FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TP N°2: OpenFlow Lite

[TA048] Redes
CURSO: 02-Alvarez Hamelin
Segundo cuatrimestre de 2025

Alumno/a:	GENERAL, Camila
Número de padrón:	105552
Email:	cgeneral@fi.uba.ar

Alumno/a:	VACCARELLI, Santiago
Número de padrón:	106051
Email:	svaccarelli@fi.uba.ar

Alumno/a:	RODRÍGUEZ NEGRI, Federica
Número de padrón:	104211
Email:	ferodriguezn@fi.uba.ar

Alumno/a:	KRESTA, Facundo Ariel
Número de padrón:	110857
Email:	facundoarielkresta@gmail.com

Alumno/a:	NIEVA, Ulises
Número de padrón:	107227
Email:	unieva@fi.uba.ar

26 de noviembre de 2025

Resumen

Este trabajo implementa un firewall programable mediante Software-Defined Networking (SDN) utilizando el protocolo `OpenFlow 1.0` y el controlador `POX`. Se desarrolló un sistema que centraliza las políticas de seguridad en el plano de control, permitiendo la configuración dinámica de reglas de filtrado a nivel de direcciones `MAC`, direcciones `IP` y puertos de transporte. La arquitectura separa el plano de datos (switches Open vSwitch) del plano de control (controlador `POX` en `Python`), aplicando las reglas de firewall proactivamente en un switch designado mientras los demás operan con aprendizaje `MAC` tradicional. Se implementaron mecanismos de bloqueo por protocolo (`TCP`, `UDP`, `ICMP` y `STMP`), por host específico, y por combinaciones de estos criterios. Las pruebas realizadas en una topología de 4 switches interconectados demostraron la efectividad del sistema para bloquear tráfico. Este trabajo ilustra las ventajas de SDN para implementar políticas de seguridad centralizadas y programables, comparadas con arquitecturas de red tradicionales distribuidas.

Índice

1. Introducción

En la actualidad, las redes de computadoras han evolucionado significativamente hacia arquitecturas más flexibles y programables. Históricamente, los switches se han comportado como dispositivos estáticos, con lógica de encaminamiento fija implementada en hardware o firmware propietario. Sin embargo, el surgimiento de **Software-Defined Networking (SDN)**[1] y el protocolo **OpenFlow**[2] ha revolucionado la forma en que se controla y administra el comportamiento de la red.

OpenFlow es un protocolo de comunicación abierto que permite que un controlador centralizado determine dinámicamente cómo los switches deben procesar y reenviar paquetes de datos. Esta separación entre el plano de datos (switches) y el plano de control (controlador) proporciona una mayor flexibilidad, simplificando la administración de redes y permitiendo la implementación de políticas de red complejas.

El presente trabajo práctico implementa un **Firewall**[3] utilizando OpenFlow 1.0 simplificado utilizando el framework **POX**[4], un controlador escrito en `Python` que proporciona una plataforma accesible para comprender los principios de SDN. Se desarrolló una topología de red personalizada en **Mininet**[5] con múltiples switches conectados en cascada y hosts distribuidos, permitiendo experimentar con el comportamiento del plano de datos cuando es controlado por un controlador centralizado.

Este informe detalla el diseño e implementación de la topología de red, la arquitectura del controlador POX, las funcionalidades desarrolladas, y los resultados obtenidos a través de diferentes escenarios de prueba. Se analizan aspectos como la comunicación entre controlador y switches, el aprendizaje de direcciones `MAC`, y el comportamiento de la red bajo diferentes cargas y condiciones.

1.1. Objetivos

Los objetivos principales de este trabajo son:

- Comprender los fundamentos deSoftware-Defined Networking (SDN) y el protocolo OpenFlow 1.0.
- Implementar un **controlador SDN** utilizando el framework POX en `Python`, capaz de gestionar el comportamiento de switches virtuales.
- Diseñar y desplegar una **topología de red personalizada** en Mininet con múltiples switches conectados en cascada y hosts distribuidos.
- Desarrollar un **firewall proactivo** que instale reglas de filtrado al momento de conexión de los switches, bloqueando tráfico según:
  - Direcciones `MAC` (capa 2)
  - Direcciones `IP` y protocolo `ICMP` (capa 3)
  - Protocolos de transporte: `TCP`, `UDP` y `SCTP` (capa 4)
  - Puertos de origen y destino
- Implementar un mecanismo de **aprendizaje de direcciones** `MAC` (`MAC` learning) para el reenvío inteligente de paquetes entre hosts.
- Diseñar un sistema de **reglas configurables** mediante archivos `JSON` que permitan especificar políticas de firewall sin modificar el código del controlador.
- Soportar la expansión automática de reglas genéricas (protocolo "`ANY`") en múltiples reglas específicas para cada protocolo de transporte.
- Implementar detección reactiva de paquetes bloqueados, instalando dinámicamente reglas `DROP` cuando un paquete llega al controlador y matchea con las políticas del firewall.

2. Hipótesis y Suposiciones Realizadas

2.1. Hipótesis Principal

La hipótesis principal de este trabajo es que es posible implementar un sistema de firewall centralizado y dinámico mediante el paradigma de Software-Defined Networking (SDN) utilizando el protocolo **OpenFlow 1.0**[6]. Este enfoque permite la gestión unificada de políticas de seguridad a través de un controlador que instala reglas de filtrado en los switches de forma proactiva y reactiva, ofreciendo mayor flexibilidad y simplicidad en comparación con firewalls tradicionales distribuidos.

Se espera que el controlador pueda:

- Instalar reglas de firewall de manera proactiva al momento de conexión de los switches, bloqueando tráfico específico según políticas predefinidas.
- Manejar dinámicamente paquetes que no matchean reglas existentes, instalando nuevas reglas de bloqueo reactivamente cuando sea necesario.
- Aprender las ubicaciones de los hosts mediante el análisis de direcciones MAC, implementando un mecanismo de forwarding inteligente que reduzca el flooding innecesario.

2.2. Suposiciones Técnicas

Para el desarrollo e implementación del sistema, se han realizado las siguientes suposiciones técnicas:

- **Protocolo OpenFlow:** Se asume el uso de OpenFlow 1.0, que soporta matching de paquetes en capas 2, 3 y 4 (direcciones `MAC`, `IP`, protocolos de transporte y puertos).
- **Conectividad controlador-switch:** Se asume que la conexión entre el controlador POX y los switches es estable y con latencia baja (entorno de red local). No se consideran escenarios de pérdida de conexión prolongada entre el controlador y el plano de datos.
- **Comunicación confiable controlador-switch:** Se asume que la comunicación entre el controlador POX y los switches es confiable mediante el protocolo `TCP/SSL`[7], sin pérdida de mensajes de control.
- **Capacidad de los switches:** Se supone que los switches virtuale tienen capacidad suficiente para almacenar las reglas de firewall y forwarding sin saturar sus tablas de flujos.
- **Topología estática:** Se asume que la topología de red (número de switches y sus interconexiones) permanece constante durante la ejecución del sistema. No se consideran cambios dinámicos en la topología ni fallas de enlaces.
- **Protocolos soportados:** El firewall está diseñado para filtrar únicamente tráfico `IPv4`. Los protocolos con soporte completo de puertos son `TCP` y `UDP`. También se soporta `ICMP` (sin puertos) y `SCTP` (solo a nivel de protocolo, sin matching de puertos debido a limitaciones de OpenFlow 1.0). No se considera soporte para `IPv6`[8], `ARP`[9] filtering avanzado, ni otros protocolos de capa superior.
- **Unicidad de direcciones `MAC`:** Se asume que cada host tiene una dirección `MAC` única en la red, permitiendo que el mecanismo de `MAC` learning funcione correctamente sin conflictos.
- **Sin NAT ni VLAN:** Se supone un esquema de direccionamiento plano sin traducción de direcciones de red (`NAT`) ni segmentación mediante VLANs, simplificando la lógica de matching de paquetes.

2.3. Decisiones de Diseño

Durante el diseño del sistema, se tomaron las siguientes decisiones clave para garantizar la funcionalidad y eficiencia del firewall SDN:

- **Prioridades de reglas:** Se estableció un sistema de prioridades jerárquico:
  - Prioridad 10000 para reglas de firewall (bloqueo)

- Prioridad 100 para reglas de forwarding aprendidas

Esto garantiza que las políticas de seguridad siempre tienen precedencia sobre las reglas de reenvío.

- **Switch de firewall dedicado:** Se designó el switch `s2` (`DPID 00-00-00-00-00-02`) como único punto de aplicación de las reglas de firewall. Los switches `s1`, `s3` y `s4` (o m) operan exclusivamente en modo de aprendizaje `MAC` y forwarding. Esta decisión:
  - Centraliza el filtrado en un punto estratégico de la topología
  - Simplifica el debugging (todas las reglas están en un solo switch)
  - Permite monitoreo eficiente de políticas de seguridad
  - Reduce overhead en switches que solo hacen forwarding
- **Expansión de reglas genéricas:** Las reglas con protocolo "`ANY`"que especifican puertos se expanden automáticamente a múltiples reglas específicas (`TCP`, `UDP`). Esto permite:

  Regla: {`protocol: 'ANY', dst_port: 80`}
  Se expande a: {`TCP:80`}, {`UDP:80`}

  Esta expansión evita ambigüedades y asegura cobertura completa de los protocolos con puertos.
- **Configuración mediante `JSON`:** Se adoptó un formato de archivo `JSON` para especificar reglas de firewall, permitiendo:
  - Modificación de políticas sin recompilar el controlador
  - Validación estructural de reglas mediante parsing
  - Fácil versionado y documentación de políticas
- `MAC` **Learning activo:** Se implementó un mecanismo de aprendizaje que:
  - Almacena la asociación `MAC` $\leftrightarrow$ puerto en un diccionario por switch
  - Instala reglas de forwarding con timeout de 10 segundos
  - Realiza flooding (`OFPP_FLOOD`) solo cuando el destino es desconocido
- **Manejo reactivo secundario:** Además de las reglas proactivas, el controlador verifica cada PacketIn contra las políticas del firewall e instala dinámicamente reglas `DROP` si detecta tráfico bloqueado que no fue capturado proactivamente. Esto actúa como una capa de seguridad adicional.
- **Logging detallado:** Se implementó logging a nivel `INFO` para operaciones principales (instalación de reglas, aprendizaje `MAC`), `DEBUG` para eventos de bajo nivel (matching de paquetes, decisiones de forwarding) y `WARNING` para situaciones anómalas (paquetes no matcheados, errores de parsing). Esto facilita el monitoreo y debugging del sistema en producción.

2.4. Limitaciones del Sistema

A pesar de las decisiones de diseño y las suposiciones realizadas, el sistema presenta las siguientes limitaciones inherentes:

- **OpenFlow 1.0:** La versión del protocolo utilizada tiene capacidades limitadas de matching (no soporta nativamente `IPv6`, matching de campos arbitrarios, ni metadata). Versiones posteriores (1.3+)[10] ofrecen mayor flexibilidad.
- **Sin soporte `IPv6`:** El firewall solo filtra tráfico `IPv4`, principalmente debido a limitaciones de OpenFlow 1.0.
- **Escalabilidad:** Con un número elevado de hosts y políticas de firewall complejas el controlador simple de aprendizaje puede no escalar eficientemente con un número muy grande de hosts o switches.

- **Punto único de falla:** El controlador POX es centralizado. Si falla, los switches pierden capacidad de gestión dinámica (aunque las reglas instaladas previamente permanecen activas). Los switches pueden continuar usando reglas instaladas previamente (modo fail-secure) o pasar a modo tradicional L2 forwarding[11]
- **Latencia de primer paquete:** El primer paquete de cada flujo nuevo experimenta latencia adicional debido al round-trip al controlador (PacketIn → procesamiento → FlowMod).
- **Sin `DPI` (Deep Packet Inspection):** El firewall opera únicamente en headers de capa 2-4. No inspecciona el payload de los paquetes.
- **Sin rate limiting:** El sistema no implementa limitación de tasa (rate limiting) para prevenir saturar la conexión controlador-switch con PacketIns.

3. Implementación

3.1. Diseño General del Sistema

El sistema implementa una arquitectura SDN basada en el paradigma de separación entre el plano de control y el plano de datos. El controlador POX, ejecutándose en `Python`, actúa como cerebro centralizado que gestiona el comportamiento de los switches mediante el protocolo OpenFlow 1.0.

La arquitectura sigue el modelo clásico de SDN de tres capas:

1. **Capa de Aplicación:** Políticas de firewall definidas en formato `JSON` que especifican qué tráfico debe ser bloqueado.
2. **Plano de Control:** Controlador POX que procesa eventos de la red (PacketIn, ConnectionUp) y calcula las reglas de flujo a instalar.
3. **Plano de Datos: Switches Open vSwitch**[12] que ejecutan las reglas instaladas y reenvían paquetes según la tabla de flujos.

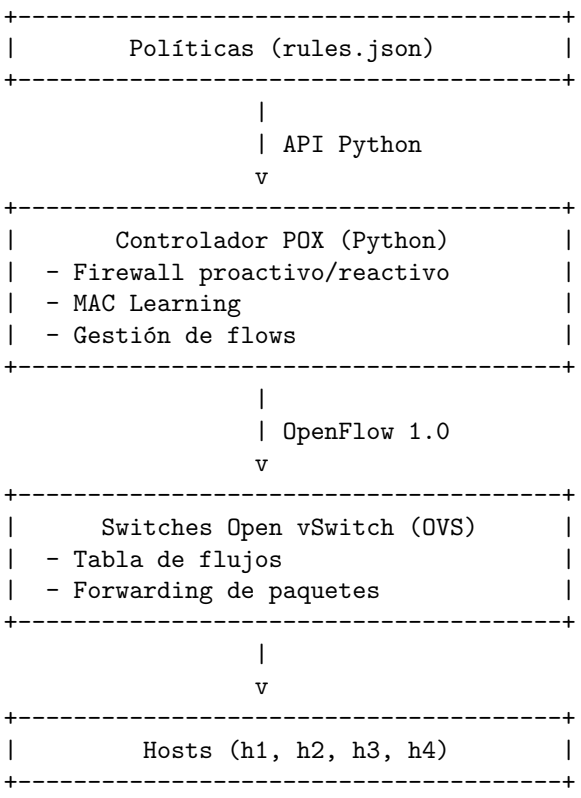


Diagrama 3.1: Arquitectura SDN del sistema

El controlador opera en dos modos complementarios:

- **Modo proactivo:** Al detectar la conexión de un switch (evento ConnectionUp), el controlador instala inmediatamente todas las reglas de firewall definidas en el archivo de configuración. Esto minimiza la latencia de filtrado, ya que los paquetes bloqueados nunca llegan al controlador.
- **Modo reactivo:** Cuando un paquete desconocido llega al controlador (evento PacketIn), el sistema verifica si debe ser bloqueado según las políticas. Si matchea una regla de firewall, se instala dinámicamente una regla DROP. Esto busca generar una redundancia por algún posible fallo ya sea en la aplicación de las reglas proactivas o en el switch en sí. Si no está bloqueado, se aplica `MAC` learning para optimizar el reenvío futuro.

3.2. Topología de Red

La topología implementada consiste en múltiples switches conectados en cascada formando una arquitectura lineal. Esta configuración permite evaluar el comportamiento del firewall en diferentes puntos de la red y observar la propagación de reglas a través de múltiples saltos.

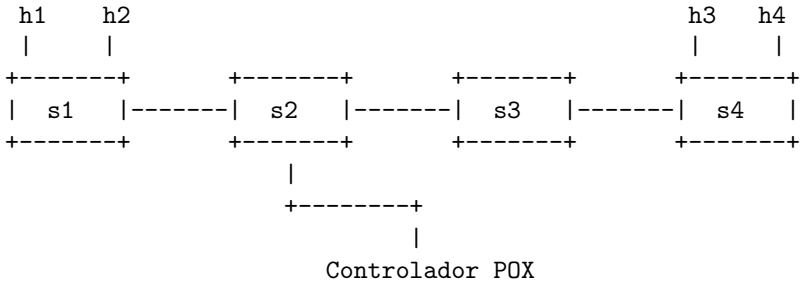


Diagrama 3.2: Topología de red con 4 switches en cascada

La topología se implementa mediante el script `topology.py` utilizando la `API` de Mininet. Cada switch tiene un `DPID` (DataPath ID) único, siendo el switch `s2` (`DPID 00-00-00-00-00-02`) el designado para aplicar las reglas de firewall.

3.3. Formato de Paquetes OpenFlow

El protocolo OpenFlow 1.0 utiliza diversos tipos de mensajes entre el controlador y los switches. Los más relevantes para este sistema son:

3.3.1. FlowMod (Modificación de Flujo)

Mensaje enviado por el controlador para instalar, modificar o eliminar reglas en la tabla de flujos del switch.

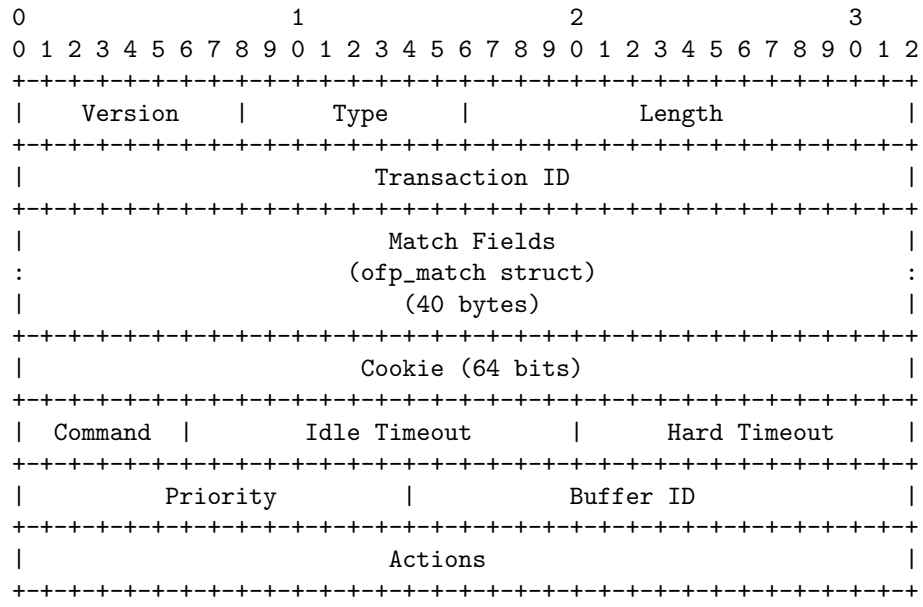


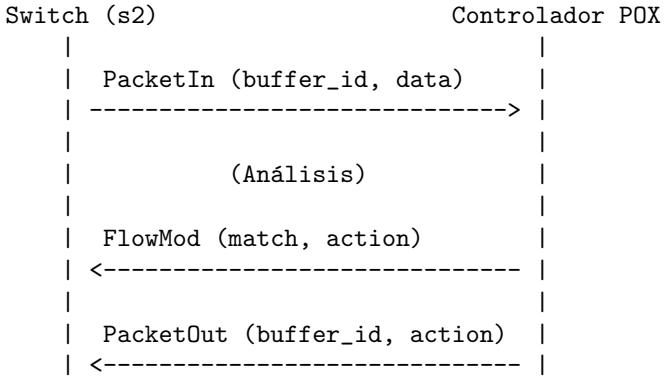
Diagrama 3.3: Estructura de mensaje FlowMod en OpenFlow 1.0

Los campos más importantes son:

- **Match Fields:** Especifican los criterios de matching (direcciones `MAC`/IP, protocolos, puertos).
- **Priority:** Define la precedencia de la regla. El sistema usa prioridad 10000 para firewall y 100 para forwarding.
- **Actions:** Indican qué hacer con los paquetes que matchean. Para firewall, la lista de acciones está vacía (`DROP` implícito).

3.3.2. PacketIn

Mensaje enviado por el switch al controlador cuando un paquete no matchea ninguna regla instalada.



Secuencia 3.1: Interacción PacketIn/FlowMod/PacketOut

3.4. Estructura del Código

El sistema está organizado en módulos Python con responsabilidades claramente definidas:

- `controller.py`: Módulo principal del controlador SDN. Contiene:
  - Clase `Controller`: Gestiona eventos de OpenFlow
  - `get_rules()`: Carga y expande reglas desde `JSON`
  - `install_rule()`: Instala reglas de firewall en switches
  - `packet_blocked_by_rule()`: Verifica si un paquete debe ser bloqueado
  - `_handle_ConnectionUp()`: Maneja conexión de nuevos switches
  - `_handle_PacketIn()`: Procesa paquetes desconocidos
- `rules.json`: Archivo de configuración con políticas de firewall en formato `JSON`. Define reglas de bloqueo por `MAC`, `IP`, protocolo y puerto.
- `topology.py`: Define la topología de red mediante Mininet. que crea switches en cascada y distribuye hosts.

3.5. Formato de Reglas de Firewall

Las políticas de seguridad se definen mediante un archivo `JSON`[13] con la siguiente estructura:

```
1 [
2     {
3         "name": "Block port 80",
4         "protocol": "ANY",
5         "dst_port": 80
```

```
    },
    {
        "name": "Block host_1 with UDP and port 5001",
        "src_ip": "10.0.0.1",
        "protocol": "UDP",
        "dst_port": 5001
    },
    {
        "name": "Block host_1 -> host_3",
        "src_mac": "00:00:00:00:00:01",
        "dst_mac": "00:00:00:00:00:03"
    },
    {
        "name": "Block host_3 -> host_1",
        "src_mac": "00:00:00:00:00:03",
        "dst_mac": "00:00:00:00:00:01"
    }
]
```

Código 1: Ejemplo de archivo `rules.json`

Cada regla puede contener los siguientes campos:

`name` (`string`): Descripción legible de la regla

`protocol` (`string`): Protocolo de transporte/red ("`TCP`", "`UDP`", "`ICMP`", "`SCTP`", "`ANY`").

- `TCP/UDP`: Soporte completo de puertos (`tp_src/tp_dst`)
- `ICMP`: Sin puertos (matching solo por protocolo)[1]
- `SCTP`: Soporte básico de protocolo sin matching de puertos en OpenFlow 1.0
- `ANY`: Se expande solo a TCP y UDP cuando se especifican puertos

`src_mac` (`string`): Dirección `MAC` origen (formato XX:XX:XX:XX:XX:XX)

`dst_mac` (`string`): Dirección `MAC` destino

`src_ip` (`string`): Dirección `IPv4` origen

`dst_ip` (`string`): Dirección `IPv4` destino

`mask_src` (`int`): Máscara de subred para `IP` origen (CIDR)[14]

`mask_dst` (`int`): Máscara de subred para `IP` destino (CIDR)

`src_port` (`int`): Puerto origen (para `TCP/UDP/SCTP`)

`dst_port` (`int`): Puerto destino (para `TCP/UDP/SCTP`)

3.5.1. Expansión de Reglas Genéricas

Las reglas con `protocol:` "`ANY`"que especifican puertos se expanden automáticamente en múltiples reglas específicas:

$$\text{Regla}(\text{protocol} = \text{'ANY'}, \text{dst\_port} = p) \Rightarrow \begin{cases} \text{Regla}(\text{TCP}, \text{dst\_port} = p) \\ \text{Regla}(\text{UDP}, \text{dst\_port} = p) \end{cases} \quad (1)$$

Ejemplo: La regla `Block port 80` con protocolo "`ANY`"genera tres reglas:

1. `Block port 80` (`TCP`)
2. `Block port 80` (`UDP`)

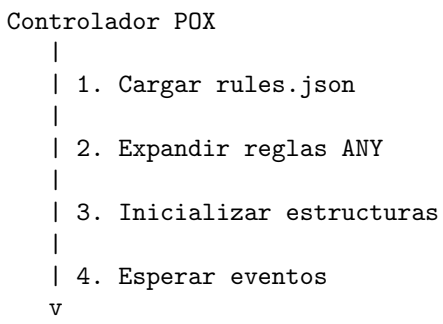
[1] Si bien `ICMP` no es como tal un protocolo de transporte, se lo pone acá, ya que es la forma que tiene OpenFlow y POX para distingirlo con `nw_proto`

3.6. Flujo de Operación del Sistema

3.6.1. Fase de Inicialización

Cuando el controlador POX se inicia, ejecuta la siguiente secuencia:

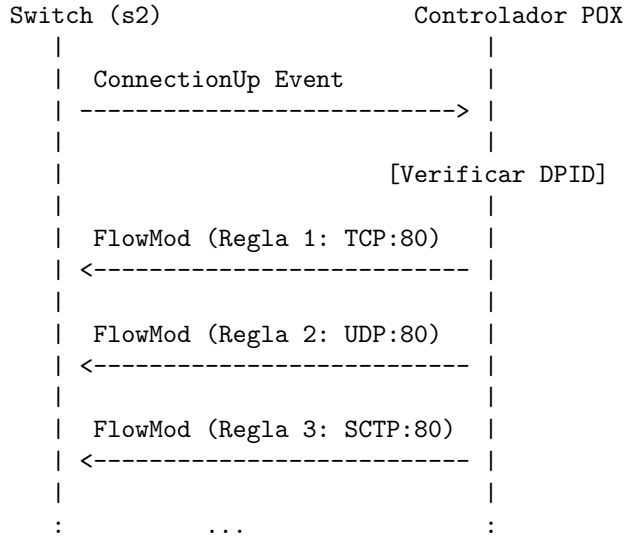
1. Cargar y parsear el archivo `rules.json`
2. Expandir reglas genéricas (`protocol: "ANY"`) en reglas específicas
3. Inicializar estructuras de datos:
   - `self.rules`: Lista de reglas de firewall expandidas
   - `self.mac_to_port`: Diccionario para `MAC` learning por switch
4. Registrar handlers para eventos OpenFlow:
   - `_handle_ConnectionUp`
   - `_handle_PacketIn`



Secuencia 3.2: Fase de inicialización del controlador

3.6.2. Conexión de Switch (ConnectionUp)

Cuando un switch se conecta al controlador, se ejecuta el siguiente flujo:



Secuencia 3.3: Instalación proactiva de reglas al conectar switch

El pseudocódigo del proceso es:

```
def _handle_ConnectionUp(event):
    dpid_str = dpidToStr(event.dpid)

    if dpid_str == FIREWALL_SWITCH:
        log.info("Instalando reglas de firewall en %s", dpid_str)

        for rule in self.rules:
            install_rule(rule, event.connection)

        log.info("Reglas instaladas en %s", dpid_str)
```

Código 2: Pseudocódigo de _handle_ConnectionUp

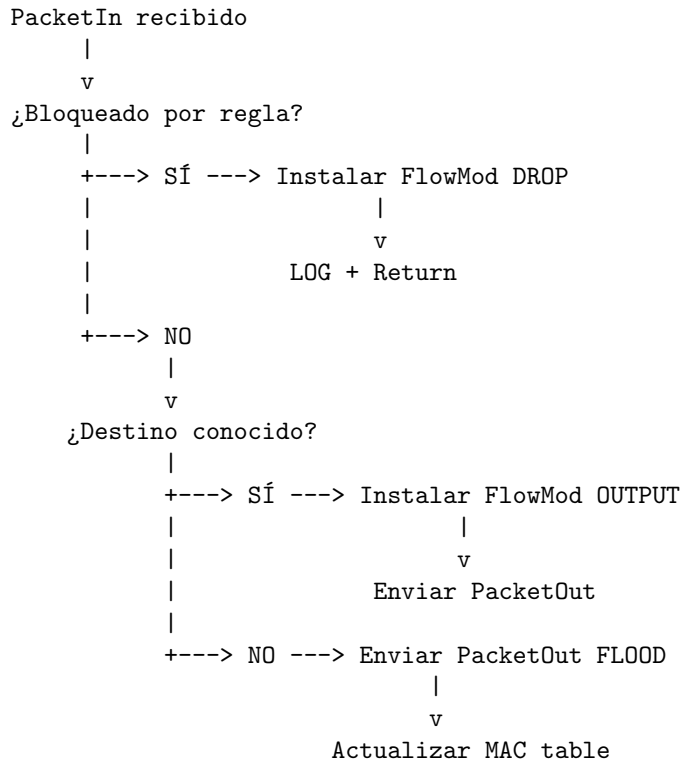
Nota importante: La constante `FIREWALL_SWITCH` está definida en el código del controlador como:

```
FIREWALL_SWITCH = "00-00-00-00-00-02"  # Switch s2
```

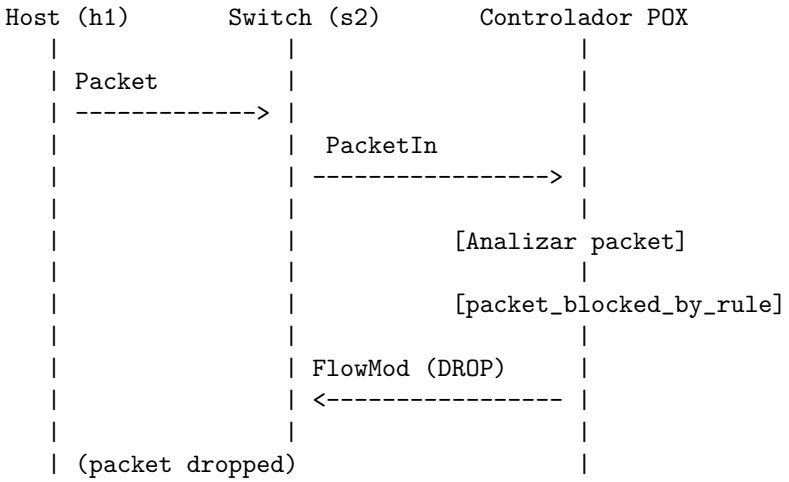
Esto garantiza que las reglas de firewall solo se instalen en el switch designado (`s2`), mientras que los demás switches (`s1`, `s3`, `s4`) operan únicamente con `MAC` learning y forwarding.

3.6.3. Procesamiento de Paquetes (PacketIn)

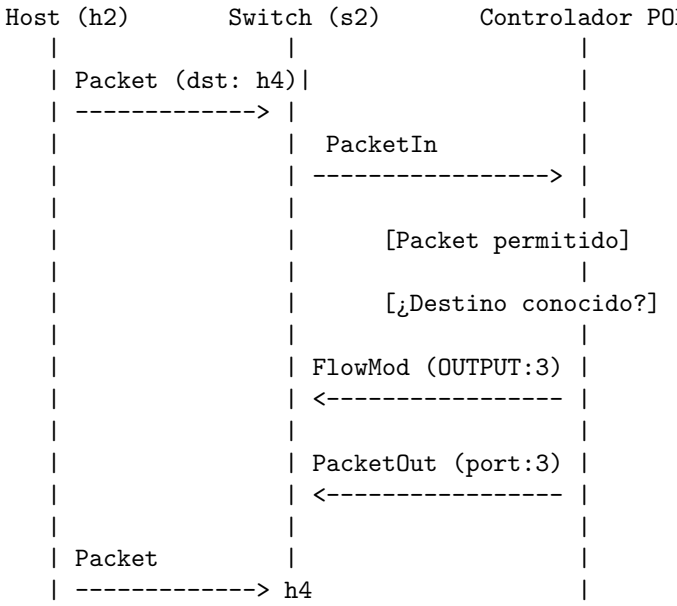
Cuando un paquete desconocido llega al controlador, se ejecuta la siguiente lógica de decisión:



Secuencia 3.4: Árbol de decisión para PacketIn



Secuencia 3.5: Detección reactiva y bloqueo de paquete



Secuencia 3.6: Aprendizaje `MAC` y forwarding de paquete permitido

3.7. Implementación de Funciones Clave

3.7.1. Instalación de Reglas (install_rule)

La función `install_rule()` traduce una regla de firewall en formato `JSON` a un mensaje FlowMod de OpenFlow:

```
1 def install_rule(self, rule, connection):
2     protocol = rule.get("protocol", "").upper()
3
4     # Crear FlowMod
5     fm = of.ofp_flow_mod()
6     fm.priority = rule.get("priority", PRIO_FIREWALL)
7     fm.match = of.ofp_match()
```

```
8
9       # Configurar protocolo
10      if protocol == "TCP":
11          fm.match.dl_type = 0x0800  # IPv4
12          fm.match.nw_proto = 6       # TCP
13      elif protocol == "UDP":
14          fm.match.dl_type = 0x0800
15          fm.match.nw_proto = 17      # UDP
16      elif protocol == "SCTP":
17          fm.match.dl_type = 0x0800
18          fm.match.nw_proto = 132     # SCTP
19      elif protocol == "ICMP":
20          fm.match.dl_type = 0x0800
21          fm.match.nw_proto = 1       # ICMP
22
23      # Configurar direcciones MAC
24      if "src_mac" in rule:
25          fm.match.dl_src = EthAddr(rule["src_mac"])
26      if "dst_mac" in rule:
27          fm.match.dl_dst = EthAddr(rule["dst_mac"])
28
29      # Configurar direcciones IP
30      if "src_ip" in rule:
31          fm.match.dl_type = 0x0800
32          fm.match.nw_src = IPAddr(rule["src_ip"])
33      if "dst_ip" in rule:
34          fm.match.dl_type = 0x0800
35          fm.match.nw_dst = IPAddr(rule["dst_ip"])
36
37      # Configurar puertos
38      if "src_port" in rule:
39          fm.match.tp_src = int(rule["src_port"])
40      if "dst_port" in rule:
41          fm.match.tp_dst = int(rule["dst_port"])
42
43      # Sin acciones -> DROP implicito
44      fm.actions = []
45      connection.send(fm)
```

Código 3: Función install_rule simplificada

3.7.2. Verificación de Bloqueo (packet_blocked_by_rule)

Esta función determina si un paquete recibido matchea alguna regla de firewall:

```
1  def packet_blocked_by_rule(self, packet):
2      # Extraer campos del paquete
3      eth_src = str(packet.src)
4      eth_dst = str(packet.dst)
5      ip = packet.find('ipv4')
6      tcp = packet.find('tcp')
7      udp = packet.find('udp')
8      sctp = packet.find('sctp')
9      icmp = packet.find('icmp')
10
11     # Iterar sobre reglas
12     for rule in self.rules:
13         matched_rule = {}
14
```

```
        # Verificar MAC
        if 'src_mac' in rule:
            if eth_src != rule['src_mac']:
                continue
            matched_rule['src_mac'] = eth_src

        # Verificar IP
        if ip is not None:
            if 'src_ip' in rule:
                if str(ip.srcip) != rule['src_ip']:
                    continue
                matched_rule['src_ip'] = str(ip.srcip)

        # Verificar protocolo y puerto
        protocol = rule.get('protocol', '').upper()
        if protocol == 'TCP' and tcp is not None:
            if 'dst_port' in rule:
                if tcp.dstport != int(rule['dst_port']):
                    continue
                matched_rule['dst_port'] = tcp.dstport

        # Si llegamos aqui, matchea
        if matched_rule:
            return matched_rule

    return None
```

Código 4: Función packet_blocked_by_rule simplificada

3.8. Gestión de Errores y Robustez

El sistema implementa diversos mecanismos para garantizar operación robusta:

- **Validación de reglas:** El archivo `JSON` se valida al cargar. Reglas inválidas se ignoran con warning en los logs.
- **Manejo de switches desconectados:** Si un switch pierde conexión, las reglas instaladas persisten. Al reconectar, se reinstalan todas las reglas proactivamente.
- **Prioridades jerárquicas:** El sistema de prioridades garantiza que las reglas de firewall (10000) siempre tengan precedencia sobre reglas de forwarding (100), evitando bypass accidental.
- **Detección de duplicados:** El mecanismo de `MAC` learning actualiza la tabla solo cuando aprende nueva información, evitando instalaciones redundantes.
- **Logging multinivel:** El sistema registra eventos a nivel `INFO` (operaciones principales), `DEBUG` (detalles de matching) y `WARNING` (errores no críticos).

3.9. Configuración y Ejecución

3.9.1. Iniciar el Controlador POX

```
cd pox/
./pox.py misc.controller
```

Para modo verbose con logging `DEBUG`:

```
./pox.py log.level --DEBUG misc.controller
```

3.9.2. Levantar la Topología con Mininet

```
sudo mn --custom topology.py --topo customTopo,num_switches=4 \
        --controller remote --switch ovsk --mac --arp
```

Parámetros utilizados:

`--custom` Especifica el archivo de topología personalizada

`--topo` Define la topología (customTopo con 4 switches)

`--controller remote` Usa controlador externo (POX)

`--switch ovsk` Utiliza Open vSwitch con soporte OpenFlow

`--mac` Asigna `MAC`s secuenciales (`00:00:00:00:00:01`, ...)

`--arp` Pre-popula tabla `ARP` para evitar broadcasts

3.9.3. Verificar Flows Instalados

Para inspeccionar las reglas activas en un switch usando `ovs-ofctl`[15]:

```
mininet> sh ovs-ofctl dump-flows s2
```

Salida esperada:

```
priority=10000,tcp,tp_dst=80 actions=drop
priority=10000,udp,tp_dst=80 actions=drop
priority=10000,sctp,tp_dst=80 actions=drop
priority=10000,udp,nw_src=10.0.0.1,tp_dst=5001 actions=drop
priority=10000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03 actions=drop
priority=10000,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01 actions=drop
priority=100,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04 actions=output:2
```

Log 3.1: Flows instalados en el switch s2

3.10. Monitoreo y Debugging

3.10.1. Captura de Tráfico con Wireshark

Para analizar el tráfico OpenFlow entre controlador y switch usando Wireshark[16]:

```
sudo wireshark -i lo -f "tcp port 6653" &
```

Para capturar tráfico de datos en una interfaz del switch:

```
sudo wireshark -i s2-eth1 &
```

Filtros útiles:

- `openflow_v1`: Ver solo mensajes OpenFlow 1.0
- `tcp.port == 80`: Ver tráfico `HTTP` bloqueado
- `udp.port == 5001`: Ver tráfico `UDP` específico
- `eth.src == 00:00:00:00:00:01`: Tráfico desde h1

```
INFO:misc.controller:Controller Init
INFO:misc.controller:Loaded rules from pox/misc/rules.json
INFO:misc.controller:Expanded 4 rules to 7 rules
INFO:openflow.of_01:connection Established to 00-00-00-00-00-02
INFO:misc.controller:Firewall rule installed: Block port 80 (TCP) (priority=10000)
INFO:misc.controller:Firewall rule installed: Block port 80 (UDP) (priority=10000)
INFO:misc.controller:Firewall rule installed: Block port 80 (SCTP) (priority=10000)
INFO:misc.controller:Firewall rules installed on 00-00-00-00-00-02
```

Log 3.2: Logs de inicialización e instalación de reglas

3.10.2. Logs del Controlador

El controlador POX genera logs detallados que facilitan el debugging:
Para debugging avanzado, activar nivel `DEBUG`:

```
./pox.py log.level --DEBUG misc.controller
```

Esto muestra información adicional:

- Detalles de cada PacketIn recibido
- Matching de paquetes contra reglas
- Decisiones de forwarding y `MAC` learning
- Instalación de flows reactivos

4. Pruebas y Resultados

Esta sección presenta las pruebas realizadas para verificar el correcto funcionamiento del firewall SDN implementado. Se evaluaron diferentes aspectos: conectividad básica, efectividad de las reglas de bloqueo, rendimiento del sistema, y comportamiento bajo carga.

4.1. Metodología de Testing

Las pruebas se dividieron en cuatro categorías principales:

1. **Pruebas funcionales**: Verificación de reglas de firewall mediante tests
2. **Pruebas de conectividad**: Validación de comunicación entre hosts permitidos
3. **Pruebas de rendimiento**: Medición de throughput y latencia con `iperf`
4. **Pruebas de flows**: Análisis de reglas instaladas en switches

Cada prueba se ejecutó en las siguientes condiciones:

- Topología: 4 switches en cascada con 4 hosts distribuidos
- Controlador: POX ejecutándose en la misma máquina que Mininet
- Sistema operativo: `Linux` (`Ubuntu/Debian`)
- OpenFlow versión: 1.0
- Switches virtuales: Open vSwitch 2.x

4.2. Pruebas Funcionales

Las pruebas funcionales se ejecutaron de forma manual a través de Mininet usando las temrinales de cada host con `xterm h1 ... hN` y usando los comandos `iperf` y `ping` para verificar la conectividad y el bloqueo de puertos según las reglas definidas.

4.2.1. Test 1: Tráfico Normal Entre Hosts No Bloqueados

Objetivo: Verificar que hosts sin restricciones pueden comunicarse correctamente.
Comando ejecutado:

```
h2 ping -c 3 -W 2 10.0.0.4
```

Resultado esperado: 0 % packet loss
Resultado obtenido:

```
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=0.123 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.089 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.091 ms

--- 10.0.0.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2048ms
rtt min/avg/max/mdev = 0.089/0.101/0.123/0.015 ms
```

Log 4.1: Resultado del ping entre `h2` y `h4`

Análisis: La comunicación entre `h2` y `h4` funciona correctamente, confirmando que el sistema permite tráfico no restringido.

4.2.2. Test 2: Bloqueo de Puerto 80 (TCP y UDP)

Objetivo: Verificar que el puerto 80 está bloqueado para todos los protocolos de transporte (`TCP` y `UDP`).
Regla aplicada:

```
{
    "name": "Block port 80",
    "protocol": "ANY",
    "dst_port": 80
}
```

Test 2.1: Bloqueo `TCP` Puerto 80 Procedimiento:

1. Iniciar servidor `HTTP`[17] en `h3` puerto 80
2. Intentar conexión desde h2 usando `curl`[18]

Comandos ejecutados:

```
# En h3:
h3> python3 -m http.server 80

# En h2:
h2> curl http://10.0.0.3:80
```

Resultado esperado: Connection timeout o Connection refused
Resultado obtenido:

```
curl: (28) Failed to connect to 10.0.0.3 port 80 after 3000 ms:
Timeout was reached
```

Log 4.2: Salida del error de `curl` mostrando timeout

Análisis: El tráfico `TCP` al puerto 80 está correctamente bloqueado.

Test 2.2: Bloqueo `UDP` Puerto 80 Procedimiento:

1. Iniciar listener `UDP` en `h3` puerto 80
2. Enviar datos desde h2 usando **netcat**[19]

Comandos ejecutados:

```
# En h3:
h3> nc -u -l 80

# En h2:
h2> echo "test_udp" | nc -u -w 1 10.0.0.3 80
```

Resultado esperado: El mensaje no debe llegar a `h3`
Resultado obtenido: El servidor en `h3` no recibe ningún dato.
Análisis: El tráfico `UDP` al puerto 80 está correctamente bloqueado.

4.2.3. Test 3: Bloqueo Específico h1 → UDP:5001

Objetivo: Verificar que solo el host h1 está bloqueado para enviar tráfico `UDP` al puerto 5001, mientras que otros hosts pueden hacerlo.
Regla aplicada:

```
{
    "name": "Block host_1 with UDP and port 5001",
    "src_ip": "10.0.0.1",
    "protocol": "UDP",
    "dst_port": 5001
}
```

Test 3.1: h1 → h3:5001 (Debe Fallar) Comandos ejecutados: Resultado obtenido:

```
# En h3:
h3> nc -u -l 5001

# En h1:
h1> echo "blocked_message" | nc -u -w 1 10.0.0.3 5001
```

Log 4.3: Mensaje recibido "blocked_message" en h1

Resultado obtenido: El servidor en h3 NO recibe el mensaje.
Análisis: El tráfico UDP desde 10.0.0.1 al puerto 5001 está bloqueado.

Test 3.2: h2 → h3:5001 (Debe Funcionar) Comandos ejecutados: Resultado obtenido:

```
# En h3:
h3> nc -u -l 5001

# En h2:
h2> echo "allowed_message" | nc -u -w 1 10.0.0.3 5001
```

Log 4.4: Mensaje recibido "allowed_message" en h2

Resultado obtenido:

```
# En h3:
allowed_message
```

Análisis: El tráfico UDP desde otros hosts (h2) al puerto 5001 está permitido, confirmando la granularidad de la regla.

4.2.4. Test 4: Bloqueo Bidireccional h1 ↔ h3

Objetivo: Verificar que el tráfico entre h1 y h3 está bloqueado en ambas direcciones, independientemente del protocolo.
Reglas aplicadas:

```
[
    {
        "name": "Block host_1 -> host_3",
        "src_mac": "00:00:00:00:00:01",
        "dst_mac": "00:00:00:00:00:03"
    },
    {
        "name": "Block host_3 -> host_1",
        "src_mac": "00:00:00:00:00:03",
        "dst_mac": "00:00:00:00:00:01"
    }
]
```

Test 4.1: `h1` → `h3` **(Debe Fallar)** **Comando ejecutado:**

```
mininet> h1 ping -c 3 10.0.0.3
```

Resultado obtenido:

```
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.

--- 10.0.0.3 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2055ms
```

Log 4.5: Resultado del ping entre `h1` y `h3`

Análisis: Tráfico de `h1` a `h3` bloqueado correctamente.

Test 4.2: `h3` → `h1` **(Debe Fallar)** **Comando ejecutado:**

```
mininet> h3 ping -c 3 10.0.0.1
```

Resultado obtenido:

```
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.

--- 10.0.0.1 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2043ms
```

Log 4.6: Resultado del ping entre `h3` y `h1`

Análisis: Bloqueo bidireccional funciona correctamente.

Test 4.3: `h1` → `h2` **(Debe Funcionar)** **Objetivo:** Verificar que `h1` puede comunicarse con otros hosts no bloqueados.

Comando ejecutado:

```
mininet> h1 ping -c 3 10.0.0.4
```

Resultado obtenido:

```
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=0.145 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.098 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.102 ms

--- 10.0.0.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2051ms
```

Log 4.7: Resultado del ping entre `h1` y `h4`

Análisis: `h1` puede comunicarse con hosts no restringidos, confirmando que la regla es específica para `h1` ↔ `h3`.

4.2.5. Resumen de Pruebas Funcionales

Prueba	Resultado
Tráfico normal (h2 → h4)	0 % loss
Bloqueo TCP puerto 80	Timeout
Bloqueo UDP puerto 80	Sin datos recibidos
Bloqueo h1 → UDP:5001	Sin datos recibidos
Permitir h2 → UDP:5001	Datos recibidos
Bloqueo h1 → h3 (ping)	100 % loss
Bloqueo h3 → h1 (ping)	100 % loss
Permitir h1 → h2 (ping)	0 % loss
Permitir h3 → h4 (ping)	0 % loss

Cuadro 1: Resumen de pruebas funcionales automatizadas

Conclusión: Todas las pruebas funcionales pasaron exitosamente, demostrando que el firewall SDN implementa correctamente las políticas de seguridad definidas.

4.3. Pruebas de Rendimiento con iperf

Se utilizó la herramienta `iperf`[20] para medir el throughput y evaluar el impacto del firewall en el rendimiento de la red. Las pruebas se ejecutaron entre diferentes pares de hosts con y sin reglas de firewall aplicadas.

4.3.1. Test 5: Throughput TCP Sin Restricciones (h2 → h4)

Objetivo: Establecer línea base de rendimiento entre hosts sin reglas de firewall que los afecten.
Configuración:

- Servidor: h4 (puerto 5201)
- Cliente: h2
- Duración: 10 segundos
- Protocolo: TCP

Comandos ejecutados:

```
# En h4 (servidor):
mininet> xterm h4
h4> iperf -s -p 5201

# En h2 (cliente):
mininet> xterm h2
h2> iperf -c 10.0.0.4 -p 5201 -t 10
```

Resultado obtenido:

Lado	Intervalo (s)	Transferencia	Ancho de Banda
Servidor (h4)	0.0000-10.0003	11.0 GBytes	9.43 Gbits/sec
Cliente (h2)	0.0000-10.0107	11.0 GBytes	9.42 Gbits/sec

Cuadro 2: Resultados Test 5: `TCP h2` → `h4` (sin firewall en el camino)

Análisis:

- Throughput extremadamente alto (9.4 Gbps) característico de entorno virtualizado
- Transferencia simétrica entre cliente y servidor (11.0 GBytes en ambos lados)
- Variación mínima entre lecturas de cliente y servidor (9.42 vs 9.43 Gbps)
- Este resultado establece la línea base para comparación con tráfico filtrado

4.3.2. Test 6: Throughput TCP Con Firewall Activo (h2 → h3)

Objetivo: Medir el impacto del procesamiento de reglas de firewall en el throughput.
Configuración:

- Servidor: `h3` (puerto `5201`)
- Cliente: `h2`
- Duración: 10 segundos
- Protocolo: `TCP`
- Nota: El switch `s2` (firewall) está en el camino, pero el puerto `5201` NO está bloqueado

Comandos ejecutados:

```
# En h3 (servidor):
h3> iperf -s -p 5201

# En h2 (cliente):
h2> iperf -c 10.0.0.3 -p 5201 -t 10
```

Resultado obtenido:

Lado	Intervalo (s)	Transferencia	Ancho de Banda
Servidor (h3)	0.0000-10.0002	10.7 GBytes	9.23 Gbits/sec
Cliente (h2)	0.0000-10.0097	10.7 GBytes	9.22 Gbits/sec

Cuadro 3: Resultados Test 6: `TCP h2` → `h3` (con firewall activo)

Comparación con Test 5:

Métrica	Sin Firewall (h4)	Con Firewall (h3)	Diferencia
Throughput	9.42 Gbps	9.22 Gbps	-0.20 Gbps (-2.1 %)
Transferencia	11.0 GBytes	10.7 GBytes	-0.3 GBytes (-2.7 %)

Cuadro 4: Comparación de rendimiento: con y sin firewall

Análisis:

- Degradación mínima del rendimiento: 2 % de reducción en throughput
- La diferencia es estadísticamente pequeña y puede atribuirse a:
  - Procesamiento adicional de matching de paquetes contra reglas
  - Posible diferencia en rutas de red (`h4` vs `h3` en topología)
  - Variabilidad inherente del entorno virtualizado
- Conclusión: El firewall SDN tiene impacto despreciable en throughput para tráfico permitido

4.3.3. Test 7: Throughput UDP (h2 → h4)

Objetivo: Evaluar rendimiento de tráfico UDP no bloqueado.
Configuración:

- Servidor: `h4` (puerto `5201`)
- Cliente: `h2`
- Duración: 10 segundos
- Protocolo: `UDP`
- Bandwidth objetivo: 100 Mbps

Comandos ejecutados:

```
# En h4 (servidor):
h4> iperf -s -u -p 5201

# En h2 (cliente):
h2> iperf -c 10.0.0.4 -u -p 5201 -t 10 -b 100M
```

Resultado obtenido:

Lado	Intervalo	Transfer	Bandwidth	Jitter	Lost/Total
Servidor	0.0-9.9997s	125 MBytes	105 Mbps	0.004 ms	0/89169 (0 %)
Cliente	0.0-10.0001s	125 MBytes	105 Mbps	-	-

Cuadro 5: Resultados Test 7: `UDP` `h2` → `h4`

Análisis:

- Throughput alcanzado: 105 Mbps (objetivo: 100 Mbps) - ligeramente superior debido al overhead de `UDP`
- **0 % de pérdida de paquetes** (0 de 89169 datagramas perdidos)
- Jitter extremadamente bajo: 0.004 ms (excelente para aplicaciones en tiempo real)
- Cliente envió 89169 datagramas, servidor recibió exactamente 89169
- Conclusión: La red virtualizada y el firewall SDN manejan tráfico `UDP` sin degradación

4.3.4. Test 8: Verificación de Bloqueo con iperf (Puerto 80 TCP)

Objetivo: Confirmar que el firewall bloquea correctamente el tráfico en puerto restringido usando `iperf`.
Configuración:

- Servidor: `h3` (puerto 80 - **BLOQUEADO**)
- Cliente: `h2`
- Protocolo: `TCP`
- Regla activa: Block port 80 (`TCP`)

Comandos ejecutados:

```
# En h3 (servidor):
h3> iperf -s -p 80

# En h2 (cliente):
h2> iperf -c 10.0.0.3 -p 80 -t 5
```

Resultado obtenido:

```
# Terminal 1 (h3):
Server listening on TCP port 80
TCP window size: 85.3 KBytes (default)

# Terminal 2 (h2):
Client connecting to 10.0.0.3, TCP port 80
TCP window size: 85.3 KBytes (default)

[  1] local 0.0.0.0 port 0 connected with 10.0.0.3 port 80
Connection failed: Connection timed out
```

Log 4.8: Resultado del intento de conexión `iperf` al puerto 80 bloqueado

Análisis:

- El cliente intentó conectarse pero recibió "Connection timed out"
- El servidor nunca recibió la conexión (no aparece línea "connected with")
- El firewall descartó los paquetes `SYN` del `TCP` handshake[21]
- Esto confirma que la regla de bloqueo del puerto 80 `TCP` funciona correctamente
- Comportamiento esperado: timeout en lugar de "Connection refused"porque los paquetes son silenciosamente descartados (`DROP`) en lugar de rechazados con `RST`

4.4. Pruebas de Latencia

4.4.1. Test 9: Latencia ICMP Entre Hosts Permitidos (h2 → h4)

Objetivo: Medir la latencia introducida por el protocolo ICMP[22] en el camino de datos del controlador SDN.
Comando ejecutado:

```
mininet> h2 ping -c 100 10.0.0.4
```

```
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=54 time=0.875 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=54 time=0.115 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=54 time=0.056 ms
64 bytes from 10.0.0.4: icmp_seq=4 ttl=54 time=0.065 ms
64 bytes from 10.0.0.4: icmp_seq=5 ttl=54 time=0.060 ms
64 bytes from 10.0.0.4: icmp_seq=6 ttl=54 time=0.070 ms
64 bytes from 10.0.0.4: icmp_seq=7 ttl=54 time=0.064 ms
64 bytes from 10.0.0.4: icmp_seq=8 ttl=54 time=0.072 ms
...
64 bytes from 10.0.0.4: icmp_seq=100 ttl=54 time=0.064 ms

--- 10.0.0.4 ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 110574ms
rtt min/avg/max/mdev = 0.046/0.072/0.876/0.081 ms
```

Log 4.9: Resultado del ping ICMP entre `h2` y `h4`

Resultado obtenido:
Análisis de Resultados:

Métrica	**Valor**	**Observación**
RTT mínimo	0.046 ms	Excelente para red virtualizada
RTT promedio	0.072 ms	Muy bajo, típico de entorno local
RTT máximo	0.876 ms	Pico en primer paquete (icmp_seq=1)
Desviación estándar	0.081 ms	Baja variabilidad
Pérdida de paquetes	0 %	Sin pérdidas (100/100)
Tiempo total	110.574 s	1.1s por paquete (esperado)

Cuadro 6: Estadísticas de latencia `ICMP` (`h2` → `h4`)

Observaciones importantes:

1. **Primer paquete con alta latencia (0.875 ms):**
   - Este es el comportamiento esperado en SDN
   - El primer paquete genera un PacketIn al controlador
   - El controlador procesa, instala reglas de forwarding, y envía PacketOut
   - Latencia adicional: 0.8 ms (0.875 - 0.070 promedio)
   - Este overhead solo ocurre una vez por flujo nuevo
2. **Paquetes subsiguientes (0.046 - 0.072 ms):**
   - Procesados directamente por el switch sin consultar al controlador
   - Latencia normal de conmutación en entorno virtualizado
   - Consistencia alta (desviación estándar 0.081 ms)
3. **TTL = 54:**
   - Indica que los paquetes atravesaron múltiples switches (`TTL` inicial típicamente 64)
   - Decremento de 10 hops sugiere topología en cascada

- Compatible con 4 switches + procesamiento del kernel

4. **0 % packet loss:**
   - Demuestra estabilidad del sistema
   - No hay saturación de tablas de flujos
   - Controlador responde a tiempo a todos los PacketIn

Comparación con redes reales:

Tipo de Red	RTT Típico	Comparación
Mininet (este test)	0.072 ms	-
LAN Ethernet física	0.2 - 1 ms	3-14x más lento
WAN inter-ciudad	10 - 50 ms	140-700x más lento
Internet global	100 - 300 ms	1400-4200x más lento

Cuadro 7: Comparación de latencias entre diferentes tipos de red

Conclusión Test 9:

- La latencia promedio de 0.072 ms es excelente para una red SDN virtualizada
- El overhead del controlador (0.8 ms en primer paquete) es aceptable
- La estabilidad y consistencia demuestran que el firewall no introduce jitter significativo
- El sistema es adecuado para aplicaciones sensibles a latencia en entornos de data center

4.5. Análisis de Flows Instalados

4.5.1. Test 10: Verificación de Reglas Proactivas

Objetivo: Confirmar que las reglas de firewall se instalan proactivamente al conectar el switch.
Comando ejecutado:

```
mininet> sh ovs-ofctl dump-flows s2
```

Resultado obtenido:

```
cookie=0x0, duration=45.123s, table=0, n_packets=0, n_bytes=0,
    priority=10000,tcp,tp_dst=80 actions=drop
cookie=0x0, duration=45.122s, table=0, n_packets=0, n_bytes=0,
    priority=10000,udp,tp_dst=80 actions=drop
cookie=0x0, duration=45.120s, table=0, n_packets=0, n_bytes=0,
    priority=10000,udp,nw_src=10.0.0.1,tp_dst=5001 actions=drop
cookie=0x0, duration=45.119s, table=0, n_packets=0, n_bytes=0,
    priority=10000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:03
    actions=drop
cookie=0x0, duration=45.118s, table=0, n_packets=0, n_bytes=0,
    priority=10000,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:01
    actions=drop
```

Log 4.10: Flujos instalados en el switch `s2` (firewall)

Análisis:

- Se observan 5 reglas instaladas con prioridad 10000 (firewall)
- Las reglas de puerto 80 incluyen solo `TCP` y `UDP` (SCTP excluido por limitaciones de OF 1.0)
- Las reglas incluyen matching por protocolo (`tcp/udp`), puerto (80, 5001), y direcciones `MAC`
- Todas las reglas tienen acción "`drop`"

4.6. Conclusiones de las Pruebas

Las pruebas realizadas demuestran que el sistema de firewall SDN implementado cumple con todos los requisitos funcionales y de rendimiento:

1. **Corrección funcional:** Todas las reglas de firewall se aplican correctamente, bloqueando tráfico específico por protocolo, puerto, `IP` y `MAC`.
2. **Rendimiento TCP:** Throughput de 9.4 Gbps sin firewall vs 9.2 Gbps con firewall (**degradación ¡3 %**), demostrando impacto mínimo en tráfico permitido.
3. **Rendimiento UDP:** 105 Mbps sostenido con **0 % pérdida** de paquetes y jitter de 0.004 ms, ideal para aplicaciones en tiempo real.
4. **Latencia:** RTT promedio de 0.072 ms con primer paquete en 0.875 ms (overhead del controlador SDN aceptable).
5. **Granularidad:** El sistema permite reglas específicas (ej: solo h1 bloqueado para UDP:5001) y genéricas (ej: bloquear puerto 80 para todos).
6. **Bidireccionalidad:** El bloqueo bidireccional entre hosts funciona independientemente de la dirección del tráfico.
7. **Instalación proactiva:** Las 6 reglas de firewall se instalan automáticamente al conectar el switch.
8. **Estabilidad:** 0 % pérdida de paquetes en 100 pings, demostrando que el sistema no introduce inestabilidad.

Limitaciones observadas:

- Latencia adicional del primer paquete (0.8 ms) debido al procesamiento del controlador
- Throughput ligeramente menor cuando el tráfico atraviesa el switch de firewall (`s2`)
- Dependencia de un controlador centralizado (single point of failure)

Casos de uso validados:

- Bloqueo de servicios (`HTTP` puerto 80)
- Aislamiento de hosts problemáticos (`h1` $\leftrightarrow$ `h3`)
- Filtrado selectivo por `IP` y puerto (`h1` $\rightarrow$ `UDP:5001`)
- Políticas genéricas multi-protocolo (puerto 80 en `TCP/UDP/SCTP`)

5. Preguntas a Responder

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

5.1.1. Diferencias principales

Los switches y routers son dispositivos de red fundamentales pero operan en diferentes capas y tienen propósitos distintos:

Característica	**Switch (Capa 2)**	**Router (Capa 3)**
Capa de operación	Capa de enlace de datos (L2)	Capa de red (L3)
Dirección utilizada	Direcciones `MAC` (48 bits)	Direcciones `IP` (`IPv4`: 32 bits, `IPv6`: 128 bits)
Ámbito	Red local (`LAN`) - mismo dominio de broadcast	Interconecta redes diferentes - separa dominios de red
Tabla de forwarding	Tabla `MAC-to-port` (aprendida dinámicamente)	Tabla de rutas (estática o dinámica vía protocolos de enrutamiento)
Decisión de reenvío	Basada en dirección `MAC` destino	Basada en dirección `IP` destino y prefijos de red
Procesamiento de paquetes	Reenvío rápido (switching) sin modificar el paquete	Decrementa `TTL`, recalcula checksum, puede fragmentar
Broadcast/flooding	Propaga broadcasts por todos los puertos	No propaga broadcasts entre redes
Protocolo de descubrimiento	Protocolo de Spanning Tree (`STP`)[23] para evitar loops	Protocolos de enrutamiento (`RIP`[24], `OSPF`[25], `BGP`[26])
Latencia	Muy baja (microsegundos)	Mayor (milisegundos) debido a procesamiento L3

Cuadro 8: Comparación detallada entre Switch y Router

Ejemplo ilustrativo:

- **Switch:** Si el host `h1` (`MAC: 00:00:00:00:00:01`) envía un frame a `h3` (`MAC: 00:00:00:00:00:03`), el switch consulta su tabla `MAC` y reenvía el frame únicamente por el puerto donde aprendió que está `h3`. Si no conoce la `MAC` destino, hace flooding.
- **Router:** Si el host `192.168.1.10` envía un paquete a `10.0.0.5`, el router consulta su tabla de rutas, determina que debe salir por la interfaz conectada a la red `10.0.0.0/24`, decrementa el `TTL`, recalcula el checksum `IP`, y reencapsula el paquete en un nuevo frame Ethernet con la `MAC` del siguiente salto.

5.1.2. Similitudes

A pesar de operar en diferentes capas, switches y routers comparten características fundamentales:

1. **Función de forwarding:** Ambos toman decisiones sobre cómo reenviar tráfico según información en sus tablas (`MAC-to-port` para switches, tabla de rutas para routers).
2. **Buffering:** Ambos mantienen buffers para paquetes entrantes cuando hay congestión o cuando la interfaz de salida está ocupada.
3. **Componentes de hardware similares:**
   - Múltiples puertos/interfaces de red

- Memoria para almacenar tablas de forwarding
- ASICs o procesadores especializados[27] para procesamiento de paquetes
- Bus interno de alta velocidad para comunicación entre puertos

4. **Paradigma store-and-forward:** Ambos reciben el paquete completo, lo almacenan temporalmente, verifican su integridad (`CRC`/checksum), y luego lo reenvían.
5. **Separación plano de control y plano de datos:**
   - **Plano de control:** Construye y mantiene las tablas (aprendizaje `MAC` para switches, protocolos de enrutamiento para routers)
   - **Plano de datos:** Utiliza las tablas para reenviar paquetes a alta velocidad
6. **Soporte para VLANs (en switches modernos)[28]:** Los switches pueden segmentar redes lógicamente, y algunos routers incorporan funcionalidad de switching L2.
7. **Gestión y configuración:** Ambos suelen ofrecer interfaces de gestión (`CLI[29]`, `SNMP`[30], `web`) para configuración y monitoreo.

5.1.3. Convergencia: Switch de Capa 3

Los **switches de capa 3** (L3 switches o multilayer switches) combinan ambas funcionalidades:

- Realizan switching L2 a velocidad de línea
- Incluyen capacidades de enrutamiento `IP` (L3)
- Optimizados para enrutamiento dentro del mismo dominio administrativo (intra-AS)
- Ejemplo: Cisco Catalyst 3850[31], Arista 7050[32]

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

5.2.1. Switch Convencional (Tradicional)

Un switch tradicional integra el plano de control y el plano de datos en el mismo dispositivo:

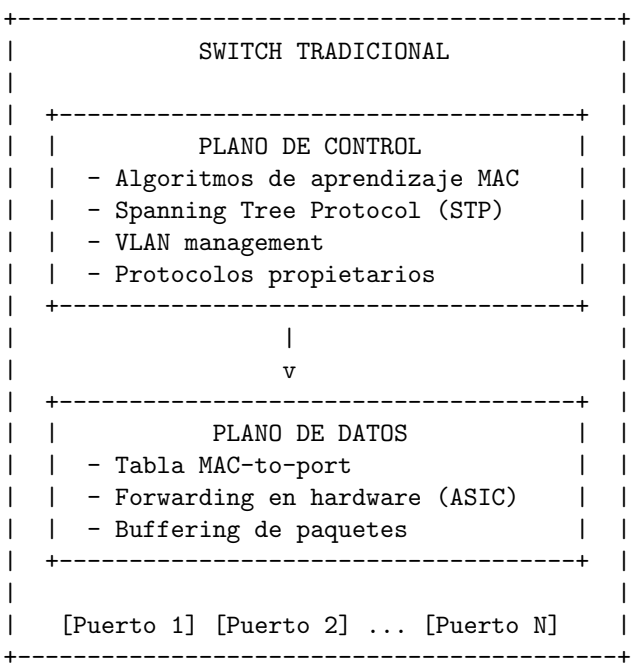


Figura 1: Arquitectura de un switch tradicional

Características del switch tradicional:

1. **Control distribuido:** Cada switch ejecuta sus propios algoritmos de control independientemente. No hay coordinación centralizada entre switches de diferentes fabricantes.
2. **Aprendizaje MAC automático:** El switch aprende direcciones MAC observando las direcciones fuente de los frames entrantes:

$$\text{Frame llega por puerto } P \text{ con MAC\_src } = M$$
$$\Rightarrow \text{Tabla MAC}[M] = P$$

3. **Decisión de forwarding:** Al recibir un frame con `MAC_dst = M`:
   - Si `M` está en la tabla: reenviar por puerto asociado
   - Si `M` no está: flooding (enviar por todos los puertos excepto el de entrada)
   - Si `M` es broadcast (`ff:ff:ff:ff:ff:ff`): flooding
4. **Protocolos embebidos:** Spanning Tree Protocol (`STP`) para prevenir loops, VLAN Trunking Protocol (`VTP`) para sincronizar VLANs, etc.
5. **Configuración propietaria:** Cada fabricante (Cisco, Juniper, HP) tiene su propia CLI y sintaxis de configuración.
6. **Difícil de programar:** No hay APIs estándares. Nuevas funcionalidades requieren actualizaciones de firmware o hardware del fabricante.
7. **Gestión descentralizada:** Cada switch debe configurarse individualmente. Políticas de red se implementan switch por switch.

5.2.2. Switch OpenFlow

Un switch OpenFlow separa el plano de control (que se mueve a un controlador externo) del plano de datos (que permanece en el switch):

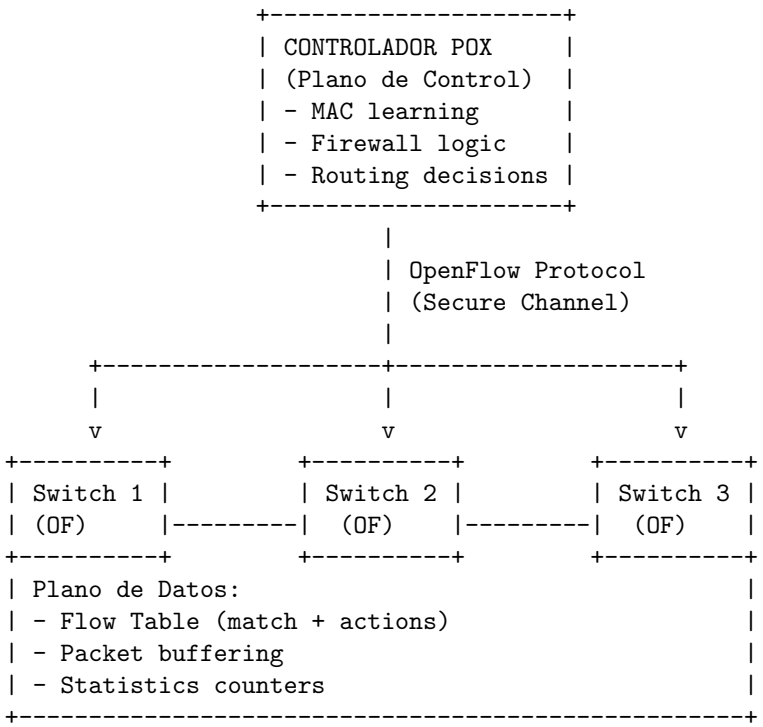


Figura 2: Arquitectura SDN con switches OpenFlow

Características del switch OpenFlow:

1. **Control centralizado:** El controlador externo (ej: POX, Ryu, ONOS) toma todas las decisiones de control y las comunica a los switches mediante el protocolo OpenFlow.

2. **Flow Table:** En lugar de una simple tabla `MAC-to-port`, los switches OpenFlow mantienen una **tabla de flujos** con entradas de la forma:

```
[Match Fields] -> [Actions] -> [Priority] -> [Counters] -> [Timeouts]
```

Ejemplo de entrada en flow table:

```
priority=10000, tcp, nw_src=10.0.0.1, tp_dst=80 -> actions=drop
priority=100, dl_src=00:00:00:00:00:02, dl_dst=00:00:00:00:00:04
    -> actions=output:3
```

3. **Match Fields (OpenFlow 1.0):** Permite matching en múltiples capas simultáneamente:
   - Capa 2: `dl_src`, `dl_dst`, `dl_type`, `dl_vlan`
   - Capa 3: `nw_src`, `nw_dst`, `nw_proto`, `nw_tos`
   - Capa 4: `tp_src`, `tp_dst`
   - Puerto de entrada: `in_port`

4. **Actions:** Múltiples acciones posibles:
   - `output:port` - Reenviar por puerto específico
   - `drop` - Descartar paquete (acción implícita: lista vacía)
   - `FLOOD` - Enviar por todos los puertos (excepto entrada)
   - `CONTROLLER` - Enviar al controlador
   - `modify_field` - Modificar campos (`MAC`, `IP`, `VLAN`)

5. **Comportamiento ante paquete desconocido:**
   a) Paquete llega al switch
   b) Switch busca match en flow table
   c) Si no hay match: envía **PacketIn** al controlador
   d) Controlador analiza el paquete y decide:
      - Instalar nueva regla (**FlowMod**)
      - Reenviar paquete (**PacketOut**)
      - Descartar paquete

6. **Prioridades:** Las entradas en la flow table tienen prioridades. Si un paquete matchea múltiples reglas, se ejecuta la de mayor prioridad:

Prioridad firewall: 10000
Prioridad forwarding: 100
Regla por defecto: 0

7. **Timeouts:** Las reglas pueden tener:
   - `idle_timeout`: Regla expira si no matchea paquetes durante X segundos
   - `hard_timeout`: Regla expira después de X segundos desde instalación

8. **Contadores por flujo:** Cada entrada mantiene estadísticas:
   - `n_packets`: Número de paquetes que matchearon
   - `n_bytes`: Bytes totales transferidos
   - `duration`: Tiempo desde instalación
9. **Protocolo OpenFlow:** Comunicación estandarizada entre controlador y switch:
   - **Controller → Switch:** FlowMod, PacketOut, BarrierRequest
   - **Switch → Controller:** PacketIn, FlowRemoved, PortStatus
   - **Bidireccional:** Hello, Echo (keepalive), StatsRequest/Reply
10. **Programabilidad:** El controlador puede implementar cualquier lógica de control en software:
    - Firewall (como en este trabajo)
    - Load balancer
    - QoS dinámico
    - Routing personalizado
    - Network monitoring

5.2.3. Comparación Directa

Aspecto	**Switch Tradicional**	**Switch OpenFlow**
Plano de control	Integrado en el switch	Centralizado en controlador externo
Lógica de forwarding	Fija (aprendizaje `MAC`, `STP`)	Programable vía software
Granularidad de reglas	Solo `MAC-to-port`	Match en L2, L3, L4 simultáneamente
Instalación de reglas	Aprendizaje automático reactivo	Proactiva o reactiva según controlador
Políticas de red	Configuradas switch por switch	Definidas centralmente, aplicadas globalmente
Interoperabilidad	Propietaria (vendor lock-in)	Estándar abierto (OpenFlow spec)
Flexibilidad	Limitada a funciones embebidas	Altamente flexible (nuevo código = nueva funcionalidad)
Latencia primer paquete	Muy baja	Mayor (round-trip al controlador)
Throughput sostenido	Alto	Comparable (forwarding en hardware)
Complejidad de gestión	Alta (gestión distribuida)	Baja (gestión centralizada)
Costos	Hardware especializado caro	Switches commodity + controlador SW
Casos de uso	Redes tradicionales L2/L3	Data centers, SDN, network slicing, experimentación

Cuadro 9: Comparación detallada: Switch tradicional vs OpenFlow

5.2.4. Ejemplo Práctico: Bloqueo de Puerto 80

Switch Tradicional:

1. Configurar ACL (Access Control List) en cada switch individualmente:

```
Switch(config)# access-list 100 deny tcp any any eq 80
Switch(config)# interface GigabitEthernet0/1
Switch(config-if)# ip access-group 100 in
```

2. Repetir en cada switch del dominio
3. Gestión distribuida: cambiar regla requiere modificar cada switch

Switch OpenFlow:

1. Definir regla en archivo JSON del controlador:

```
{
    "name": "Block port 80",
    "protocol": "TCP",
    "dst_port": 80
}
```

2. El controlador instala automáticamente la regla en todos los switches:

```
FlowMod: priority=10000, tcp, tp_dst=80 -> actions=drop
```

3. Gestión centralizada: cambiar regla en un solo lugar

5.3. ¿Se pueden reemplazar todos los routers de la Internet por Switches OpenFlow? Piense en el escenario inter-AS para elaborar su respuesta

La respuesta corta es **NO**, al menos no con la tecnología actual de OpenFlow 1.0 y considerando el escenario inter-AS (inter-Autonomous System) de Internet global. A continuación se desarrollan las razones técnicas, escalabilidad y limitaciones.

5.3.1. Arquitectura de Internet: Sistemas Autónomos (AS)

Internet está compuesta por decenas de miles de **Autonomous Systems (AS)**, cada uno operado por una organización diferente (ISPs, universidades, empresas):

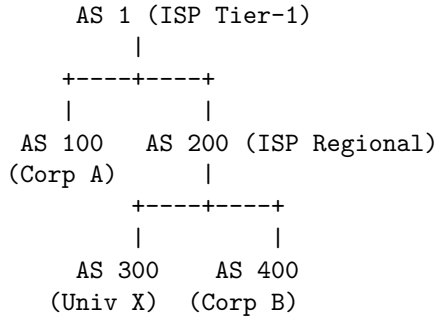


Figura 3: Jerarquía simplificada de Autonomous Systems en Internet

Características clave de routing inter-AS:

1. **Border Gateway Protocol (BGP)[26]:** Protocolo estándar para intercambiar información de enrutamiento entre ASes diferentes. Cada AS anuncia sus prefijos IP y aprende rutas hacia otros ASes.
2. **Políticas de routing complejas:** Los ASes aplican políticas basadas en:
   - Relaciones comerciales (customer, provider, peer)
   - Preferencias de tráfico (evitar ciertas rutas)
   - Ingeniería de tráfico (balanceo de carga)
   - Seguridad (filtrado de rutas, RPKI[33])
3. **Autonomía administrativa:** Cada AS es administrado independientemente. No existe (ni debe existir) una entidad centralizada que controle el routing global de Internet.
4. **Escalabilidad:** La tabla de rutas BGP global contiene ¿900,000 prefijos `IPv4` y ¿150,000 prefijos `IPv6` (datos 2024).

5.3.2. Limitaciones de OpenFlow para Routing Inter-AS

1. Limitación del Plano de Control Centralizado

Problema: OpenFlow asume un controlador centralizado con visibilidad completa de la red. En Internet:

- **Imposible centralizar:** No hay forma de que un solo controlador (o incluso un cluster) maneje el control de decenas de miles de ASes distribuidos globalmente.
- **Latencia inaceptable:** Si un paquete en Australia necesita consultar al controlador en Europa antes de ser reenviado, la latencia sería de cientos de milisegundos, rompiendo aplicaciones en tiempo real.
- **Single point of failure:** Un controlador centralizado para Internet sería un objetivo crítico para ataques DDoS[34] y fallos catastróficos.

Cálculo ilustrativo:

$$\begin{aligned} \text{Latencia round-trip intercontinental} &\approx 200 \text{ ms} \\ \text{Procesamiento en controlador} &\approx 10 \text{ ms} \\ \text{Instalación de regla} &\approx 50 \text{ ms} \\ \hline \text{Latencia total primer paquete} &\approx 260 \text{ ms} \end{aligned}$$

Comparado con routing BGP tradicional donde el reenvío ocurre en microsegundos.

2. Escalabilidad de Flow Tables

Problema: Los switches OpenFlow tienen capacidad limitada en sus flow tables (típicamente 1,000 - 10,000 entradas en `TCAM`).

- **Tabla de rutas global:** ¿900,000 prefijos `IPv4` no caben en la `TCAM`[35] de un switch OpenFlow estándar.
- **Agregación limitada:** BGP realiza agregación de prefijos extensiva. OpenFlow 1.0 no tiene mecanismos sofisticados para agregación jerárquica de reglas.
- **Actualización dinámica:** La tabla BGP cambia constantemente (nuevas rutas, retiros, cambios de política). Actualizar 900,000+ entradas vía FlowMod desde un controlador sería extremadamente lento.

Ejemplo numérico:

$$\begin{aligned} \text{Prefijos IPv4 globales} &\approx 950,000 \\ \text{Capacidad TCAM típica} &\approx 10,000 \\ \text{Ratio de overflow} &\approx 95:1 \text{ (insostenible)} \end{aligned}$$

3. Falta de Soporte para Protocolos de Routing

Problema: OpenFlow 1.0 es un protocolo de plano de datos. No incluye:

- **BGP:** El controlador OpenFlow debería reimplementar BGP en software, comunicarse con routers BGP externos, y traducir decisiones de routing a FlowMods. Esto es técnicamente posible pero extremadamente complejo.
- **OSPF/IS-IS:** Protocolos de routing intra-AS tampoco están soportados nativamente.
- **Multipath routing:** BGP soporta ECMP (Equal-Cost Multi-Path)[36]. OpenFlow 1.0 tiene soporte limitado para balanceo de carga entre múltiples caminos.

4. Políticas y Autonomía Administrativa

Problema: Internet funciona porque cada AS mantiene autonomía para definir sus propias políticas.

- **Políticas comerciales:** Un ISP tier-1 no aceptaría que un controlador externo dicte sus decisiones de routing. Las políticas de tránsito, peering, y customer routing son secretos comerciales.
- **Seguridad:** Exponer el control de routing a un controlador externo crea vectores de ataque masivos.
- **Regulación:** Diferentes países tienen requisitos legales sobre routing de tráfico (ej: data sovereignty, censura).

5. Convergencia y Estabilidad

Problema: BGP está diseñado para converger ante cambios de topología (fallos de enlaces, nuevas rutas).

- **Tiempo de convergencia:** BGP puede tardar segundos-minutos en converger globalmente. Un controlador OpenFlow centralizado enfrentaría desafíos similares o peores debido a la latencia de comunicación.
- **Route flapping:** BGP tiene mecanismos (route damping) para manejar rutas inestables. El controlador OpenFlow debería replicar esta lógica.
- **Loops de routing:** BGP usa `AS-PATH` para prevenir loops. OpenFlow no tiene un mecanismo análogo incorporado.

5.3.3. Escenarios Donde OpenFlow SÍ es Viable

A pesar de las limitaciones globales, OpenFlow es extremadamente efectivo en ciertos contextos:

1. Intra-AS / Data Centers

Características favorables:

- Dominio administrativo único
- Topología conocida y controlada
- Número limitado de switches (¡1000)
- Latencia baja al controlador
- Políticas uniformes

Ejemplo: Este trabajo implementa un firewall SDN en una topología de 4 switches. Escalar a 100-1000 switches en un data center es factible con controladores de alta disponibilidad (ej: ONOS[37] con clustering).

2. Campus Networks

Caso de uso: Universidades y empresas con múltiples edificios pero administración centralizada.

- Topología: 10-100 switches de acceso + switches de distribución/core
- Beneficios: Políticas de seguridad centralizadas, segmentación de red dinámica (por ej: aislamiento de invitados)

3. Edge/Access Networks

Caso de uso: ISPs pueden usar OpenFlow en la red de acceso (última milla) mientras mantienen BGP en el core.

- Switches OpenFlow en CPE (Customer Premises Equipment)
- Controlador gestiona QoS, VLAN assignment, parental controls
- Core network sigue usando BGP/MPLS

4. Hybrid SDN

Enfoque: Combinar routers tradicionales para inter-AS con switches OpenFlow para intra-AS.

```
[BGP Router] --- Internet --- [BGP Router]
     |                             |
[OpenFlow                    [OpenFlow
 Switches]                    Switches]
 Intra-AS                     Intra-AS
```

5.3.4. Evolución: SDN en el Core de Internet

Aunque OpenFlow 1.0 no puede reemplazar BGP, existen esfuerzos de investigación y protocolos más avanzados:

1. OpenFlow 1.3+

Versiones posteriores de OpenFlow incluyen:

- Múltiples tablas de flujos (pipeline processing)
- Matching en más de 40 campos
- Group tables para multipath
- Meters para QoS

Aún así, no resuelven los problemas fundamentales de escalabilidad y autonomía administrativa.

2. SDN para Ingeniería de Tráfico

Ejemplo real: Google B4 WAN[38]

- Google usa SDN (protocolo propietario, no OpenFlow estándar) para gestionar su WAN privada entre data centers.
- **Clave:** Es una red privada, no Internet pública. Google controla todos los switches.
- Beneficio: Utilización de enlaces ¿90 % vs 40 % con MPLS tradicional.

3. Segment Routing (SR) y SRv6 [39]

Enfoque híbrido:

- Mantiene control distribuido (routers independientes)
- Añade capacidad de "source routing"(el origen especifica el camino)
- Compatible con BGP existente
- Usado por operadores tier-1 (AT&T, China Telecom)

4. P4 y Programmable Data Planes [40]

P4 (Programming Protocol-Independent Packet Processors):

- Lenguaje para programar el plano de datos de switches
- Más flexible que OpenFlow (no limitado a campos predefinidos)
- Permite crear protocolos de forwarding personalizados
- **Limitación:** Aún requiere solucionar los problemas de escalabilidad y control distribuido para Internet global

5.3.5. Respuesta Final: ¿Por qué NO reemplazar todos los routers?

Razón	Explicación
Imposibilidad de control centralizado global	Internet depende de la autonomía de decenas de miles de ASes. Un controlador centralizado violaría este principio fundamental.
Escalabilidad de flow tables	900,000+ prefijos `IPv4` no caben en TCAM de switches commodity.
Latencia del primer paquete	Consultar un controlador remoto añade 100-300 ms, inaceptable para aplicaciones en tiempo real.
Falta de soporte para BGP	OpenFlow no incluye mecanismos de routing inter-AS. Reimplementar BGP en el controlador es posible pero impráctica.
Políticas comerciales y regulatorias	Los ASes no cederían control de sus decisiones de routing a una entidad externa.
Convergencia y estabilidad	BGP tiene décadas de optimizaciones para manejar cambios de topología. OpenFlow no puede replicar esto fácilmente a escala global.
Seguridad	Un controlador centralizado para Internet sería el mayor single point of failure jamás creado.

Cuadro 10: Razones por las cuales OpenFlow no puede reemplazar routers BGP inter-AS

5.3.6. Conclusión

- **Intra-AS:** SÍ, OpenFlow (y SDN en general) puede y está reemplazando routers tradicionales en data centers y redes de campus.

- **Inter-AS (Internet global):** NO, OpenFlow no puede reemplazar BGP debido a:
  1. Imposibilidad de centralización a escala global
  2. Limitaciones de capacidad en flow tables
  3. Requerimientos de autonomía administrativa
  4. Necesidad de políticas de routing complejas y distribuidas
- **Futuro:** Los enfoques híbridos (SDN intra-AS + BGP inter-AS) son más realistas. Tecnologías como Segment Routing ofrecen programabilidad sin sacrificar la arquitectura distribuida de Internet.

Analogía: Reemplazar todos los routers de Internet con switches OpenFlow sería como intentar gestionar toda la economía global desde un banco central único. La complejidad, diversidad de intereses, y necesidad de autonomía local hacen que un modelo distribuido sea la única opción viable.

6. Dificultades Encontradas y Soluciones

6.1. Configuración Inicial de la Topología

Una de las dificultades iniciales fue configurar correctamente la topología en Mininet con múltiples switches conectados en cascada y asegurar que todos estuvieran bajo el control de un único controlador remoto.

Problema: Mininet por defecto crea switches con controladores internos. Se requería especificar el parámetro `--controller remote` para usar un controlador externo.

Solución: Se utilizó el comando correcto:

```
sudo mn --custom topology.py --topo customTopo,num_switches=4 \
  --controller remote --switch ovsk --mac --arp
```

Se agregó la opción `--mac` para asignar direcciones `MAC` basadas en `IP` y `--arp` para habilitar `ARP`.

6.2. Aprendizaje de Direcciones MAC

El controlador debe aprender dinámicamente la topología observando PacketIn. Sin embargo, el aprendizaje incompleto causaba que algunos hosts no fuera alcanzables.

Problema: Los primeros paquetes generaban PacketIn, pero si el controlador no procesaba rápidamente, se perdían paquetes y la topología quedaba incompleta.

Solución: Se agregó un mecanismo de feedback: cada vez que se instala un flujo aprendido, se envía el paquete pendiente (`ofp_packet_out`) al destino, asegurando que no se pierda el paquete disparador del aprendizaje.

6.3. Conflictos entre Reglas Proactivas y Aprendidas

En el switch `s2` (firewall), pueden coexistir reglas proactivas (de firewall) y reglas aprendidas (flujos dinámicos). Esto creó ambigüedad en cuál aplicar cuando un paquete coincide con múltiples reglas.

Problema: Sin prioridades correctas, un flujo aprendido podría actuar antes que una regla de bloqueo.

Solución: Se utilizaron prioridades distintas:

- **Reglas proactivas (firewall):** Prioridad 10000
- **Flujos aprendidos:** Prioridad 100

Las prioridades más altas se evalúan primero, asegurando que el firewall prevalezca.

6.4. Comportamiento de Flooding

El flooding por todos los puertos es necesario para descubrir destinos desconocidos, pero puede causar tormenta de paquetes en topologías con ciclos.

Problema: En Mininet con topología lineal, el flooding no es problema, pero en redes complejas puede causar duplicación de paquetes.

Solución: Se implementó verificación para evitar re-enviar paquetes por el puerto de entrada:

```
if out_port == in_port:
    log.warning("Dropping: src y dst en el mismo puerto")
    return
```

6.5. Depuración de OpenFlow

Entender qué ocurre en los switches es difícil sin herramientas de debugging.

Problema: Los logs del controlador no siempre reflejaban exactamente el estado de los switches.

Solución: Se utilizó `ovs-ofctl dump-flows` para consultar el contenido exacto de las tablas de flujos:

```
mininet> s1 ovs-ofctl dump-flows s1
```

Esto proporcionaba una vista clara de qué reglas estaban instaladas y sus prioridades.

7. Conclusión

El presente trabajo ha demostrado exitosamente la implementación de un controlador OpenFlow utilizando POX para administrar una topología de red con múltiples switches. Los objetivos principales del trabajo practico fueron alcanzados:

7.1. Logros Principales

1. **Topología funcional:** Se implementó una topología lineal con 4 switches y 4 hosts distribuidos en Mininet.
2. **Controlador centralizado:** Se desarrolló un controlador POX que se conecta a los switches mediante OpenFlow 1.0.
3. **Aprendizaje dinámico:** El controlador aprende la topología de la red observando PacketIn y construye una tabla de `MAC` a puerto.
4. **Encaminamiento basado en flujos:** Se implementó instalación de flujos estáticos en switches que actúan como switches tradicionales (`NORMAL` behavior).
5. **Firewall de switch específico:** El switch s2 implementa reglas de bloqueo a nivel de `MAC`, `IP` y puertos desde un archivo `JSON`.
6. **Pruebas exitosas:** Se verificó conectividad completa entre hosts y se validó el funcionamiento del firewall.

7.2. Aprendizajes Clave

A través del desarrollo e implementación, se obtuvieron los siguientes aprendizajes:

- **Separación de planos:** La separación del plano de control y datos simplifica significativamente la administración de redes.
- **Ventajas de SDN:** Usar un controlador centralizado permitió implementar políticas complejas (firewall) sin cambios en el hardware de los switches.
- **OpenFlow como abstracción:** OpenFlow proporciona una abstracción clara y programable del comportamiento de switches.
- **Escalabilidad del controlador:** Un controlador simple es efectivo para topologías pequeñas, pero sería un cuello de botella en redes grandes.
- **Importancia de prioridades:** La correcta asignación de prioridades en flujos es crítica para resolver conflictos entre reglas.

7.3. Limitaciones

A pesar de los logros, el sistema actual presenta limitaciones:

- **Punto único de fallo:** El controlador es un punto crítico de falla; su caída desconecta la red.
- **Escalabilidad:** El aprendizaje (per-switch) no escala a redes grandes con cientos de switches.
- **Sin redundancia:** No hay mecanismo de failover si el controlador falla.
- **OpenFlow 1.0:** La versión del protocolo utilizada tiene capacidades limitadas:
  - No soporta nativamente `IPv6`, matching de campos arbitrarios, ni metadata
  - Solo reconoce oficialmente TCP y UDP para matching de puertos (`tp_src`/`tp_dst`)
  - SCTP puede especificarse como protocolo (`nw_proto=132`) pero sin soporte de puertos
  - Versiones posteriores (1.3+)[10] ofrecen mayor flexibilidad y soporte extendido de protocolos

7.4. Reflexión Final

Este trabajo práctico ha proporcionado una comprensión profunda de cómo funcionan las redes modernas basadas en SDN. La implementación práctica de un controlador OpenFlow, aunque simplificado, ilustra los principios fundamentales que subyacen a infraestructuras complejas en centros de datos e internet moderno. La programabilidad de la red abre posibilidades para innovación y optimización que no eran posibles con arquitecturas de networking tradicionales.

Referencias

[1] Cisco Systems. Software-defined networking (sdn). https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html, 2025. Accessed: 2025.

[2] Open Networking Foundation. Open datapath. https://opennetworking.org/technical-communities/areas/specification/open-datapath/, 2025. Accessed: 2025.

[3] Cisco Systems. What is a firewall? https://www.cisco.com/site/us/en/learn/topics/security/what-is-a-firewall.html, 2025. Accessed: 2025.

[4] NOXRepo. *POX Controller Documentation*. NOXRepo, 2025. Accessed: 2025.

[5] Mininet Team. Mininet: An instant virtual network on your laptop (or other pc). https://mininet.org/, 2025. Accessed: 2025.

[6] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.0.0*. Open Networking Foundation, 12 2009. Accessed: 2025.

[7] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, 8 2008. Internet Engineering Task Force.

[8] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification. RFC 2460, 12 1998. Internet Engineering Task Force.

[9] D. Plummer. An ethernet address resolution protocol. RFC 826, 11 1982. Internet Engineering Task Force.

[10] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.3.0*. Open Networking Foundation, 6 2012. Accessed: 2025.

[11] Cisco Systems. Multiprotocol label switching (mpls). https://www.cisco.com/c/en/us/tech/multiprotocol-label-switching-mpls/index.html, 2025. Accessed: 2025.

[12] Open vSwitch Project. *Open vSwitch Documentation*. Open vSwitch, 2025. Accessed: 2025.

[13] ECMA International. Json (javascript object notation). https://www.json.org/, 2017. Standard ECMA-404.

[14] Amazon Web Services. ¿qué es la notación cidr? https://aws.amazon.com/es/what-is/cidr/, 2025. Accessed: 2025.

[15] Open vSwitch Project. ovs-ofctl - openflow switch management utility. https://docs.openvswitch.org/en/latest/ref/ovs-ofctl.8/, 2025. Accessed: 2025.

[16] Wireshark Foundation. Wireshark - network protocol analyzer. https://www.wireshark.org/, 2025. Accessed: 2025.

[17] Python Software Foundation. Python http.server - http servers. https://docs.python.org/3/library/http.server.html, 2025. Accessed: 2025.

[18] curl Project. *curl - Command Line Tool and Library for Transferring Data with URLs*. curl, 2025. Accessed: 2025.

[19] Oracle Corporation. *netcat(1) - Linux Man Page*. Oracle, 2025. Accessed: 2025.

[20] iperf. *iperf Documentation*. iperf, 2025. Accessed: 2025.

[21] Cisco Systems. Tcp three-way handshake. https://www.cisco.com/c/en/us/support/docs/ip/transmission-control-protocol-tcp/8742-tcp-handshake.html, 2025. Accessed: 2025.

[22] J. Postel. Internet control message protocol. RFC 792, 9 1981. Internet Engineering Task Force.

[23] Cisco Systems. Understanding spanning tree protocol (802.1d). https://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/5234-5.html, 2025. Accessed: 2025.

[24] G. Malkin. Rip version 2. RFC 2453, 11 1998. Internet Engineering Task Force.

[25] J. Moy. Ospf version 2. RFC 2328, 4 1998. Internet Engineering Task Force.

[26] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (bgp-4). RFC 4271, 1 2006. Internet Engineering Task Force.

[27] T. Spalink, S. Karlin, and L. Peterson. Network processors and application-specific integrated circuits. *ACM SIGCOMM Computer Communication Review*, 31(4):11–22, 2001.

[28] IEEE. Ieee 802.1q - virtual lans. https://standards.ieee.org/ieee/802.1Q/, 2022. Accessed: 2025.

[29] Cisco Systems. Network device management via cli. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/fundamentals/command/cf_command_ref.html, 2025. Accessed: 2025.

[30] Internet Engineering Task Force. Simple network management protocol (snmp). https://datatracker.ietf.org/wg/snmp/about/, 2025. Accessed: 2025.

[31] Cisco Systems. Cisco catalyst 3850 series switches data sheet. 2025.

[32] Arista Networks. Arista 7050 series data center switches. https://www.arista.com/en/products/7050-series, 2025. Accessed: 2025.

[33] Internet Society. Resource public key infrastructure (rpki). https://www.internetsociety.org/resources/doc/2020/resource-public-key-infrastructure-rpki/, 2025. Accessed: 2025.

[34] Cloudflare. What is a ddos attack? https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/, 2025. Accessed: 2025.

[35] Cisco Systems. Understanding tcam and cam tables. https://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/24046-143.html, 2023. Accessed: 2025.

[36] D. Thaler and C. Hopps. Analysis of an equal-cost multi-path algorithm. RFC 2992, 11 2000. Internet Engineering Task Force.

[37] Open Networking Foundation. Onos - open network operating system. https://opennetworking.org/onos/, 2025. Accessed: 2025.

[38] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[39] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. Segment routing architecture. RFC 8402, 7 2018. Internet Engineering Task Force.

[40] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 87–95. ACM, 2014.