FACULTAD DE INGENIERIA
Universidad de Buenos Aires

# Trabajo Práctico N°2
# Software-Defined Networks

[TA048] Redes
Cátedra Hamelin-Alvarez
Curso martes (Lopez/Horn)
Primer cuatrimestre del 2025


| **Alumno** | **Padrón** | **Email** |
|---|---|---|
| Lorenzo Sebastián Ahumada | 106780 | lahumada@fi.uba.ar |
| Edgar Matías Campillay | 106691 | ecampillay@fi.uba.ar |
| Lara Daniela Converso | 107632 | lconverso@fi.uba.ar |
| Hernán de San Vicente | 108800 | hdesanvicente@fi.uba.ar |
| Tobías Emilio Serpa | 108266 | tserpa@fi.uba.ar |

# Índice


**1. Introduccion** **2**

**2. Implementación** **2**
2.1. Idea inicial . . . 2
2.1.1. Enfoque Basado en `PacketIn` . . . 2
2.1.2. Problemas Encontrados . . . 2
2.2. Implementación final . . . 2
2.2.1. Topología de Red . . . 3
2.2.2. Módulo Firewall . . . 3
2.2.3. Constructor de Flujos . . . 4

**3. Scripts** **4**
3.1. `start_pox.sh` . . . 4
3.2. `run_topo.sh` . . . 5

**4. Pruebas** **5**
4.1. Regla 1: Se deben descartar todos los mensajes cuyo puerto destino sea 80. . . . 6
4.2. Regla 2: Se deben descartar todos los mensajes que provengan del host 1, tengan como puerto destino el 5001, y estén utilizando el protocolo UDP. . . . 7
4.3. Regla 3: Se deben elegir dos hosts cualesquiera y los mismos no deben poder comunicarse de ninguna forma. . . . 8

**5. Preguntas a responder** **10**
5.1. ¿Cuál es la diferencia entre un Switch y un router? ¿Qué tienen en común? . . . 10
5.1.1. Similitudes . . . 10
5.1.2. Diferencias . . . 10
5.1.3. Método de reenvío . . . 10
5.1.4. Capacidad de redireccionamiento y seguridad . . . 11
5.2. ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow? . . . 11
5.2.1. Switch Convencional . . . 11
5.2.2. Switch OpenFlow . . . 11
5.3. ¿Se pueden reemplazar todos los routers de la Intenet por Switches OpenFlow? Piense en el escenario interASes para elaborar su respuesta 12
5.3.1. Funciones especializadas en el ruteo inter-AS . . . 12
5.3.2. Decisiones de enrutamiento basadas en políticas . . . 12
5.3.3. Autonomía y resiliencia de los dispositivos . . . 12
5.3.4. Escalabilidad y complejidad de Internet . . . 12

# 1. Introduccion

En este trabajo práctico implementamos un firewall utilizando el controlador POX y el emulador de redes Mininet. El objetivo es aplicar reglas de filtrado de tráfico en una topología de red definida, bloqueando o permitiendo el paso de paquetes según criterios configurables. Para ello, desarrollamos un módulo de firewall en Python que se integra con POX y lee reglas desde un archivo JSON, permitiendo así modificar el comportamiento del firewall sin necesidad de cambiar el código fuente.

# 2. Implementación

## 2.1. Idea inicial

### 2.1.1. Enfoque Basado en `PacketIn`

Nuestra idea inicial consistía en manejar eventos `PacketIn` para interceptar paquetes en el controlador y validar las reglas de firewall en tiempo real. El enfoque contemplaba un `rule_validator` que analizaría cada paquete recibido y decidiría si redireccionarlo o descartarlo.

### 2.1.2. Problemas Encontrados

Durante la implementación de este enfoque, nos encontramos con dos problemas principales:

- **Saturación de `PacketIn`:** Al interceptar todos los paquetes con PacketIn, recibíamos una cantidad masiva de tráfico (ARP, DHCP, broadcast, etc.) que no era relevante para las reglas de firewall. Esto saturaba el controlador con paquetes que no necesitábamos procesar.
- **Conflicto con `L2_Learning`:** El módulo `L2_learning` de POX instalaba flujos en las tablas de los switches basándose en las direcciones MAC aprendidas. Esto significaba que muchos paquetes nunca llegaban a nuestro controlador porque eran manejados directamente por los switches, evitando el evento `PacketIn` que necesitábamos para nuestro firewall. Buscando cómo poder recibir todos los mensajes, descubrimos que podíamos configurar reglas directamente en la tabla de flujos de los switches, lo que nos llevó a la implementación final.

Estos problemas nos llevaron a cambiar nuestro enfoque hacia la instalación proactiva de reglas de flujo en las tablas de los switches, evitando así la necesidad de procesar cada paquete individualmente en el controlador.

## 2.2. Implementación final

Nuestra implementación del firewall se basa en una arquitectura que separa claramente las responsabilidades entre el controlador POX y los switches OpenFlow. El sistema está compuesto por cuatro componentes principales:

1. Módulo Firewall (`firewall.py`): Maneja la lógica principal del firewall y los eventos OpenFlow.
2. Constructor de Flujos (`flow_builder.py`): Construye los mensajes OpenFlow a partir de las reglas de firewall.
3. Configuración de Reglas (`rules.json`): Define las políticas de bloqueo en formato JSON.
4. Topología de Red (`topology.py`): Define la estructura de la red con switches configurables.

### 2.2.1. Topología de Red

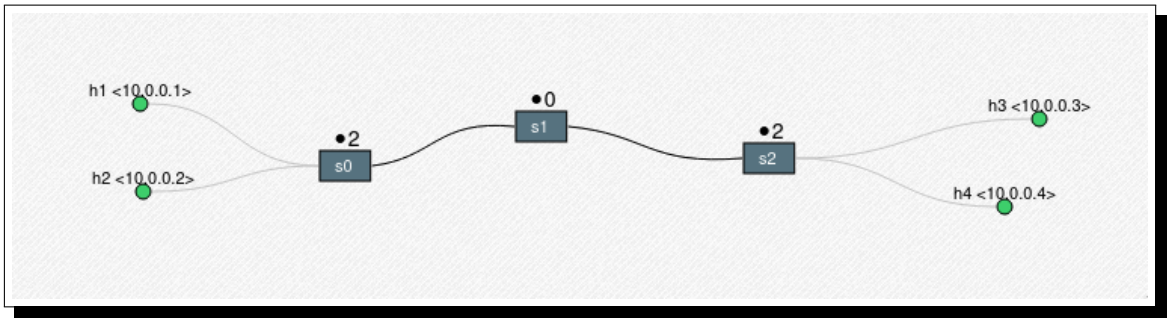
La topología implementada consiste en una red lineal configurable con N switches interconectados (N se recibe por parámetro). Los hosts se distribuyen estratégicamente:

- **Hosts 1 y 2:** Conectados al primer switch (s1)
- **Hosts 3 y 4:** Conectados al último switch (sN)

Esta configuración permite probar diferentes escenarios de comunicación:

- Comunicación local entre hosts del mismo switch
- Comunicación a través de múltiples switches

Comunicación a través de múltiples switches Aplicación selectiva de reglas de firewall en switches específicos La topología se define en `topology.py` y utiliza Mininet para crear un entorno de red virtual que simula switches OpenFlow reales conectados a un controlador POX remoto.




### 2.2.2. Módulo Firewall

El módulo firewall (`firewall.py`) implementa la clase Firewall que hereda de `EventMixin` para manejar eventos OpenFlow. Sus características principales incluyen:

- Inicialización parametrizada: El constructor acepta un parámetro `target_switches` que permite especificar en qué switches se instalarán las reglas. Este parámetro debe seguir el formato `a,b,...` teniendo los DPIDs de los switches donde se quiere tener las reglas separados por coma. Si no se especifica, las reglas se instalan en todos los switches.

- Manejo de conexiones: El método `_handle_ConnectionUp` se ejecuta cuando un switch se conecta al controlador. Verifica si el switch está en la lista de objetivos y, en caso afirmativo, procede a instalar las reglas del firewall.

- Instalación de reglas: El sistema instala reglas solo en los switches especificados, permitiendo un control granular sobre dónde se aplican las políticas de seguridad. Cada regla se traduce en un flujo OpenFlow con prioridad 100 y sin acciones (lo que resulta en el descarte del paquete).

### 2.2.3. Constructor de Flujos

El `FlowBuilder` es responsable de traducir las reglas JSON en mensajes OpenFlow válidos. Implementa dos métodos principales:

- Construcción de Flujos de Bloqueo: El método `build_blocking_flow` crea mensajes `ofp_flow_mod` con prioridad 100 y sin acciones, efectivamente bloqueando el tráfico que coincida con los criterios especificados.

- Construcción de Criterios de Coincidencia: El método `build_match` configura los campos de coincidencia OpenFlow basándose en las reglas JSON. Soporta múltiples criterios:

  - **Tipo de IP:** IPv4 (por defecto) o IPv6
  - **Protocolo de transporte:** TCP o UDP
  - **Direcciones IP:** Origen y destino
  - **Puertos:** Origen y destino
  - **Direcciones MAC:** Origen y destino

- El constructor maneja automáticamente la configuración de campos de puerto solo cuando se especifica un protocolo de transporte válido, evitando errores de OpenFlow.

# 3. Scripts

Desarrollamos dos scripts principales para automatizar el despliegue del sistema:

## 3.1. `start_pox.sh`

Inicia el controlador POX con el módulo firewall, sus parámetros son:

- **Sin parámetros:** Instala reglas en todos los switches

- **Un número:** Instala reglas solo en el switch especificado

```
./start_pox.sh 1 # Reglas instaladas en s1
```

- **Múltiples números separados por comas:** Instala reglas en los switches especificados

```
./start_pox.sh 1,3 # Reglas instaladas en s1 y s3
```

### 3.2. `run_topo.sh`

Ejecuta la topología Mininet, sus parámetros son:

- **Un número:** Crea una topología con el número de switches especificado

```
./run_topo.sh 5 # Agrega 5 switches a la topologia
```

Ambos scripts incluyen validación de archivos requeridos y manejo de errores para garantizar un despliegue confiable del sistema.

## 4. Pruebas

En primer lugar, aclaramos algunos datos importantes sobre el contexto en el que se prueba el cumplimiento de las reglas para que sea más fácil entender las capturas.

Las direcciones IP de los hosts son las siguientes:

- **h1:** 10.0.0.1,
- **h2:** 10.0.0.2,
- **h3:** 10.0.0.3,
- **h4:** 10.0.0.4

Para la topología colocamos 3 switches y el firewall se instaló en el primer switch, el cual está conectado a los hosts h1 y h2.

Por último es importante aclarar que las capturas de la consola provienen de las terminales de xterm, por lo que en los comandos no es necesario colocar el host al principio (cosa que sí es necesaria de ejecutar esto directamente en la consola de mininet)

## 4.1. Regla 1: Se deben descartar todos los mensajes cuyo puerto destino sea 80.

Para probar esta regla, establecemos como servidor al host h3 con el comando `iperf -s -p 80` y ejecutamos al cliente en el host h2 con el comando `iperf -c 10.0.0.3 -p 80` Esto enviará tráfico TCP desde h2 hacia el puerto 80 de h3

```
root@lorenzo-VirtualBox:/home/lorenzo/Escritorio/Redes/TP2# iperf -s -p 80
------------------------------------------------------------
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

```
root@lorenzo-VirtualBox:/home/lorenzo/Escritorio/Redes/TP2# iperf -c 10.0.0.3 -p 80
connect failed: Operation now in progress
```

Como podemos observar, h3 se queda escuchando en el puerto 80 y al intentar enviar tráfico desde h2, la consola nos devuelve `connect failed`. Esto ocurre debido a que se bloquean los paquetes del 3-way handshake y nunca se llega a establecer la conexión entre ambos hosts.

En Wireshark podemos apreciar esto mejor:

| No. | Time | Source | Destination | Protoco | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000000 | 10.0.0.2 | 10.0.0.3 | TCP | 74 | 39632 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1 |
| 2 | 1.029436882 | 10.0.0.2 | 10.0.0.3 | TCP | 74 | [TCP Retransmission] 39632 → 80 [SYN] Seq=0 |
| 3 | 3.044126915 | 10.0.0.2 | 10.0.0.3 | TCP | 74 | [TCP Retransmission] 39632 → 80 [SYN] Seq=0 |
| 4 | 5.219636910 | de:a5:27:… | 1a:41:dd:… | ARP | 42 | Who has 10.0.0.3? Tell 10.0.0.2 |
| 5 | 5.285039225 | 1a:41:dd:… | de:a5:27:… | ARP | 42 | 10.0.0.3 is at 1a:41:dd:1a:f0:2c |
| 6 | 7.268050431 | 10.0.0.2 | 10.0.0.3 | TCP | 74 | [TCP Retransmission] 39632 → 80 [SYN] Seq=0 |
| 7 | 23.3955303… | fe80::1c4… | ff02::2 | ICMPv6 | 70 | Router Solicitation from 1e:46:67:f7:4c:ee |

h2 envía un paquete a h3 para establecer la conexión, y al no recibir respuesta envía 3 retransmisiones más, sin lograr conectarse.

Si probamos esta regla, pero ahora enviando tráfico UDP, podemos ver una salida diferente de la consola que también nos demuestra que se bloquea correctamente:

```
root@lorenzo-VirtualBox:/home/lorenzo/Escritorio/Redes/TP2# iperf -c 10.0.0.3 -u -p 80
------------------------------------------------------------
Client connecting to 10.0.0.3, UDP port 80
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 21] local 10.0.0.2 port 56257 connected with 10.0.0.3 port 80
[ 21] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval       Transfer     Bandwidth
[ 21]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 21] Sent 892 datagrams
```

Al ser UDP, h2 puede enviar los paquetes hacia h3, pero como observamos en la captura, nunca recibe respuesta, inclusive luego de 10 intentos.

## 4.2. Regla 2: Se deben descartar todos los mensajes que provengan del host 1, tengan como puerto destino el 5001, y estén utilizando el protocolo UDP.

Para probar esta regla, establecemos como servidor al host h2 con el comando `iperf -s -u -p 5001` y ejecutamos al cliente en el host h1 con el comando `iperf -c 10.0.0.2 -u -p 5001` Esto enviará tráfico UDP desde h1 hacia el puerto 5001 de h2

```
root@lorenzo-VirtualBox:/home/lorenzo/Escritorio/Redes/TP2# iperf -s -u -p 5001
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
```

```
root@lorenzo-VirtualBox:/home/lorenzo/Escritorio/Redes/TP2# iperf -c 10.0.0.2 -u -p 5001
------------------------------------------------------------
Client connecting to 10.0.0.2, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 21] local 10.0.0.1 port 60915 connected with 10.0.0.2 port 5001
[ 21] WARNING: did not receive ack of last datagram after 10 tries.
[ ID] Interval       Transfer     Bandwidth
[ 21]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 21] Sent 892 datagrams
```

Podemos ver que h2 se queda escuchando en el puerto 5001, y cuando h1 envía tráfico udp hacia ese puerto, no recibe respuesta.

Si intentamos enviar tráfico al puerto 5001 de h2, pero esta vez desde h3, podemos ver que la salida de ambas terminales es distinta:

```
root@lorenzo-VirtualBox:/home/lorenzo/Escritorio/Redes/TP2# iperf -s -u -p 5001
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 21] local 10.0.0.2 port 5001 connected with 10.0.0.3 port 36674
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[ 21]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.015 ms    0/  893 (0%)
[ 21] 0.0000-9.9638 sec  5 datagrams received out-of-order
```

```
root@lorenzo-VirtualBox:/home/lorenzo/Escritorio/Redes/TP2# iperf -c 10.0.0.2 -
u -p 5001
------------------------------------------------------------
Client connecting to 10.0.0.2, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[ 21] local 10.0.0.3 port 36674 connected with 10.0.0.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[ 21]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec
[ 21] Sent 893 datagrams
[ 21] Server Report:
[ 21]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.014 ms    0/  893 (0%)
[ 21] 0.0000-9.9638 sec  5 datagrams received out-of-order
```

En este caso la regla no bloquea el tráfico debido a que la ip origen es 10.0.0.3, y podemos apreciar que los paquetes llegan a destino.

Observar esto en Wireshark es un poco más difícil, debido a que solo veremos cómo los paquetes son enviados a h2, pero en h2 veremos una captura vacía. Pese a esto, hay una pequeña diferencia entre el caso de bloqueo y el caso exitoso que podemos ver cuando enviamos los paquetes a h2.

```
901 11.5158709…  10.0.0.1   10.0.0.2   UDP      1512 60177 → 5001 Len=1470
902 11.7661953…  10.0.0.1   10.0.0.2   UDP      1512 60177 → 5001 Len=1470
903 12.0164795…  10.0.0.1   10.0.0.2   UDP      1512 60177 → 5001 Len=1470
904 12.2673645…  10.0.0.1   10.0.0.2   UDP      1512 60177 → 5001 Len=1470
```

```
894 9.992076274  10.0.0.3   10.0.0.2   UDP      1512 36674 → 5001 Len=1470
895 10.0040264…  10.0.0.3   10.0.0.2   UDP      1512 36674 → 5001 Len=1470
896 10.0365844…  10.0.0.3   10.0.0.2   UDP      1512 36674 → 5001 Len=1470
897 10.1186209…  10.0.0.2   10.0.0.3   UDP      1512 5001 → 36674 Len=1470
```

Cuando h1 envía el último paquete, no recibe ningún tipo de respuesta, mientras que cuando h3 envía el último paquete, podemos ver que recibió un paquete enviado por h2. Este último paquete que envía el servidor es lo que luego nos muestra la consola a modo de reporte de los paquetes que lograron llegar a destino.

## 4.3. Regla 3: Se deben elegir dos hosts cualesquiera y los mismos no deben poder comunicarse de ninguna forma.

Para probar esta regla, establecemos como servidor al host h4 con el comando `iperf -s` y ejecutamos al cliente en el host h2 con el comando `iperf -c 10.0.0.4` Esto enviará tráfico TCP desde h2 hacia h4

```
root@lorenzo-VirtualBox:/home/lorenzo/Escritorio/Redes/TP2# iperf -s
}------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

```
root@lorenzo-VirtualBox:/home/lorenzo/Escritorio/Redes/TP2# iperf -c 10.0.0.4
connect failed: Operation now in progress
```

Al igual que en el caso de la primera regla, h4 se queda escuchando e intentamos mandar tráfico desde h2. Nuevamente falla al intentar lograr la conexión debido a que se bloquean los paquetes del 3-way handshake.

Y en Wireshark podemos apreciarlo mejor:

| No. | Time | Source | Destination | Protoco | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000000 | 10.0.0.2 | 10.0.0.4 | TCP | 74 | 39074 → 5001 [SYN] Seq=0 Win=42340 Len=0 MSS=1 |
| 2 | 1.013557375 | 10.0.0.2 | 10.0.0.4 | TCP | 74 | [TCP Retransmission] 39074 → 5001 [SYN] Seq=0 |
| 3 | 3.023265266 | 10.0.0.2 | 10.0.0.4 | TCP | 74 | [TCP Retransmission] 39074 → 5001 [SYN] Seq=0 |
| 4 | 5.003655686 | fa:0d:c6:… | 0a:24:ef:… | ARP | 42 | Who has 10.0.0.4? Tell 10.0.0.2 |
| 5 | 5.065886641 | 0a:24:ef:… | fa:0d:c6:… | ARP | 42 | 10.0.0.4 is at 0a:24:ef:bd:fc:05 |
| 6 | 7.052942219 | 10.0.0.2 | 10.0.0.4 | TCP | 74 | [TCP Retransmission] 39074 → 5001 [SYN] Seq=0 |

h2 intenta realizar la conexión con h4 múltiples veces, pero falla en el intento y nunca establece la conexión.

Si invertimos los roles, estableciendo h2 como servidor y h4 como cliente, obtenemos el mismo resultado:

| No. | Time | Source | Destination | Protoco | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000000 | 10.0.0.4 | 10.0.0.2 | TCP | 74 | 60762 → 5001 [SYN] Seq=0 Win=42340 Len=0 MSS=14 |
| 2 | 1.015303761 | 10.0.0.4 | 10.0.0.2 | TCP | 74 | [TCP Retransmission] 60762 → 5001 [SYN] Seq=0 W |
| 3 | 3.032357770 | 10.0.0.4 | 10.0.0.2 | TCP | 74 | [TCP Retransmission] 60762 → 5001 [SYN] Seq=0 W |
| 4 | 5.084478233 | 0a:24:ef:… | fa:0d:c6:… | ARP | 42 | Who has 10.0.0.2? Tell 10.0.0.4 |
| 5 | 5.125123685 | fa:0d:c6:… | 0a:24:ef:… | ARP | 42 | 10.0.0.2 is at fa:0d:c6:7f:74:f0 |
| 6 | 7.129885967 | 10.0.0.4 | 10.0.0.2 | TCP | 74 | [TCP Retransmission] 60762 → 5001 [SYN] Seq=0 W |

h4 falla al intentar conectarse a h2 luego de múltiples intentos.

# 5. Preguntas a responder

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

Los switches y routers son dispositivos fundamentales para el funcionamiento de las redes informáticas, ya que permiten la transmisión eficiente de datos. A pesar de que ambos tienen funciones relacionadas con la conectividad y el direccionamiento de tráfico, operan en niveles distintos del modelo OSI y cumplen roles diferentes dentro de la red.

### 5.1.1. Similitudes

Ambos dispositivos comparten algunas características clave:

1. **Conectividad:** permiten la conexión de dispositivos para facilitar la comunicación en redes de computadoras.
2. **Toma de decisiones:** analizan la información que reciben para decidir cómo reenviar los datos.
3. **Seguridad:** pueden incluir mecanismos de seguridad, como control de acceso o segmentación del tráfico.
4. **Gestión centralizada:** en arquitecturas basadas en *Software Defined Networking (SDN)*, tanto switches como routers pueden ser administrados mediante un controlador central, lo que simplifica su configuración y operación.

### 5.1.2. Diferencias

**Función principal**

- **Switch:** su objetivo es facilitar la comunicación dentro de una red local *(LAN)*, operando en la **capa 2 (Enlace de datos)** del modelo OSI. Se encarga de reenviar tramas basándose en direcciones **MAC**.
- **Router** se utiliza para interconectar diferentes redes, actuando en la **capa 3 (Red)** del modelo OSI. Toma decisiones de enrutamiento basadas en direcciones IP.

### 5.1.3. Método de reenvío

- **Switch** reenvía los paquetes directamente al puerto correspondiente, ya que mantiene una tabla con las direcciones MAC asociadas a cada uno de sus puertos. Esto permite que la comunicación en una LAN sea rápida y eficiente.
- **Router** utiliza **tablas de enrutamiento** y protocolos (como OSPF o BGP) para determinar la mejor ruta hacia el destino. Su operación es más compleja y permite el envío de datos entre distintas redes.

### 5.1.4. Capacidad de redireccionamiento y seguridad

- **Switch** ofrece funciones básicas de segmentación y seguridad, pero no permite la comunicación entre redes distintas.
- **Router** tiene la capacidad de enrutar datos entre múltiples redes y aplicar reglas de seguridad avanzadas (por ejemplo, mediante **firewalls**, listas de control de acceso o NAT), lo cual lo convierte en un componente clave para la seguridad perimetral de una red.

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

### 5.2.1. Switch Convencional

Un switch convencional es un dispositivo que opera principalmente en redes locales (LAN) y cuya lógica de funcionamiento está definida por el fabricante. Sus principales características son: Reenvío basado en direcciones MAC: decide cómo reenviar tramas dentro de la red utilizando una tabla de direcciones MAC que construye de forma automática.

1. **Comportamiento autónomo:** el switch toma decisiones de manera local, sin requerir control externo, aprendiendo dinámicamente qué dispositivos están conectados a cada puerto.
2. **Dispositivo cerrado:** su comportamiento no puede ser modificado fácilmente; está predefinido por el fabricante y no permite personalización en su lógica de reenvío.

### 5.2.2. Switch OpenFlow

Un switch OpenFlow, en cambio, está diseñado para ser parte de una arquitectura de red definida por software (SDN). Se diferencia principalmente por lo siguiente:

1. **No toma decisiones por sí mismo:** delega la lógica de reenvío a un **controlador SDN** central, que actúa como cerebro de la red.
2. **Control externo mediante OpenFlow:** recibe instrucciones del controlador a través del protocolo OpenFlow, que define cómo debe manejar cada paquete.
3. Dispositivo programable: permite definir reglas personalizadas según múltiples criterios (dirección IP o MAC, tipo de tráfico, puertos, etiquetas, etc.).
4. **Separación de planos:** diferencia claramente el **plano de control** (donde se toman las decisiones) del **plano de datos** (donde se aplican), lo que ofrece mayor **flexibilidad, escalabilidad y control** sobre el comportamiento de la red.

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

No, no es viable reemplazar todos los routers de Internet por switches OpenFlow, especialmente en escenarios de interconexión entre Sistemas Autónomos (inter-AS). A continuación, se detallan las razones principales:

### 5.3.1. Funciones especializadas en el ruteo inter-AS

Los routers desempeñan un papel esencial en la comunicación entre diferentes redes autónomas (AS), utilizando protocolos de enrutamiento como **BGP (Border Gateway Protocol)**. Este tipo de ruteo requiere capacidades que van más allá de lo que un switch OpenFlow puede ofrecer actualmente.

### 5.3.2. Decisiones de enrutamiento basadas en políticas

En el contexto inter-AS, las decisiones de ruteo no se basan únicamente en la eficiencia (como la ruta más corta), sino también en **políticas administrativas** (por ejemplo, evitar ciertos países, proveedores o rutas por cuestiones legales, de seguridad o de costos). Estas políticas requieren lógica compleja y flexible, algo que no está completamente soportado por OpenFlow, que está más orientado a redes internas (intra-AS).

### 5.3.3. Autonomía y resiliencia de los dispositivos

Internet es una red distribuida y descentralizada. Los routers deben ser **autónomos** para continuar funcionando incluso ante fallos o desconexiones. En cambio, los switches OpenFlow dependen de un **controlador SDN central** para tomar decisiones. Si ese controlador falla o se vuelve inaccesible, los switches podrían quedar inoperativos, lo que resulta inaceptable en el contexto global de Internet.

### 5.3.4. Escalabilidad y complejidad de Internet

La infraestructura de Internet es altamente dinámica y de gran escala, con millones de dispositivos y rutas. Aunque SDN y OpenFlow son muy útiles en entornos controlados —como redes corporativas, centros de datos o dentro de un único sistema autónomo—, **no escalan fácilmente** al nivel de complejidad que presenta el enrutamiento global entre múltiples AS. OpenFlow requiere una **vista centralizada de la red** para operar eficazmente, algo que es prácticamente imposible en un entorno tan distribuido y heterogéneo como Internet.