

**MÁSTER EN INGENIERÍA INFORMÁTICA
TECNOLOGÍAS Y PROTOCOLOS DE COMUNICACIÓN**



Escuela Politécnica

PRÁCTICA 4

ROUTER SDN/IP

**LUCAS BONILLA RODRÍGUEZ
JOSÉ JAVIER MORENO GONZÁLEZ**

2022-2023

Índice

1. Introducción	1
2. Creación de la topología	2
3. Creación del controlador POX	2
4. Instalación de reglas OVS	7
5. Pruebas del funcionamiento del controlador	7

1. Introducción

El objetivo de esta práctica consiste en crear un switch estático de capa 3 mediante el software de red POX. No se comportará en su totalidad como un switch debido a que no decrementa el IP TTL ni recalcula la suma de verificación en cada salto.

El router será completamente estático, sin BGP ni OSPF. En caso de que un paquete se encuentre destinado para un host dentro de esa subred el nodo actúa como un conmutador y reenvía el paquete sin cambios y en caso de que el paquete se encuentre destinado a una IP de la cual no se conoce el siguiente salto se debe modificar el destino y reenviar el paquete al puerto correspondiente.

La escenario en Mininet que se debe seguir para comprobar el funcionamiento de la práctica es el siguiente:

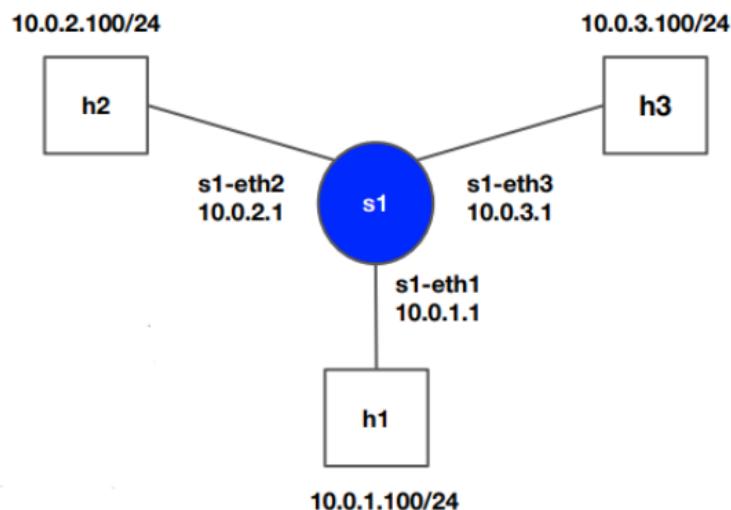


Fig. 1. Escenario de la práctica.

2. Creación de la topología

En primer lugar, es necesario crear un script de python para generar la topología descrita en la figura 1. Para ello se declaran en el script todos los hosts y switches necesarios con sus respectivas IPs y direcciones MAC.

```

class LeafSpine( Topo ):
    def __init__( self ):
        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        host1 = self.addHost( 'h1' , ip='10.0.1.100/24',mac='00:00:00:00:00:01',defaultRoute="via 10.0.1.1")
        host2 = self.addHost('h2', ip='10.0.2.100/24',mac='00:00:00:00:00:02',defaultRoute="via 10.0.2.1")
        host3 = self.addHost('h3', ip='10.0.3.100/24',mac='00:00:00:00:00:03',defaultRoute="via 10.0.3.1")

        centralSwitch = self.addSwitch('s1')
        # Add links

        self.addLink( host1, centralSwitch )
        self.addLink( host2, centralSwitch )
        self.addLink( host3, centralSwitch )

topos = { 'mytopo': ( lambda: LeafSpine() ) }

```

Fig. 2. Código fuente de creación de la topología.

Cómo se puede apreciar en la figura 2 se crean un switch central y 3 hosts, cada uno con la dirección IP de su subred, su MAC y la ruta que sigue hacia el switch. Tras crearlo, se establecen los enlaces para fijar las conexiones entre las redes.

El host 1 se encuentra enlazado con el switch por el puerto 1 de este.

El host 2 se encuentra enlazado con el switch por el puerto 2 de este.

El host 3 se encuentra enlazado con el switch por el puerto 3 de este.

3. Creación del controlador POX

Tras crear la topología solicitada, se requiere establecer un controlador que gestione todos los paquetes que circulen entre las redes de la topología. Mediante el software de POX se puede crear un controlador que cumpla las condiciones para actuar cómo el switch solicitado.

En primer lugar, se crean una serie de variables tipo diccionario que sirven para almacenar correspondencia entre direcciones MACs, direcciones IPs y puertos.

`mac_to_port`: almacena las direcciones MACs de los dispositivos y el puerto del switch correspondiente a cada uno de ellos.

`switch_interfaces`: almacena la correspondencia entre las direcciones IPs, MACs y puertos de las interfaces del switch.

`ip_to_mac`: almacena las direcciones MACs de los dispositivos y su correspondencia con su dirección IP.

Además, la variable `buffer` se crea para almacenar los paquetes IP mandados por los dispositivos, pero el switch no conoce todavía el puerto de salida del mismo. Por ende, este paquete debe ser enviado y no desecharse al conocer esa dirección.

```

self.mac_to_port = {}
self.buffer = []

self.switch_interfaces = {'10.0.1.1':['00:00:00:00:00:11',1],
                           '10.0.2.1':['00:00:00:00:00:22',2],
                           '10.0.3.1':['00:00:00:00:00:33',3]}

self.ip_to_mac = {}

self.switch_ports = {'00:00:00:00:00:11':1,'00:00:00:00:00:22':2,'00:00:00:00:00:33':3}

```

Fig. 3. Creación de diccionarios y buffer de paquetes.

Por otro lado, hay que permitir al controlador encontrar las direcciones MAC que correspondan a las determinadas direcciones IP a las que se quieran enviar los paquetes, es decir, permitir que el router utilice el protocolo ARP. El router recibe peticiones ARP Request de los hosts que buscan conocer la dirección ethernet que corresponde y el controlador es el encargado en este caso de construir los ARP Reply y reenviarlos al puerto apropiado.

Independientemente de si el paquete que llega al controlador es de tipo IP o de tipo ARP se almacena en un diccionario la MAC del origen y el puerto por el que se debe redireccionar.

```

#Entry packet is ARP or IPv4
if packet.type == packet.ARP_TYPE or packet.type == packet.IP_TYPE:

    #Add to memory the port of the device specifies by the MAC of the last hosts' packet
    sourceAddr = str(packet.src)
    inputPort = packet_in.in_port
    self.mac_to_port[sourceAddr] = inputPort

```

Fig. 4. Asignación de puertos a la MAC del paquete entrante.

Se comprueba que llega un ARP Request y en el caso de que esto se cumpla se crea un ARP Reply para responder al host que realiza el ARP Request.

```

self.ip_to_mac[str(packet.payload.protosrc)]=str(packet.src)

```

Fig. 5. Almacenamiento de IP y MAC correspondientes al paquete entrante.

Previo a la creación del Reply, se almacena la información relativa a la IP y la MAC del host que envía el paquete.

Se crea el ARP Reply con destino el host que había realizado previamente el Request y origen el propio servidor que es el que se encarga de enviar este ARP Reply. Se empaqueta en un Ethernet y se envía el Ethernet por el puerto indicado.

```
arp_reply = pkt.arp()
arp_reply.hwsrc = pkt.EthAddr(self.switch_interfaces[str(packet.payload.protodst)][0]) #funciona
arp_reply.hwdst = packet.src
arp_reply.opcode = pkt.arp.REPLY
arp_reply.protosrc = packet.payload.protodst
arp_reply.protodst = packet.payload.protosrc

ether = pkt.ethernet()
ether.type = pkt.ethernet.ARP_TYPE
ether.dst = packet.src
ether.src = pkt.EthAddr(self.switch_interfaces[str(packet.payload.protodst)][0]) #funciona
ether.payload = arp_reply

self.resend_packet(ether, out_port=self.switch_interfaces[str(packet.payload.protodst)][1])
```

Fig. 6. Creación de ARP Reply.

Este caso se contempla al recibir ARP de los hosts, los cuales son respondidos por el switch ya que los paquetes ARP no pueden salir de su propia red. El switch responde enviando un paquete a la dirección IP fuente con su propia dirección IP y su MAC asociada. De esta forma el host conoce la dirección MAC de la interfaz de su switch de salida.

En el caso de que llegue un paquete IP, el proceso a seguir es similar al anterior cuando se recibe un paquete ARP.

En primer lugar, se explica el comportamiento si el paquete IP llega con destino a una interfaz del switch. Como el switch no responde por sí solo se debe crear este paquete de ICMP Echo Reply y enviarlo al host.

Se crea un paquete ICMP Echo Reply que va a responder a los ICMP Echo Request que lleguen a la interfaz del router. Este se crea con el tipo y código correspondientes a un Echo Reply y con el payload del paquete que le llega del source.

Este paquete ICMP se empaqueta dentro de un paquete IP con la dirección de origen del router y con destino la dirección del host. Para poder reenviar el paquete correctamente se crea otra vez un paquete Ethernet que contiene al paquete IP y se reenvía por el puerto indicado.

```

#IP packet with ICMP reply resend the to the host
icmp = pkt.icmp()
icmp.type = pkt.TYPE_ECHO_REPLY #ECHO REPLY
icmp.code = pkt.TYPE_ECHO_REPLY
icmp.payload = packet.payload.payload.payload

ip = pkt.ipv4()
ip.protocol = packet.payload.protocol
ip.srcip = packet.payload.dstip
ip.dstip = packet.payload.srcip
ip.payload = icmp

ether = pkt.ethernet()
ether.type = pkt.ethernet.IP_TYPE
ether.payload = ip
ether.src = packet.dst
ether.dst = packet.src

print(self.switch_interfaces[str(packet.payload.dstip)][1])
self.resend_packet(ether, out_port=self.switch_interfaces[str(packet.payload.dstip)][1])

```

Fig. 7. Paquete repuesta de la interfaz del switch.

De esta manera, con este código el controlador POX ya puede manejar y redirigir todos los paquetes que pasan a través de una interfaz del switch, pero existe un problema y es que cuando el controlador POX tiene que mandar un paquete a una dirección que no se encuentra dentro del diccionario el primer paquete que se envía desde el host que inicia la comunicación hasta el host desconocido se pierde.

Con el fin de solucionar este problema es necesario utilización del buffer que almacena el paquete IP que se envía hacia un host desconocido para poder reenviarlo una vez el controlador conoce el destino. Una vez almacenado el switch crea un ARP Broadcast para obtener la dirección MAC correspondiente

a la dirección IP del paquete almacenado en el buffer y de esta manera reenviarlo.

El paquete IP que llega se almacena en un buffer para su posterior reenvío y se construye un paquete de tipo ARP Request con destino broadcast y dirección IP el host desconocido, es decir, se envía este paquete a todos los demás hosts que se encuentren conectados al switch y aquel host con el que coincida la dirección IP responderá con un paquete ARP_Reply que contendrá la dirección MAC y el puerto por el que dirigirse a dicho host.

Una vez llegue el paquete ARP_Reply al host se almacena la información cómo se ha comentado anteriormente y ahora es necesario comprobar si el buffer contiene información, debido a que si el buffer contiene información significa que hay que reenviar el paquete a su respectivo destino.

```

#IP packet send to an unknown host known
print("Broadcast and store IP packet")
self.buffer.append(packet)
print(self.buffer[-1])

arp_request = pkt.arp()
arp_request.hwsrc = packet.dst
arp_request.hwdst = pkt.EthAddr.BROADCAST
arp_request.opcode = pkt.arp.REQUEST
arp_request.protosrc = packet.payload.srcip
arp_request.protodst = packet.payload.dstip
ether = pkt.ethernet()
ether.type = pkt.ethernet.ARP_TYPE
ether.dst = pkt.EthAddr.BROADCAST
ether.src = packet.dst
ether.payload = arp_request

self.resend_packet(ether, of.OFPP_ALL)

```

Fig. 8. Almacenamiento en buffer, creación y envío de broadcast.

Se realiza la comprobación del buffer para saber si contiene información y en caso de que contenga información y la IP del destino se encuentre registrada en el diccionario de direcciones IP se crea un paquete Ethernet que contendrá el paquete IP almacenado en el buffer. Este paquete Ethernet se reenvía al host indicado en el paquete IP del buffer y se elimina el paquete del buffer para evitar que vuelva a ser reenviado.

```

if len(self.buffer) > 0 and str(self.buffer[0].payload.dstip) in self.ip_to_mac.keys():

    ether = pkt.ethernet()
    ether.type = pkt.ethernet.IP_TYPE
    ether.payload = self.buffer[0].payload
    ether.src = packet.dst
    ether.dst = pkt.EthAddr(self.ip_to_mac[str(self.buffer[0].payload.dstip)])

    print(self.mac_to_port[self.ip_to_mac[str(self.buffer[0].payload.dstip)]])
    self.resend_packet(ether, out_port=self.mac_to_port[str(self.ip_to_mac[str(self.buffer[0].payload.dstip)]])
    self.buffer.pop()

```

Fig. 9. Envío del paquete almacenado en el buffer.

En último lugar, se tiene en cuenta que el paquete IP recibido al switch va con destino a un host conocido por el switch por lo que lo único necesario es crear un paquete ethernet

con el mismo paquete ip de payload e introducir la MAC origen del switch y la MAC destino del host destino.

```
elif str(packet.payload.dstip) in self.ip_to_mac.keys() and self.ip_to_mac[str(packet.payload.dstip)] in self.mac_to_port:

    #Resend to destination host changing MACs
    ether = pkt.ethernet()
    ether.type = pkt.ethernet.IP_TYPE
    ether.payload = packet.payload
    ether.src = packet.dst
    ether.dst = pkt.EthAddr(self.ip_to_mac[str(packet.payload.dstip)])
    print(ether)
    self.resend_packet(ether, out_port=self.mac_to_port[self.ip_to_mac[str(packet.payload.dstip)]])
```

Fig. 10. Reenvío de paquete a host conocido.

4. Instalación de reglas OVS

En esta sección se presenta la instalación de las reglas Openflow en el propio switch. Esto permite al switch saber el enrutamiento de paquetes de forma correcta sin tener que consultar al controlador POX. Estas consultas provocan retrasos temporales por lo que la instalación de reglas disminuye el tiempo de envío entre paquetes para destinos instalados en esta.

En la siguiente figura se muestra el código para insertar las reglas de enrutamiento en el switch. Esta regla realiza un cambio de direcciones MACs para que el switch sepa hacia qué puerto debe de sacar los paquetes sin consultar el controlador. El comportamiento de la regla es el mismo que el del controlador POX en esos casos, al enviar un paquete de un host a otro la MAC debe de ser cambiada por la del próximo salto, esto se realiza en la regla gracias a la dirección IP almacenada en el paquete y a los diccionarios con la información de IPs, MACs y puertos de los dispositivos e interfaces.

```
msg = of.ofp_flow_mod()
msg.match.dl_src = packet.src
msg.match.dl_dst = packet.dst
msg.match.nw_dst = packet.payload.dstip
msg.match.dl_type = 0x0800

port = None
port = self.mac_to_port[str(pkt.EthAddr(self.ip_to_mac[str(packet.payload.dstip)]))]
msg.actions.append(of.ofp_action_dl_addr.set_src(packet.dst))
msg.actions.append(of.ofp_action_dl_addr.set_dst(self.ip_to_mac[str(packet.payload.dstip)]))
msg.actions.append(of.ofp_action_output(port = port))
msg._validate()
print(msg)
self.connection.send(msg)
```

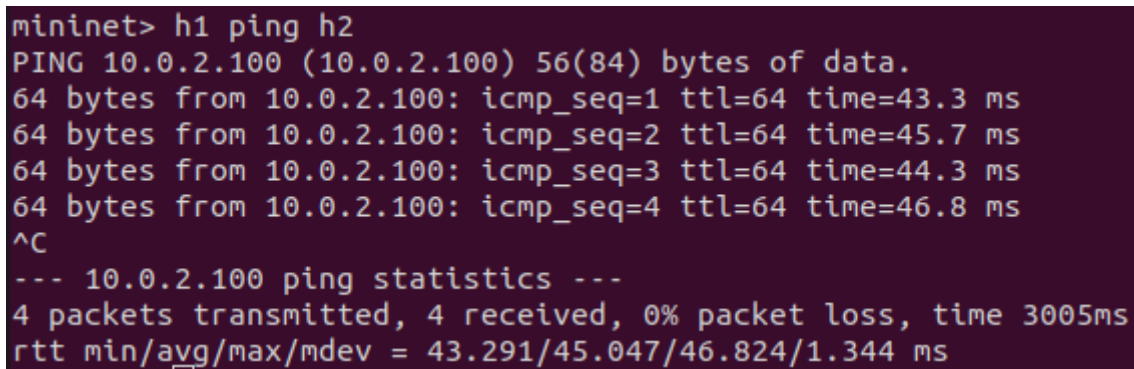
Fig. 11. Instalación de regla de enrutamiento OVS.

5. Pruebas del funcionamiento del controlador

Para comprobar el correcto funcionamiento del controlador POX se realizan una serie de pruebas que permiten determinar que el controlador funciona cómo se establece y permite las comunicaciones entre los diversos hosts que componen la topología de la red.

La primera prueba que se realiza es una prueba de conectividad entre los distintos hosts de las distintas subredes que componen la topología, en este caso, se envían paquetes con el comando ping entre los distintos hosts y se comprueba que llegan y responden cómo es debido.

Se realiza un ping entre los hosts H1 y H2, cómo se puede observar en la imagen no se pierde ninguno de los paquetes y mediante el comando tcpdump se observa el intercambio de respuestas ICMP Echo Request e ICMP Echo Reply entre ambos hosts.



```
mininet> h1 ping h2
PING 10.0.2.100 (10.0.2.100) 56(84) bytes of data.
64 bytes from 10.0.2.100: icmp_seq=1 ttl=64 time=43.3 ms
64 bytes from 10.0.2.100: icmp_seq=2 ttl=64 time=45.7 ms
64 bytes from 10.0.2.100: icmp_seq=3 ttl=64 time=44.3 ms
64 bytes from 10.0.2.100: icmp_seq=4 ttl=64 time=46.8 ms
^C
--- 10.0.2.100 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 43.291/45.047/46.824/1.344 ms
```

Fig. 12. Comunicación entre H1 y H2.

Antes la comunicación H1 ha mandado un ARP Request al switch preguntando por la MAC de la interfaz del switch la cual el switch desconoce. En la siguiente imagen se muestra la llegada de ese ARP Request al switch. Como se ha explicado anteriormente, el switch responde con su MAC al host y este envía el primer paquete ICMP para el H2. Al llegar este paquete al switch este desconoce la ubicación de H2 por lo que manda un broadcast el cual también se aprecia en la figura 12. El primer paquete ICMP se encuentra almacenado en el buffer esperando que el switch conozca la MAC de H2. Una vez llega el ARP Reply de H2 al switch este guarda su MAC y envía el paquete almacenado en el buffer. Seguidamente la comunicación se conforma como dos hosts conocidos para el switch.

```

length 16
22:54:12.174113 IP6 fe80::200:ff:fe00:1 > ip6-allrouters: ICMP6, router solicitation, length 16
22:54:13.966551 IP6 lucas-Lenovo.mdns > ff02::fb.mdns: 0 [3q] PTR (QM)? _pgpkey-hkp._tcp.local. PTR (QM)? _ipp._tcp.local. PTR (QM)? _ipps._tcp.local. (63)
22:54:19.770623 ARP, Request who-has 10.0.1.1 tell 10.0.1.100, length 28
22:54:19.772868 ARP, Reply 10.0.1.1 is-at 00:00:00:00:00:11 (oui Ethernet), length 28
22:54:19.772885 IP 10.0.1.100 > 10.0.2.100: ICMP echo request, id 4170, seq 1, length 64
22:54:19.775824 ARP, Request who-has 10.0.2.100 (Broadcast) tell 10.0.1.100, length 28
22:54:20.782282 IP 10.0.1.100 > 10.0.2.100: ICMP echo request, id 4170, seq 2, length 64
22:54:20.787613 IP 10.0.2.100 > 10.0.1.100: ICMP echo reply, id 4170, seq 2, length 64

```

Fig. 13. Paquete ARP Request preguntando por la interfaz del switch.

```

21:48:40.850289 IP 10.0.1.100 > 10.0.2.100: ICMP echo request, id 43734, seq 1, length 64
21:48:40.855333 IP 10.0.2.100 > 10.0.1.100: ICMP echo reply, id 43734, seq 1, length 64
21:48:41.851962 IP 10.0.1.100 > 10.0.2.100: ICMP echo request, id 43734, seq 2, length 64
21:48:41.897649 IP 10.0.2.100 > 10.0.1.100: ICMP echo reply, id 43734, seq 2, length 64
21:48:42.853323 IP 10.0.1.100 > 10.0.2.100: ICMP echo request, id 43734, seq 3, length 64
21:48:42.897605 IP 10.0.2.100 > 10.0.1.100: ICMP echo reply, id 43734, seq 3, length 64
21:48:43.854866 IP 10.0.1.100 > 10.0.2.100: ICMP echo request, id 43734, seq 4, length 64
21:48:43.901637 IP 10.0.2.100 > 10.0.1.100: ICMP echo reply, id 43734, seq 4, length 64
21:48:40.852439 IP 10.0.1.100 > 10.0.2.100: ICMP echo request, id 43734, seq 1, length 64
21:48:40.852496 IP 10.0.2.100 > 10.0.1.100: ICMP echo reply, id 43734, seq 1, length 64
21:48:41.854226 IP 10.0.1.100 > 10.0.2.100: ICMP echo request, id 43734, seq 2, length 64
21:48:41.854280 IP 10.0.2.100 > 10.0.1.100: ICMP echo reply, id 43734, seq 2, length 64
21:48:42.855610 IP 10.0.1.100 > 10.0.2.100: ICMP echo request, id 43734, seq 3, length 64
21:48:42.855674 IP 10.0.2.100 > 10.0.1.100: ICMP echo reply, id 43734, seq 3, length 64
21:48:43.857114 IP 10.0.1.100 > 10.0.2.100: ICMP echo request, id 43734, seq 4, length 64
21:48:43.857187 IP 10.0.2.100 > 10.0.1.100: ICMP echo reply, id 43734, seq 4, length 64

```

Fig. 14. Paquetes intercambiados en la comunicación entre H1 y H2.

Se realiza un ping entre los hosts h1 y h3, cómo se puede observar en la imagen no se pierde ninguno de los paquetes y mediante el comando tcpdump se observa el intercambio de respuestas ICMP Echo Request e ICMP Echo Reply entre ambos hosts.

```

mininet> h1 ping h3
PING 10.0.3.100 (10.0.3.100) 56(84) bytes of data.
64 bytes from 10.0.3.100: icmp_seq=1 ttl=64 time=53.3 ms
64 bytes from 10.0.3.100: icmp_seq=2 ttl=64 time=45.9 ms
64 bytes from 10.0.3.100: icmp_seq=3 ttl=64 time=44.7 ms
64 bytes from 10.0.3.100: icmp_seq=4 ttl=64 time=42.8 ms
^C
--- 10.0.3.100 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 42.839/46.677/53.266/3.956 ms

```

Fig. 15. Comunicación entre H1 y H3.

```

# tcpdump -i eth0 -s 1500 -n -v -e -A -B 1024 -C 1024 -F 10.0.1.100 > 10.0.3.100: ICMP echo request, id 43971, seq 1, length 64
21:57:14.318184 IP 10.0.1.100 > 10.0.3.100: ICMP echo request, id 43971, seq 1, length 64
21:57:14.365490 IP 10.0.3.100 > 10.0.1.100: ICMP echo reply, id 43971, seq 1, length 64
21:57:15.319959 IP 10.0.1.100 > 10.0.3.100: ICMP echo request, id 43971, seq 2, length 64
21:57:15.365689 IP 10.0.3.100 > 10.0.1.100: ICMP echo reply, id 43971, seq 2, length 64
21:57:16.322067 IP 10.0.1.100 > 10.0.3.100: ICMP echo request, id 43971, seq 3, length 64
21:57:16.365646 IP 10.0.3.100 > 10.0.1.100: ICMP echo reply, id 43971, seq 3, length 64
21:57:17.323859 IP 10.0.1.100 > 10.0.3.100: ICMP echo request, id 43971, seq 4, length 64
21:57:17.373634 IP 10.0.3.100 > 10.0.1.100: ICMP echo reply, id 43971, seq 4, length 64
21:57:19.321258 ARP, Request who-has 10.0.1.1 tell 10.0.1.100, length 28
21:57:14.320037 IP 10.0.1.100 > 10.0.3.100: ICMP echo request, id 43971, seq 1, length 64
21:57:14.320093 IP 10.0.3.100 > 10.0.1.100: ICMP echo reply, id 43971, seq 1, length 64
21:57:15.321989 IP 10.0.1.100 > 10.0.3.100: ICMP echo request, id 43971, seq 2, length 64
21:57:15.322042 IP 10.0.3.100 > 10.0.1.100: ICMP echo reply, id 43971, seq 2, length 64
21:57:16.324339 IP 10.0.1.100 > 10.0.3.100: ICMP echo request, id 43971, seq 3, length 64
21:57:16.324400 IP 10.0.3.100 > 10.0.1.100: ICMP echo reply, id 43971, seq 3, length 64
21:57:17.325749 IP 10.0.1.100 > 10.0.3.100: ICMP echo request, id 43971, seq 4, length 64
21:57:17.325800 IP 10.0.3.100 > 10.0.1.100: ICMP echo reply, id 43971, seq 4, length 64
21:57:19.321258 ARP, Request who-has 10.0.3.1 tell 10.0.3.100, length 28
21:57:14.322459 ARP, Request who-has 10.0.3.1 tell 10.0.3.100, length 28

```

Fig. 16. Paquetes intercambiados en la comunicación entre H1 y H3.

Se realiza un ping entre los hosts h3 y h1, cómo se puede observar en la imagen no se pierde ninguno de los paquetes y mediante el comando tcpdump se observa el intercambio de respuestas ICMP Echo Request e ICMP Echo Reply entre ambos hosts.

```
mininet> h3 ping h1
PING 10.0.1.100 (10.0.1.100) 56(84) bytes of data.
64 bytes from 10.0.1.100: icmp_seq=1 ttl=64 time=43.8 ms
64 bytes from 10.0.1.100: icmp_seq=2 ttl=64 time=46.0 ms
64 bytes from 10.0.1.100: icmp_seq=3 ttl=64 time=44.7 ms
64 bytes from 10.0.1.100: icmp_seq=4 ttl=64 time=46.8 ms
^C
--- 10.0.1.100 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 43.827/45.315/46.760/1.124 ms
```

Fig. 17. Comunicación entre H3 y H1.

```
21:53:19.059783 IP 10.0.3.100 > 10.0.1.100: ICMP echo request, id 43917, seq 1, length 64
21:53:19.059847 IP 10.0.1.100 > 10.0.3.100: ICMP echo reply, id 43917, seq 1, length 64
21:53:20.062048 IP 10.0.3.100 > 10.0.1.100: ICMP echo request, id 43917, seq 2, length 64
21:53:20.062113 IP 10.0.1.100 > 10.0.3.100: ICMP echo reply, id 43917, seq 2, length 64
21:53:21.063344 IP 10.0.3.100 > 10.0.1.100: ICMP echo request, id 43917, seq 3, length 64
21:53:21.063398 IP 10.0.1.100 > 10.0.3.100: ICMP echo reply, id 43917, seq 3, length 64
21:53:22.064923 IP 10.0.3.100 > 10.0.1.100: ICMP echo request, id 43917, seq 4, length 64
21:53:22.064992 IP 10.0.1.100 > 10.0.3.100: ICMP echo reply, id 43917, seq 4, length 64
21:53:24.313259 ARP, Request who-has 10.0.1.1 tell 10.0.1.100, length 28
21:53:19.057691 IP 10.0.3.100 > 10.0.1.100: ICMP echo request, id 43917, seq 1, length 64
21:53:19.101471 IP 10.0.1.100 > 10.0.3.100: ICMP echo reply, id 43917, seq 1, length 64
21:53:20.059738 IP 10.0.3.100 > 10.0.1.100: ICMP echo request, id 43917, seq 2, length 64
21:53:20.105635 IP 10.0.1.100 > 10.0.3.100: ICMP echo reply, id 43917, seq 2, length 64
21:53:21.061005 IP 10.0.3.100 > 10.0.1.100: ICMP echo request, id 43917, seq 3, length 64
21:53:21.105686 IP 10.0.1.100 > 10.0.3.100: ICMP echo reply, id 43917, seq 3, length 64
21:53:22.062909 IP 10.0.3.100 > 10.0.1.100: ICMP echo request, id 43917, seq 4, length 64
21:53:22.109620 IP 10.0.1.100 > 10.0.3.100: ICMP echo reply, id 43917, seq 4, length 64
21:53:24.313259 ARP, Request who-has 10.0.3.1 tell 10.0.3.100, length 28
```

Fig. 18. Paquetes intercambiados en la comunicación entre H3 y H1.

Tras la realización de estas pruebas de intercambios de paquetes entre hosts, se puede determinar que el router funciona correctamente con respecto a la comunicación entre dispositivos de diferentes subredes.

La siguiente prueba consiste en comprobar que se puede hacer ping a las interfaces del router. Se realiza un ping de h1 a la interfaz del switch y cómo se puede observar en la imagen los paquetes llegan correctamente.

Mediante el comando tcpdump se observa que el intercambio de paquetes entre el hosts h1 y la interfaz del switch es correcto, ya que el host envía el ICMP Request al switch y este responde a su petición con un ICMP Reply.

```

root@iepc-HP-ProBook-4510s:/home/usuario/pox/pox/practica4TPC# ping 10.0.1.1
PING 10.0.1.1 (10.0.1.1) 56(84) bytes of data.
64 bytes from 10.0.1.1: icmp_seq=1 ttl=64 time=4.61 ms
64 bytes from 10.0.1.1: icmp_seq=2 ttl=64 time=2.19 ms
64 bytes from 10.0.1.1: icmp_seq=3 ttl=64 time=2.15 ms
^C
--- 10.0.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 2.148/2.982/4.611/1.151 ms

```

Fig. 19. Comunicación entre H1 y la interfaz del switch.

```

22:12:55.149848 ARP, Request who-has 10.0.1.1 tell 10.0.1.100, length 28
22:12:55.152291 ARP, Reply 10.0.1.1 is-at 00:00:00:00:11 (oui Ethernet), length 28
22:12:55.152316 IP 10.0.1.100 > 10.0.1.1: ICMP echo request, id 44371, seq 1, length 64
22:12:55.154397 IP 10.0.1.1 > 10.0.1.100: ICMP echo reply, id 44371, seq 1, length 64
22:12:55.151801 IP 10.0.1.100 > 10.0.1.1: ICMP echo request, id 44371, seq 2, length 64
22:12:55.153937 IP 10.0.1.1 > 10.0.1.100: ICMP echo reply, id 44371, seq 2, length 64
22:12:57.153316 IP 10.0.1.100 > 10.0.1.1: ICMP echo request, id 44371, seq 3, length 64
22:12:57.155417 IP 10.0.1.1 > 10.0.1.100: ICMP echo reply, id 44371, seq 3, length 64

```

Fig. 20. Paquetes intercambiados en la comunicación entre H1 y la interfaz del switch.

Por último, tenemos que comprobar el funcionamiento de las reglas OVS que se aplican sobre el switch para que no tenga que interactuar siempre con el controlador.

Para realizar la prueba, se utiliza un comando ping para enviar paquetes entre las subredes y se comprueba el tiempo que tarda en realizarse dicha comunicación, así cómo también se comprueba si pasan los paquetes por las reglas que se han instaurado.

En la figura 21 se observa cómo al realizar un ping de un host a otro, el primer paquete que se envía tarda un total de 50.3 ms, mientras que el resto de paquetes tardan mucho menos, esto se debe a que tras el envío del primer paquete se crea la regla que permite al switch conocer donde tiene que redirigir los paquetes que le llegan y no necesita acceder al controlador para saber a donde tiene que reenviarlo.

```

mininet> h1 ping h2
PING 10.0.2.100 (10.0.2.100) 56(84) bytes of data.
64 bytes from 10.0.2.100: icmp_seq=1 ttl=64 time=50.3 ms
64 bytes from 10.0.2.100: icmp_seq=2 ttl=64 time=2.86 ms
64 bytes from 10.0.2.100: icmp_seq=3 ttl=64 time=0.382 ms
64 bytes from 10.0.2.100: icmp_seq=4 ttl=64 time=0.119 ms
64 bytes from 10.0.2.100: icmp_seq=5 ttl=64 time=0.121 ms
^C
--- 10.0.2.100 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4032ms
rtt min/avg/max/mdev = 0.119/10.752/50.283/19.792 ms

```

Fig. 21. Comunicación entre H1 y H2.

Tras la creación de la regla, las comunicaciones en ambas direcciones entre los hosts únicamente son controladas por el switch sin la necesidad de la intervención del controlador.

Cómo se observa en la figura 22 el tiempo que tardan en llegar los paquetes entre el host h2 y h1 es mucho menor, esto es debido a que la regla ya está implementada y los paquetes pasan por ella cómo se muestra en la figura 23.

```
mininet> h2 ping h1
PING 10.0.1.100 (10.0.1.100) 56(84) bytes of data.
64 bytes from 10.0.1.100: icmp_seq=1 ttl=64 time=0.508 ms
64 bytes from 10.0.1.100: icmp_seq=2 ttl=64 time=0.113 ms
64 bytes from 10.0.1.100: icmp_seq=3 ttl=64 time=0.107 ms
64 bytes from 10.0.1.100: icmp_seq=4 ttl=64 time=0.143 ms
64 bytes from 10.0.1.100: icmp_seq=5 ttl=64 time=0.122 ms
^C
--- 10.0.1.100 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4090ms
rtt min/avg/max/mdev = 0.107/0.198/0.508/0.155 ms
```

Fig. 22. Comunicación entre H2 y H1.

```
mininet> sh ovs-ofctl dump-flows s1
cookie=0x0, duration=485.509s, table=0, n_packets=0, n_bytes=0, arp,arp_tpa=10.0.2.100 actions=output:"s1-eth2"
cookie=0x0, duration=485.418s, table=0, n_packets=9, n_bytes=882, ip,d_l_src=00:00:00:00:00:02,d_l_dst=00:00:00:00:00:22,nw_dst=10.0.1.100 actions=mod_d_l_src:00:00:00:00:00:22,mod_d_l_dst:00:00:00:00:00:01,output:"s1-eth1"
cookie=0x0, duration=484.508s, table=0, n_packets=8, n_bytes=784, ip,d_l_src=00:00:00:00:00:01,d_l_dst=00:00:00:00:00:11,nw_dst=10.0.2.100 actions=mod_d_l_src:00:00:00:00:00:11,mod_d_l_dst:00:00:00:00:00:02,output:"s1-eth2"
```

Fig. 23. Reglas OVS del switch S1.