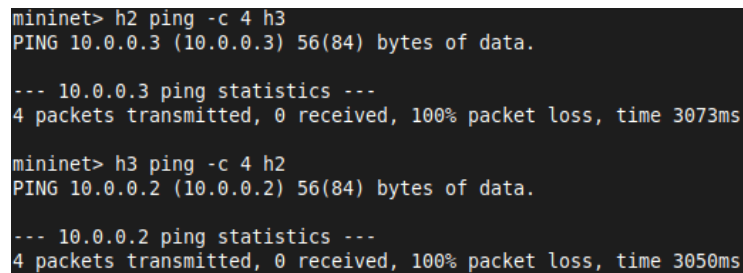
**Objetivo:** Verificar que  $h_2$  y  $h_3$  no pueden comunicarse en ninguna dirección.

#### Test 1: ICMP (ping)

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

**Resultado Esperado:** 100 % packet loss en ambas direcciones.

**Resultados Obtenidos:**



```
mininet> h2 ping -c 4 h3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.

--- 10.0.0.3 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3073ms

mininet> h3 ping -c 4 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

--- 10.0.0.2 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3050ms
```

Figura 17: Resultado de pings entre h2 y h3

#### Test 2: TCP (iperf)

```
mininet> h3 iperf -s &
mininet> h2 iperf -c 10.0.0.3 -t 5
mininet> h2 iperf -s &
mininet> h3 iperf -c 10.0.0.2 -t 5
```

**Resultado Esperado:** No se establece conexión TCP.

**Resultado Obtenido:**

```
mininet> h3 iperf -s &
mininet> h2 iperf -c 10.0.0.3 -t 5
tcp connect failed: Connection timed out
-----
Client connecting to 10.0.0.3, TCP port 5001
TCP window size: -1.00 Byte (default)
-----
[ 1] local 0.0.0.0 port 0 connected with 10.0.0.3 port 5001
mininet> h2 iperf -s &
mininet> h3 iperf -c 10.0.0.2 -t 5
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
tcp connect failed: Connection timed out
-----
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: -1.00 Byte (default)
-----
[ 1] local 0.0.0.0 port 0 connected with 10.0.0.2 port 5001
```

Figura 18: Resultado de iperf entre h2 y h3

#### Prueba de Control:

```
mininet> h2 ping -c 4 h1      # Debería funcionar
mininet> h3 ping -c 4 h4      # Debería funcionar
```

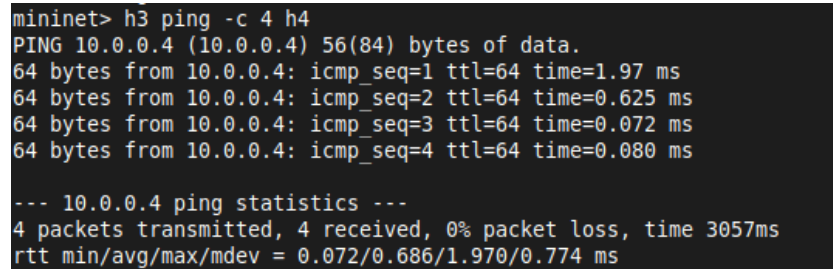
**Resultado:** Conectividad exitosa (solo está bloqueada la comunicación entre h2 y h3).

#### Resultado Obtenido:

```
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 19: Resultado de ping entre h2 y 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 20: Resultado de ping entre h3 y h4

## 4.4. Pruebas con Diferentes Topologías

### 4.4.1. Topología con N=1

**Configuración:**

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

**Características:** Todos los hosts conectados al mismo switch (topología estrella).

**Resultado:** Las reglas del firewall se aplican correctamente independientemente de la topología.

### 4.4.2. Topología con N=3

**Configuración:**

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

**Características:** Cadena de 3 switches, aumenta la distancia entre hosts extremos.

**Resultado:**

- L2 learning funciona correctamente en topología extendida
- Firewall mantiene las reglas en todos los switches
- Latencia aumenta proporcionalmente con el número de saltos

## 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 también 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