Universidad de Buenos Aires
Facultad de Ingeniería

Redes - Hamelin

Trabajo Práctico N.º 2: “Software-Defined Networks”

2do Cuatrimestre 2025

Alumno:

- Ezequiel Martín Aragón - 110643
- Martín Castro - 109807
- Nicolás Khalil Chaia - 110768
- Gonzalo Nicolas Crudo - 110816
- Franco Guardia - 109374

Introducción

El presente trabajo tiene como finalidad el diseño e implementación de un firewall basado en Software-Defined Networking (SDN) utilizando el controlador POX y el emulador de red Mininet. El objetivo principal es comprender en profundidad cómo se pueden implementar políticas de seguridad a nivel de red mediante el control centralizado que ofrece la arquitectura SDN.

Para alcanzar este objetivo, se desarrolló un módulo de firewall que permite filtrar tráfico de red mediante reglas configurables definidas en un archivo JSON. El firewall utiliza el protocolo OpenFlow para instalar reglas de flujo en los switches, permitiendo un control granular del tráfico de red.

Se implementaron tres reglas específicas que demuestran diferentes capacidades de filtrado: bloqueo del puerto 80 (HTTP), bloqueo de tráfico UDP desde un host específico hacia un puerto determinado, y bloqueo bidireccional de comunicación entre dos hosts específicos.

La aplicación desarrollada permite validar el funcionamiento del firewall mediante pruebas exhaustivas utilizando herramientas estándar como ping e iperf. Asimismo, se analiza el comportamiento del sistema ante diferentes tipos de tráfico y se evalúan las ventajas y limitaciones de los firewalls SDN en comparación con los firewalls tradicionales.

Hipótesis y suposiciones realizadas

Hipótesis

- El firewall se aplica en un único switch de la topología, designado mediante el parámetro “switch_with_firewall” en el archivo de configuración.
- Las reglas de firewall tienen mayor prioridad que la regla por defecto que permite todo el tráfico.
- El tráfico solo se filtra si pasa por el switch donde está configurado el firewall.
- Si no se especifica el protocolo en una regla que incluye puerto, se asume TCP por defecto.

Suposiciones

- El controlador POX se ejecuta en la misma máquina que Mininet.
- Los switches OpenFlow se conectan automáticamente al controlador remoto.
- Los hosts tienen direcciones IP asignadas automáticamente por Mininet (10.0.0.1, 10.0.0.2, etc.).
- El archivo rules.json se encuentra en una ruta absoluta conocida.
- Las reglas se instalan cuando un switch se conecta al controlador.

- Una vez instaladas las reglas en un switch, no se vuelven a instalar en reconexiones.

Implementación

El firewall implementado consta de los siguientes componentes principales:

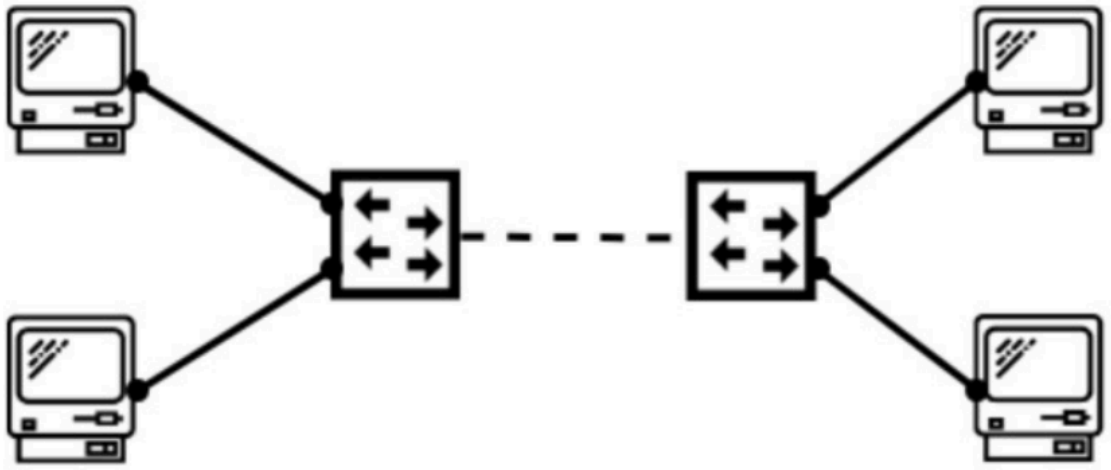
Arquitectura del sistema

El sistema se compone de tres elementos fundamentales:

1. Controlador POX: Gestiona la red y ejecuta el módulo de firewall
2. Switches OpenFlow: Dispositivos de red que aplican las reglas de flujo
3. Mininet: Emula la topología de red con hosts y switches virtuales

Topología de Red

La topología implementada consiste en una cadena lineal de switches con hosts conectados en los extremos.



La topología se define programáticamente utilizando la clase `Topo` de Mininet, creando los hosts y switches en un bucle y estableciendo los enlaces correspondientes para formar la estructura lineal descrita.

Módulo de Firewall

El módulo de firewall se implementa como un componente POX que se integra con el controlador. El flujo de funcionamiento es el siguiente:

1. Inicio POX: Se inicia el controlador.
2. Cargar reglas: Se lee el archivo rules.json.
3. Conexión de switch: Se espera a que un switch se conecte.
4. Verificación: Se comprueba si el switch conectado es el designado para el firewall.
5. SÍ: Se instalan las reglas específicas de bloqueo y luego la regla por defecto.
6. NO: Se instala solo la regla por defecto.
7. Regla por defecto: Permite todo el tráfico (acción output normal o flood).
8. Operativo: El sistema queda listo para procesar tráfico.Módulo de Firewall

Componentes principales

1. _load_rules(): Carga y parsea el archivo JSON de configuración
2. _check_firewall_rules(): Verifica si el switch conectado es el designado para el firewall
3. _add_flow_rule(): Construye e instala reglas OpenFlow en el switch
4. _handle_ConnectionUp(): Maneja el evento de conexión de un switch

Procesamiento de Reglas

El módulo procesa las reglas de la siguiente manera:

1. Carga inicial: Al iniciar POX, se carga el archivo rules.json
2. Identificación de switch: Cuando un switch se conecta, se extrae su ID numérico del DPID
3. Instalación selectiva: Solo el switch designado (switch_with_firewall: 1) recibe las reglas de bloqueo
4. Regla por defecto: Todos los switches reciben una regla de baja prioridad que permite el tráfico normal

Construcción de Matches OpenFlow

El módulo construye los criterios de coincidencia (matches) para OpenFlow siguiendo esta lógica:

- Detección de IPv4: Si la regla incluye campos de capa 3 (IP origen/destino, protocolo) o capa 4 (puertos), se establece automáticamente el tipo de Ethernet a IPv4 (0x0800).

- Campos de Match: Se configuran las direcciones IP origen/destino y el protocolo IP si están presentes en la regla.
- Inferencia de TCP: Si se especifica un puerto de destino u origen pero no se indica el protocolo de red, el sistema asume por defecto que se trata de tráfico TCP (protocolo 6).

Sistema de Prioridades

El sistema de prioridades garantiza que las reglas se evalúen en el orden correcto:
- Reglas de firewall: Prioridad 10 + ID de regla (ej. 11, 12, 13).
- Regla por defecto: Prioridad 1.
- Las reglas de mayor prioridad se evalúan primero.

Este diseño implementa una política “deny by default” para las reglas específicas configuradas: si un paquete coincide con una regla de bloqueo, se descarta; de lo contrario, se permite por la regla por defecto.

Reglas del Firewall

Se implementaron tres reglas específicas que demuestran diferentes capacidades de filtrado:

Regla 1: Bloqueo de Puerto 80 (HTTP)

- Objetivo: Bloquear todo el tráfico TCP con puerto destino 80.
- Configuración de la regla:

```
{
  "rule_id": 1,
  "description": "Descartar todos los mensajes cuyo puerto destino sea 80",
  "match": {
    "tp_dst": 80
  },
  "action": "drop"
}
```

- Características técnicas: -
  Protocolo: TCP (asumido por defecto).
  Prioridad: 11.

Regla 2: Bloqueo UDP desde h1 al Puerto 5001

- Objetivo: Bloquear tráfico UDP desde el host 10.0.0.1 hacia el puerto 5001.
- Configuración de la regla:

```
{
  "rule_id": 2,
  "description": "Descartar mensajes del host 1 con puerto destino 5001 usando UDP",
  "match": {
    "nw_src": "10.0.0.1",
    "nw_proto": 17,
    "tp_dst": 5001
  },
  "action": "drop"
}
```

- Características técnicas:
  Prioridad: 12.
  Selectividad: Solo afecta al host h1, otros hosts pueden usar UDP 5001.

Regla 3: Bloqueo Bidireccional h2 ↔ h3

- Objetivo: Bloquear toda comunicación entre los hosts 10.0.0.2 y 10.0.0.3.
- Configuración de la regla:

```
{
  "rule_id": 3,
  "description": "Bloquear comunicación entre host 2 y host 3",
  "match": [
    {
      "nw_src": "10.0.0.2",
      "nw_dst": "10.0.0.3"
    },
    {
      "nw_src": "10.0.0.3",
      "nw_dst": "10.0.0.2"
    }
  ],
  "action": "drop"
}
```

- Características técnicas:
  Prioridad: 13.
  Alcance: Bloquea todo tipo de tráfico entre estos dos hosts.

Pruebas

Prueba I: Pingall

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

El resultado muestra que **solo el tráfico entre h2 y h3 está bloqueado**, tal como establecen las reglas del firewall, mientras que el resto de los hosts pueden comunicarse normalmente.

- **h1** puede hacer ping a h2, h3 y h4 sin restricciones.
- **h2** puede comunicarse con h1 y h4, pero **no** con h3 (bloqueo esperado).
- **h3** puede comunicarse con h1 y h4, pero **no** con h2 (bloqueo esperado).
- **h4** puede comunicarse con todos los hosts.

En Wireshark también podemos apreciar esto:

3	0.000212444	10.0.0.1	10.0.0.2	ICMP	98 Echo (ping) request id=0x7a0b, seq=1/256,
4	0.000343818	10.0.0.2	10.0.0.1	ICMP	98 Echo (ping) reply id=0x7a0b, seq=1/256,
5	0.012942533	10.0.0.1	10.0.0.3	ICMP	98 Echo (ping) request id=0x7a0c, seq=1/256,
6	0.013276557	10.0.0.3	10.0.0.1	ICMP	98 Echo (ping) reply id=0x7a0c, seq=1/256,
7	0.014267385	22:e6:71:10:6e:a6	Broadcast	ARP	42 Who has 10.0.0.4? Tell 10.0.0.1
8	0.014450983	02:b4:d4:16:3f:dd	22:e6:71:10:6e:a6	ARP	42 10.0.0.4 is at 02:b4:d4:16:3f:dd
9	0.014453051	10.0.0.1	10.0.0.4	ICMP	98 Echo (ping) request id=0x7a0d, seq=1/256,
10	0.014564905	10.0.0.4	10.0.0.1	ICMP	98 Echo (ping) reply id=0x7a0d, seq=1/256,
11	0.015558140	10.0.0.2	10.0.0.1	ICMP	98 Echo (ping) request id=0x7a0e, seq=1/256,
12	0.015563998	10.0.0.1	10.0.0.2	ICMP	98 Echo (ping) reply id=0x7a0e, seq=1/256,
13	5.494625235	02:b4:d4:16:3f:dd	22:e6:71:10:6e:a6	ARP	42 Who has 10.0.0.1? Tell 10.0.0.4
14	5.494688699	22:e6:71:10:6e:a6	02:b4:d4:16:3f:dd	ARP	42 10.0.0.1 is at 22:e6:71:10:6e:a6
15	5.495433035	22:e6:71:10:6e:a6	d2:67:f6:37:62:32	ARP	42 Who has 10.0.0.3? Tell 10.0.0.1
16	5.495551128	ea:72:f2:8c:a0:a2	22:e6:71:10:6e:a6	ARP	42 Who has 10.0.0.1? Tell 10.0.0.2
17	5.495588704	22:e6:71:10:6e:a6	ea:72:f2:8c:a0:a2	ARP	42 10.0.0.1 is at 22:e6:71:10:6e:a6
18	5.496145085	d2:67:f6:37:62:32	22:e6:71:10:6e:a6	ARP	42 Who has 10.0.0.1? Tell 10.0.0.3
19	5.496156337	22:e6:71:10:6e:a6	d2:67:f6:37:62:32	ARP	42 10.0.0.1 is at 22:e6:71:10:6e:a6
20	5.496226548	d2:67:f6:37:62:32	22:e6:71:10:6e:a6	ARP	42 10.0.0.3 is at d2:67:f6:37:62:32
21	10.021043955	ea:72:f2:8c:a0:a2	Broadcast	ARP	42 Who has 10.0.0.4? Tell 10.0.0.2
22	10.025358637	10.0.0.3	10.0.0.1	ICMP	98 Echo (ping) request id=0x7a27, seq=1/256,
23	10.025381334	10.0.0.1	10.0.0.3	ICMP	98 Echo (ping) reply id=0x7a27, seq=1/256,
24	20.069018463	d2:67:f6:37:62:32	Broadcast	ARP	42 Who has 10.0.0.4? Tell 10.0.0.3
25	20.084921880	10.0.0.4	10.0.0.1	ICMP	98 Echo (ping) request id=0x7a43, seq=1/256,
26	20.084939042	10.0.0.1	10.0.0.4	ICMP	98 Echo (ping) reply id=0x7a43, seq=1/256,

Se pueden ver todos los pings realizados (tanto el request como el reply correspondiente) en aquellas comunicaciones que fueron exitosas, y solo no se muestran las que están afectadas por el bloqueo del firewall

Prueba II: Bloqueo de puerto 80 con iperf

```
------------------------------------------------------------
Client connecting to 10.0.0.3, TCP port 80
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
tcp connect failed: No route to host
[  1] local 0.0.0.0 port 0 connected with 10.0.0.3 port 80
```

Resultado del cliente (*iperf80-client.log*)

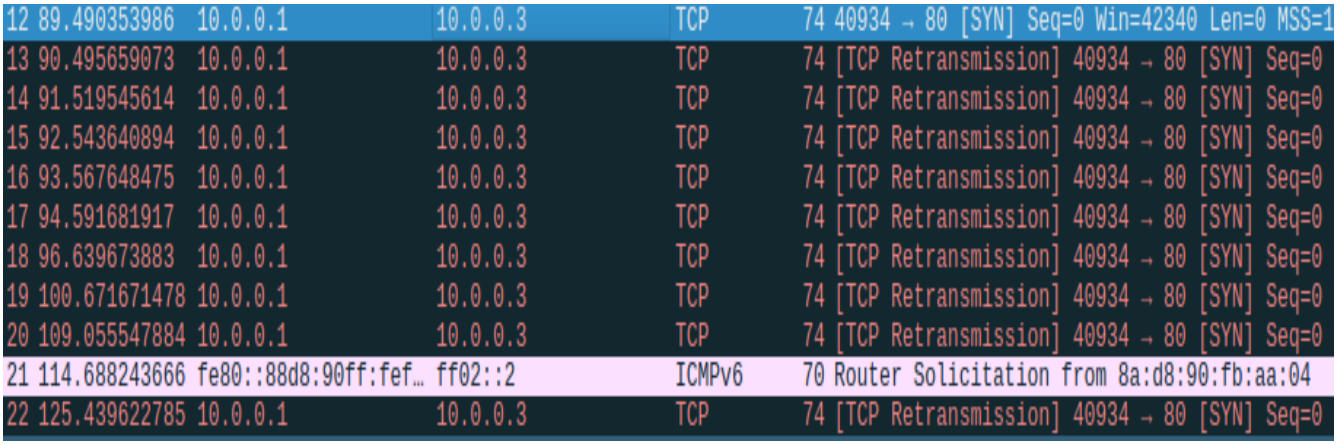
Este mensaje aparece cuando el cliente envía paquetes **SYN** para iniciar la conexión TCP, pero no recibe ninguna respuesta del servidor. Esto no es un error de configuración ni de red: ocurre porque el firewall está aplicando la regla de **DROP** sobre el tráfico TCP con destino al puerto 80. Como los paquetes son descartados silenciosamente, `iperf` interpreta la falta total de respuesta como “No route to host”, aunque en realidad la ruta existe.
Este comportamiento es el esperado: confirma que el firewall está bloqueando el puerto 80 correctamente.

```
------------------------------------------------------------
Server listening on TCP port 80
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

Resultado del servidor (*iperf80-server.log*)

El servidor muestra que está escuchando correctamente en el puerto TCP 80. Esto indica que *iperf* inició sin problemas y que el host está listo para aceptar conexiones. El hecho de que no aparezcan conexiones entrantes es coherente con la prueba: el firewall está bloqueando los paquetes de h1 hacia h3 en el puerto 80, por lo que el servidor nunca llega a recibir los SYN del cliente.

En Wireshark también podemos apreciar esto:



12	89.490353986	10.0.0.1	10.0.0.3	TCP	74 40934 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1
13	90.495659073	10.0.0.1	10.0.0.3	TCP	74 [TCP Retransmission] 40934 → 80 [SYN] Seq=0
14	91.519545614	10.0.0.1	10.0.0.3	TCP	74 [TCP Retransmission] 40934 → 80 [SYN] Seq=0
15	92.543640894	10.0.0.1	10.0.0.3	TCP	74 [TCP Retransmission] 40934 → 80 [SYN] Seq=0
16	93.567648475	10.0.0.1	10.0.0.3	TCP	74 [TCP Retransmission] 40934 → 80 [SYN] Seq=0
17	94.591681917	10.0.0.1	10.0.0.3	TCP	74 [TCP Retransmission] 40934 → 80 [SYN] Seq=0
18	96.639673883	10.0.0.1	10.0.0.3	TCP	74 [TCP Retransmission] 40934 → 80 [SYN] Seq=0
19	100.671671478	10.0.0.1	10.0.0.3	TCP	74 [TCP Retransmission] 40934 → 80 [SYN] Seq=0
20	109.055547884	10.0.0.1	10.0.0.3	TCP	74 [TCP Retransmission] 40934 → 80 [SYN] Seq=0
21	114.688243666	fe80::88d8:90ff:fef…	ff02::2	ICMPv6	70 Router Solicitation from 8a:d8:90:fb:aa:04
22	125.439622785	10.0.0.1	10.0.0.3	TCP	74 [TCP Retransmission] 40934 → 80 [SYN] Seq=0

El host 1 intenta iniciar una comunicación TCP con el host 3 con un paquete SYN y al no recibir respuesta empieza a retransmitir el mismo paquete en reiteradas ocasiones, hasta que deja de intentar y lanza el error de conexión fallida mostrado antes

Prueba III: Bloqueo UDP puerto 5001 desde h1 con iperf

```
------------------------------------------------------------
Client connecting to 10.0.0.3, UDP port 5001
Sending 1470 byte datagrams, IPG target: 2243.04 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 34411 connected with 10.0.0.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec   640 KBytes  5.24 Mbits/sec
[  1] 1.0000-2.0000 sec   640 KBytes  5.24 Mbits/sec
[  1] 0.0000-2.0036 sec  1.25 MBytes  5.25 Mbits/sec
[  1] Sent 895 datagrams
[  3] WARNING: did not receive ack of last datagram after 10 tries.
```

Resultado del cliente (*iperf5001-client.log*)

El cliente muestra que pudo enviar datagramas UDP hacia 10.0.0.3:5001, pero al final advierte que no se recibio el *ack* del último datagrama.
Esto ocurre porque el firewall está aplicando la regla que **descarta (DROP)** los paquetes UDP con puerto destino 5001. A diferencia de TCP, UDP no establece conexiones, pero *iperf* utiliza paquetes de control para estimar pérdida; como ninguno llega al servidor, no recibe ningún ACK de retorno. Por eso el cliente sigue enviando datagramas pero finalmente reporta la advertencia. Este comportamiento confirma que el tráfico UDP fue bloqueado correctamente.

```
------------------------------------------------------------
Server listening on UDP port 5001
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
```

Resultado del servidor (*iperf5001-server.log*)

Como podemos ver no se registró ningún tráfico entrante, lo cual es coherente con la regla del firewall. Aunque h1 envió datagramas, todos fueron descartados antes de llegar a h3, por lo que el servidor nunca recibe paquetes para mostrar. La ausencia total de tráfico es, justamente, la evidencia de que la política de bloqueo se está aplicando correctamente.

En Wireshark también podemos apreciar esto:

368	3.918142674	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
369	3.928441553	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
370	3.938592261	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
371	3.948731687	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
372	3.958864582	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
373	3.969146937	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
374	3.979275930	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
375	3.989523018	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
376	3.999801013	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
377	4.010094488	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
378	4.020233270	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
379	4.030520241	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
380	4.040653145	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
381	4.050848666	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
382	4.060975344	10.0.0.1	10.0.0.3	UDP	1512 40598 → 5001 Len=1470
383	5.127837592	22:e6:71:10:6e:a6	d2:67:f6:37:62:32	ARP	42 Who has 10.0.0.3? Tell 10.0.0.1
384	5.129045372	d2:67:f6:37:62:32	22:e6:71:10:6e:a6	ARP	42 10.0.0.3 is at d2:67:f6:37:62:32

El host 1 envía 382 paquetes UDP en total al host 3 y luego de enviar el último no recibe ningún tipo de respuesta ya que esos paquetes fueron dropeados. Luego de esto deja de seguir enviando y devuelve la advertencia mostrada antes ya que ninguno de los paquetes enviados recibió un ACK como respuesta

Prueba IV: Bloqueo h2 ↔ h3 con iperf

```
------------------------------------------------------------
Client connecting to 10.0.0.3, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
tcp connect failed: No route to host
[  1] local 0.0.0.0 port 0 connected with 10.0.0.3 port 5001
```

h2 → h3 (cliente: iperf-h2-client.log)

El cliente intentó iniciar el handshake TCP (envió SYN) pero no recibió respuesta. *iperf* reporta “No route to host” porque la falta de respuesta tras los intentos se interpreta así; esto es **esperado** cuando el firewall DROPea los SYN.

```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

h3 (servidor para h2→h3: iperf-h2-server.log)

El servidor está escuchando correctamente en 5001 pero **no registra conexiones entrantes**, lo que concuerda con que todos los SYN desde h2 fueron descartados antes de llegar.

```
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
tcp connect failed: No route to host
[  1] local 0.0.0.0 port 0 connected with 10.0.0.2 port 5001
```

h3 → h2 (cliente: iperf-h3-client.log)

En Wireshark también podemos apreciar esto:

3	0.000898384	10.0.0.2	10.0.0.3	TCP	74 38170 → 5001 [SYN] Seq=0 Win=42340 Len=0 MSS=1
4	1.032557960	10.0.0.2	10.0.0.3	TCP	74 [TCP Retransmission] 38170 → 5001 [SYN] Seq=0
5	2.056588059	10.0.0.2	10.0.0.3	TCP	74 [TCP Retransmission] 38170 → 5001 [SYN] Seq=0
6	3.080571479	10.0.0.2	10.0.0.3	TCP	74 [TCP Retransmission] 38170 → 5001 [SYN] Seq=0
7	4.104667734	10.0.0.2	10.0.0.3	TCP	74 [TCP Retransmission] 38170 → 5001 [SYN] Seq=0
8	5.128676529	10.0.0.2	10.0.0.3	TCP	74 [TCP Retransmission] 38170 → 5001 [SYN] Seq=0
9	7.176617702	10.0.0.2	10.0.0.3	TCP	74 [TCP Retransmission] 38170 → 5001 [SYN] Seq=0
10	11.208557846	10.0.0.2	10.0.0.3	TCP	74 [TCP Retransmission] 38170 → 5001 [SYN] Seq=0
11	19.528628504	10.0.0.2	10.0.0.3	TCP	74 [TCP Retransmission] 38170 → 5001 [SYN] Seq=0
12	24.648636323	ea:72:f2:8c:a0:a2	d2:67:f6:37:62:32	ARP	42 Who has 10.0.0.3? Tell 10.0.0.2
13	24.649743131	d2:67:f6:37:62:32	ea:72:f2:8c:a0:a2	ARP	42 10.0.0.3 is at d2:67:f6:37:62:32
14	35.912626294	10.0.0.2	10.0.0.3	TCP	74 [TCP Retransmission] 38170 → 5001 [SYN] Seq=0
15	68.168558171	10.0.0.2	10.0.0.3	TCP	74 [TCP Retransmission] 38170 → 5001 [SYN] Seq=0
16	73.288667471	ea:72:f2:8c:a0:a2	d2:67:f6:37:62:32	ARP	42 Who has 10.0.0.3? Tell 10.0.0.2
17	73.289263851	d2:67:f6:37:62:32	ea:72:f2:8c:a0:a2	ARP	42 10.0.0.3 is at d2:67:f6:37:62:32

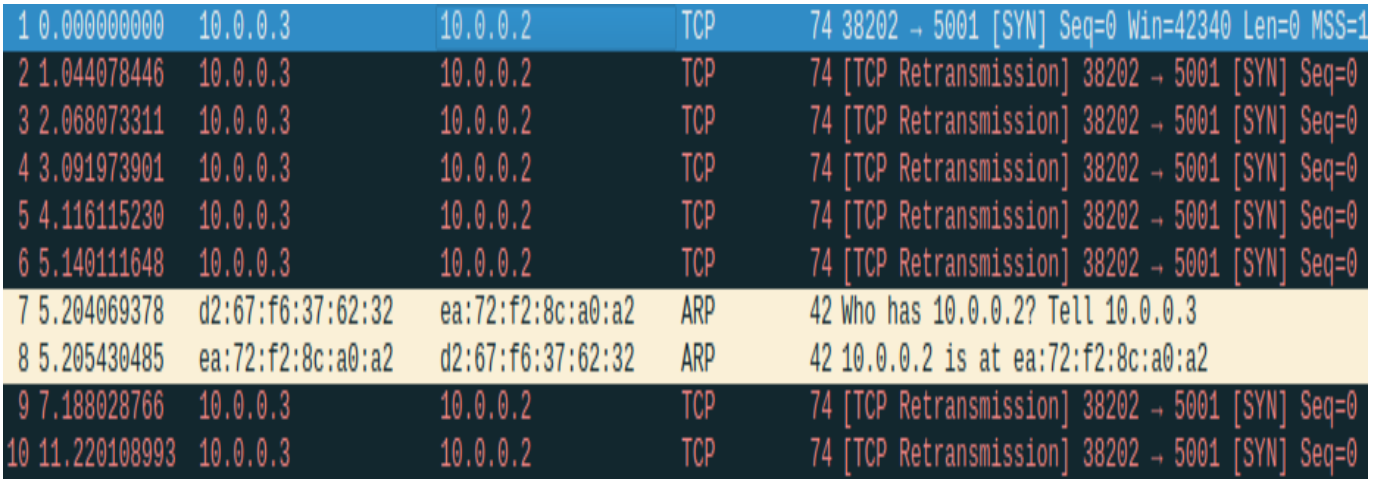
El host 2 intenta iniciar una comunicación TCP con el host 3 con un paquete SYN y al no recibir respuesta empieza a retransmitir el mismo paquete en reiteradas ocasiones, hasta que deja de intentar y lanza el error de conexión fallida mostrado antes

Mismo caso en el sentido inverso: h3 envía SYN hacia h2, no recibe respuesta y *iperf* falla. Esto demuestra que el bloqueo es bidireccional.

```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

h2 está listo para aceptar conexiones pero **no recibe tráfico**, lo que confirma que los paquetes de h3 se están descartando.

En Wireshark también podemos apreciar esto:



1	0.000000000	10.0.0.3	10.0.0.2	TCP	74 38202 → 5001 [SYN] Seq=0 Win=42340 Len=0 MSS=1
2	1.044078446	10.0.0.3	10.0.0.2	TCP	74 [TCP Retransmission] 38202 → 5001 [SYN] Seq=0
3	2.068073311	10.0.0.3	10.0.0.2	TCP	74 [TCP Retransmission] 38202 → 5001 [SYN] Seq=0
4	3.091973901	10.0.0.3	10.0.0.2	TCP	74 [TCP Retransmission] 38202 → 5001 [SYN] Seq=0
5	4.116115230	10.0.0.3	10.0.0.2	TCP	74 [TCP Retransmission] 38202 → 5001 [SYN] Seq=0
6	5.140111648	10.0.0.3	10.0.0.2	TCP	74 [TCP Retransmission] 38202 → 5001 [SYN] Seq=0
7	5.204069378	d2:67:f6:37:62:32	ea:72:f2:8c:a0:a2	ARP	42 Who has 10.0.0.2? Tell 10.0.0.3
8	5.205430485	ea:72:f2:8c:a0:a2	d2:67:f6:37:62:32	ARP	42 10.0.0.2 is at ea:72:f2:8c:a0:a2
9	7.188028766	10.0.0.3	10.0.0.2	TCP	74 [TCP Retransmission] 38202 → 5001 [SYN] Seq=0
10	11.220108993	10.0.0.3	10.0.0.2	TCP	74 [TCP Retransmission] 38202 → 5001 [SYN] Seq=0

El host 3 intenta iniciar una comunicación TCP con el host 2 con un paquete SYN y al no recibir respuesta empieza a retransmitir el mismo paquete en reiteradas ocasiones, hasta que deja de intentar y lanza el error de conexión fallida mostrado antes

En resumen, los cuatro logs de la terminal muestran el patrón esperado de un **DROP**: ambos servidores escuchan, ambos clientes intentan y fallan sin respuesta. Por lo que es una prueba satisfactoria de la regla 3 (bloqueo bidireccional entre 10.0.0.2 ↔ 10.0.0.3).

Preguntas

1) ¿Cuál es la diferencia entre un Switch y un router? ¿Qué tienen en común?

Fundamentalmente, tanto el switch como el router son dispositivos de conmutación de paquetes que comparten la funcionalidad de almacenamiento y reenvío, utilizando tablas de reenvío internas para dirigir el tráfico hacia su destino.

La diferencia crítica entre ambos reside en la capa jerárquica en la que ejecutan esta función: el switch opera en la capa de enlace de datos, procesando tramas y tomando decisiones basadas en direcciones físicas (MAC) para interconectar dispositivos de forma transparente dentro de una misma red local mediante mecanismos de autoaprendizaje.

Por el contrario, el router opera en la capa de red, procesando datagramas y basando sus decisiones en direcciones lógicas (IP) para interconectar redes distintas; a diferencia del switch, el router utiliza algoritmos de enrutamiento para determinar la ruta óptima a través de la red y tiene la capacidad de aislar dominios de difusión (broadcast), re-encapsulando los paquetes en cada salto hacia su destino final.

2) ¿Cuál es la diferencia entre un Switch convencional y un Switch OpenFlow?

La diferencia sustancial entre ambos dispositivos reside en la arquitectura de sus planos funcionales y en la flexibilidad de sus decisiones de reenvío. Un switch convencional opera bajo una arquitectura monolítica donde el plano de control y el plano de datos están integrados verticalmente en el mismo hardware, limitando su función a un reenvío basado exclusivamente en la dirección MAC de destino mediante algoritmos fijos y distribuidos.

En contraste, un switch OpenFlow materializa el paradigma de las Redes Definidas por Software (SDN) al desacoplar estos planos: delega la lógica de control a un controlador remoto centralizado y conserva únicamente el plano de datos. Esto permite transitar de un reenvío rígido a un "reenvío generalizado", donde el dispositivo actúa en base a tablas de flujo configuradas externamente que utilizan un esquema de "coincidencia y acción" (match-plus-action) sobre múltiples campos de las cabeceras (MAC, IP, puertos), permitiendo que el switch emule comportamientos de router, firewall o balanceador de carga según se programe.

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

A simple vista, no se podría reemplazar todos los routers por switches OpenFlow debido a la interacción entre sistemas autónomos (AS). La interacción entre sistemas autónomos utiliza mecanismos como BGP, el cual maneja atributos más complejos relacionados con relaciones comerciales. Este tipo de filtros no podría ser aplicable nunca en los switches OpenFlow.

Además, este tipo de switches requiere de un controlador SDN centralizado, con la cantidad de rutas existentes, que un solo equipo maneje toda esa información en tiempo real es casi imposible y generaría un cuello de botella provocando poca tolerancia a fallos.

README

El README contiene la documentación completa sobre la estructura del proyecto, la configuración del firewall mediante reglas definidas en JSON, la topología de red con múltiples switches y hosts, y un conjunto de scripts para ejecutar y probar automáticamente las diferentes reglas de bloqueo implementadas.

Scripts Principales

1. run_pox.sh - Iniciar Controlador SDN

```
./run_pox.sh
```

Resultado esperado: El controlador POX se inicia con el módulo firewall. Muestra logs de depuración y queda en ejecución esperando conexiones de switches.

2. run_mininet.sh - Iniciar Topología de Red

```
./run_mininet.sh [num_switches]
```

Parámetro: num_switches (opcional, por defecto: 2)
Resultado esperado: Se crea la topología con N switches y 4 hosts (h1-h4). Muestra el prompt mininet> para ejecutar comandos interactivos.

Scripts de Pruebas Automatizadas

3. test_pingall.sh - Prueba de Conectividad General

```
sudo ./test_pingall.sh [num_switches]
```

Resultado esperado: Ejecuta pingall y muestra la tabla de conectividad. Con las reglas actuales, solo h2↔h3 deben estar bloqueados. Mininet se cierra automáticamente al finalizar.

4. test_block_port80.sh - Verificar Bloqueo Puerto 80

```
sudo ./test_block_port80.sh [num_switches]
```

Resultado esperado: Muestra OK: puerto 80 bloqueado y el exit code de iperf (≠0). Confirma que el tráfico TCP al puerto 80 está bloqueado.

5. test_block_udp5001.sh - Verificar Bloqueo UDP desde h1

```
sudo ./test_block_udp5001.sh [num_switches]
```

Resultado esperado: Muestra OK: UDP 5001 bloqueado (no se registró tráfico). Confirma que el tráfico UDP desde h1 al puerto 5001 está bloqueado.

6. test_block_hosts23.sh - Verificar Bloqueo Bidireccional h2↔h3

```
sudo ./test_block_hosts23.sh [num_switches]
```

Resultado esperado: Muestra OK: h2 -> h3 bloqueado y OK: h3 -> h2 bloqueado. Confirma que la comunicación bidireccional entre h2 y h3 está bloqueada.

Parámetros Opcionales

- num_switches: Cantidad de switches en la topología (mínimo 2)
- WAIT_BEFORE: Variable de entorno para esperar antes de generar tráfico (por defecto: 5 segundos)

```
WAIT_BEFORE=10 sudo ./test_block_port80.sh
```

Dificultades encontradas

Durante el desarrollo del trabajo se presentaron algunas dificultades principalmente asociadas a la integración de las distintas herramientas. La generación de la topología requirió de varias iteraciones ya que al principio se malinterpretó lo que se pedía. También fue necesario depurar la lógica de *L2 learning* en POX, ya que el controlador debía aprender direcciones MAC manualmente y evitar reenvíos incorrectos.

La implementación del firewall presentó desafíos adicionales, especialmente al traducir las reglas del archivo JSON a coincidencias OpenFlow válidas. Campos como protocolo, puertos y direcciones IP requerían configuraciones explícitas, y algunas reglas exigieron generar múltiples matches. Finalmente, las pruebas con *pingall*, *iperf* y *Wireshark* mostraron problemas de sincronización entre Mininet y POX, lo que obligó a reiniciar procesos y verificar el correcto orden de ejecución para asegurar que las reglas funcionaran adecuadamente.

Conclusión

Este trabajo nos permitió comprender en mayor profundidad el funcionamiento de las Software Defined Networks (SDN), así como también la estructura y lógica detrás del protocolo OpenFlow. A través de la implementación de una topología personalizada y el desarrollo de un firewall, pudimos experimentar cómo se aplican reglas de red de forma dinámica y centralizada.

En resumen, el trabajo práctico nos brindó una visión práctica de cómo las redes pueden ser gestionadas mediante software, y cómo tecnologías como OpenFlow permiten aplicar políticas de seguridad y control.