
# Networks II Final Project

Anagnostopoulos Vasileios 03153, Georgitziki Garyfalia 03218

June 24, 2024

## Contents

# 1 ARP spoofing

## 1.1 Topology

The topology of this part is the below.

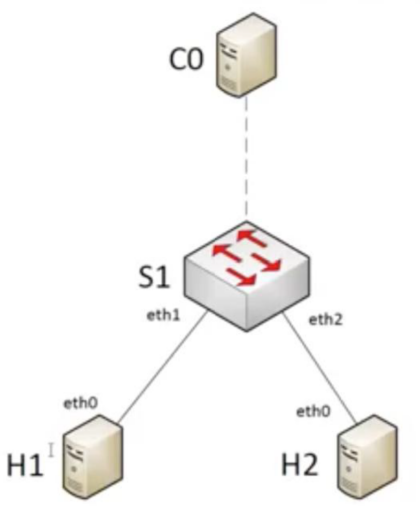


Figure 1: Part 1 Topology

The topology above is created with the below command.

```
mininet@mininet-vm:~$ sudo mn --controller remote --mac
```

Figure 2: Command to create the topology

## 1.2 Explanation of the code

The code is written to the script `arp-spoofing.py`. The packet-in handler returns, when the triggering packet has Ethertype equal to 0x86dd and code is below. This check is also done in the following Static routing, Static routing with two routers and VLAN with OpenFlow.

```
#Check if the packet is IPv6 (Ethertype 0x86dd)
if ethertype == ether_types.ETH_TYPE_IPV6:
    self.logger.info("IPv6 packet in %s %s %s %s in_port=%s\n", hex(dpid), hex(ethertype), src, dst, msg.in_port)
    return  #Exit the handler
```

Figure 3: For IPv6 packets

First check if the packet is ARP, then if the source MAC is 00:00:00:00:00:00:01 or 00:00:00:00:00:00:02 i.e. h1 sends to h2 or h2 sends to h1 respectively, then we check if

it is ARP REQUEST and if it is we call our own function shown below to send the ARP REPLY through the switch instead of h1 or h2 sending it. The first if is for `h1 ->(send to) h2`, the second one(elif) is for `h2 ->(send to) h1`.

```
if eth.ethertype == ether_types.ETH_TYPE_ARP: # this packet is ARP packet
    #Check if h1 sends traffic to h2 to handle the ARP REPLY from the switch
    if packet_arp.src_mac == '00:00:00:00:00:01':
        """
        fill in the code here for the ARP requests operation, creating and sending ARP replies.
        """
        if packet_arp.opcode == arp.ARP_REQUEST:
            #Call the arp_reply function for 1.ARP spoofing
            self.arp_reply(
                action_list=[datapath.ofproto_parser.OFPActionOutput(msg.in_port)],
                dp=datapath,
                source_mac='00:00:00:00:00:02',  #h2 MAC address
                dest_mac=packet_arp.src_mac,
                source_ip=packet_arp.dst_ip,
                dest_ip=packet_arp.src_ip
            )
        return
    elif packet_arp.src_mac == '00:00:00:00:00:02':
        if packet_arp.opcode == arp.ARP_REQUEST:
            #Call the arp_reply function for 1.ARP spoofing
            self.arp_reply(
                action_list=[datapath.ofproto_parser.OFPActionOutput(msg.in_port)],
                dp=datapath,
                source_mac='00:00:00:00:00:01',  #h1 MAC address
                dest_mac=packet_arp.src_mac,
                source_ip=packet_arp.dst_ip,
                dest_ip=packet_arp.src_ip
            )
        return
```

Figure 4: 2 cases for ARP Requests

```
#Function that is called for ARP packets to fill the ARP table
def arp_reply(self, action_list, dp, source_mac, dest_mac, source_ip, dest_ip):
    #Get the OpenFlow protocol object for the datapath (switch)
    ofproto = dp.ofproto

    #Create a new packet for the ARP reply
    arp_reply_packet = packet.Packet()

    #Add an Ethernet frame to the packet with the appropriate ethertype, destination MAC, and source MAC
    arp_reply_packet.add_protocol(ethernet.ethernet(ethertype=ether.ETH_TYPE_ARP, dst=dest_mac, src=source_mac))

    #Add an ARP protocol frame to the packet with the ARP reply opcode and the source and destination MAC and IP addresses
    arp_reply_packet.add_protocol(arp.arp(opcode=arp.ARP_REPLY, src_mac=source_mac, src_ip=source_ip, dst_mac=dest_mac, dst_ip=dest_ip))

    #Serialize the packet to prepare it for sending
    arp_reply_packet.serialize()

    #Create an OpenFlow PacketOut message to send the ARP reply packet
    out = dp.ofproto_parser.OFPPacketOut(
        datapath=dp, buffer_id=ofproto.OFP_NO_BUFFER, in_port=ofproto.OFPP_CONTROLLER,
        actions=action_list, data=arp_reply_packet.data
    )
    #Log the ARP reply being sent with the source and destination MAC and IP addresses
    self.logger.info("The ARP Reply is: SourceMAC: %s SourceIP: %s to DestinationMAC: %s DestinationIP: %s]\n", source_mac, source_ip, dest_mac, dest_ip)
    #Send the PacketOut message to the datapath (switch)
    dp.send_msg(out)
```

Figure 5: `arp_reply` function

The function `arp_reply` takes several parameters including action_list, dp (datapath), source_mac, dest_mac, source_ip, and dest_ip. It first obtains the OpenFlow protocol object (ofproto) for the given datapath. Then, it creates a new packet instance for the ARP reply and adds an Ethernet frame to this packet with the specified EtherType, destination MAC, and source MAC addresses. It also includes an ARP protocol frame with the ARP reply opcode and the provided MAC and IP addresses for both source and destination. The packet is serialized to prepare it for sending. An OpenFlow PacketOut message is created to encapsulate the ARP reply packet and includes the specified action list, buffer ID, and the controller port. This message is then sent to the switch.

## 1.3 Examples

The first example is `h1 ping h2` that has also an ICMP Reply from h2 to h1 so the arp tables of the two hosts are created and the records are below.

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=12.5 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=2.38 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.276 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.099 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3018ms
rtt min/avg/max/mdev = 0.099/3.825/12.549/5.115 ms
mininet> h1 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.0.2                 ether   00:00:00:00:00:02   C                     h1-eth0
mininet> h2 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.0.1                 ether   00:00:00:00:00:01   C                     h2-eth0
```

Figure 6: `h1 ping h2` and ARP tables

```
packet in 0x1 0x806 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff in_port=1
The ARP Reply is: SourceMAC: 00:00:00:00:00:02 SourceIP: 10.0.0.2 to DestinationMAC
: 00:00:00:00:00:01 DestinationIP: 10.0.0.1]

packet in 0x1 0x800 00:00:00:00:00:01 00:00:00:00:00:02 in_port=1
packet in 0x1 0x806 00:00:00:00:00:02 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:00:01 SourceIP: 10.0.0.1 to DestinationMAC
: 00:00:00:00:00:02 DestinationIP: 10.0.0.2]

packet in 0x1 0x800 00:00:00:00:00:02 00:00:00:00:00:01 in_port=2
packet in 0x1 0x800 00:00:00:00:00:01 00:00:00:00:00:02 in_port=1
```

Figure 7: Controller output

The second example(exit from the mininet and create it from the beginning) is `h2 ping h1` that has also an ICMP Reply from h1 to h2 so the arp tables of the two hosts are created and the records are below.

```
mininet> h2 ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=15.1 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=4.35 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.250 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.079 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.118 ms
^C
--- 10.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4030ms
rtt min/avg/max/mdev = 0.079/3.979/15.100/5.793 ms
mininet> h2 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.0.1                 ether   00:00:00:00:00:01   C                     h2-eth0
mininet> h1 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.0.2                 ether   00:00:00:00:00:02   C                     h1-eth0
```

Figure 8: `h2 ping h1` and ARP tables

```
packet in 0x1 0x806 00:00:00:00:00:02 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:00:01 SourceIP: 10.0.0.1 to DestinationMAC
: 00:00:00:00:00:02 DestinationIP: 10.0.0.2]

packet in 0x1 0x800 00:00:00:00:00:02 00:00:00:00:00:01 in_port=2
packet in 0x1 0x806 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff in_port=1
The ARP Reply is: SourceMAC: 00:00:00:00:00:02 SourceIP: 10.0.0.2 to DestinationMAC
: 00:00:00:00:00:01 DestinationIP: 10.0.0.1]

packet in 0x1 0x800 00:00:00:00:00:01 00:00:00:00:00:02 in_port=1
packet in 0x1 0x800 00:00:00:00:00:02 00:00:00:00:00:01 in_port=2
```

Figure 9: Controller output

# 2 Static routing

## 2.1 Topology

The topology of this part is the below.

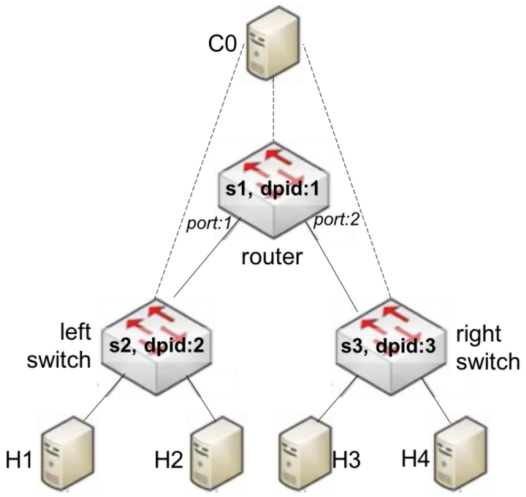


Figure 10: Part 2 Topology

The topology above is created with the script `mininet-router.py`.

```
root@mininet-vm:~# chmod 777 mininet-router.py
root@mininet-vm:~# ./mininet-router.py
```

Figure 11: Commands to create the topology

## 2.2 Explanation of the code

The code is written to the script `ryu-router-frame.py`. The variable that is used in the code below is the ARP table(the mappings of the ip addresses to mac addresses).

```
ARP_table = {
    '192.168.1.2': '00:00:00:00:01:02',
    '192.168.1.3': '00:00:00:00:01:03',
    '192.168.2.2': '00:00:00:00:02:02',
    '192.168.2.3': '00:00:00:00:02:03'
}
```

Figure 12: ARP table variable

- `packet_arp = pkt.get_protocol(arp.arp)`: Extracts the ARP protocol from the packet.
- `packet_ip = pkt.get_protocol(ipv4.ipv4)`: Extracts the IPv4 protocol from the packet.

The code identifies the router using `dpid == 1`. If the packet's ethertype indicates an ARP packet (`eth.ethertype == ether_types.ETH_TYPE_ARP`):

- The code checks if the ARP packet is an ARP request (`packet_arp.opcode == arp.ARP_REQUEST`).
  - If the destination IP of the ARP request (`packet_arp.dst_ip`) is 192.168.1.1 (Left LAN), it sends an ARP reply with:
    - Source MAC address: 00:00:00:00:01:01
    - Destination MAC address: `src` (MAC address of the requester)
    - Source IP address: `packet_arp.dst_ip` (192.168.1.1)
    - Destination IP address: `packet_arp.src_ip` (IP address of the requester)
  - If the destination IP of the ARP request (`packet_arp.dst_ip`) is 192.168.2.1 (Right LAN), it sends an ARP reply with:
    - Source MAC address: 00:00:00:00:02:01
    - Destination MAC address: `src` (MAC address of the requester)
    - Source IP address: `packet_arp.dst_ip` (192.168.2.1)
    - Destination IP address: `packet_arp.src_ip` (IP address of the requester)

If the packet's ethertype indicates an IP packet (`eth.ethertype == ether_types.ETH_TYPE_IP`):

- The code extracts the destination IP address (`packet_ip.dst`).
  - If the destination IP is within the 192.168.1. subnet, forwards the packet within the left LAN using the source MAC address 00:00:00:00:01:01 and the destination MAC address from the `ARP_table`.
  - If the destination IP is within the 192.168.2. subnet, forwards the packet within the right LAN using the source MAC address 00:00:00:00:02:01 and the destination MAC address from the `ARP_table`.

```
packet_arp = pkt.get_protocol(arp.arp)
packet_ip = pkt.get_protocol(ipv4.ipv4)

if dpid == 1:
    if eth.ethertype == ether_types.ETH_TYPE_ARP: # this packet is ARP packet
        """
        fill in the code here for the ARP requests operation, creating and sending ARP replies.
        """
        if packet_arp.opcode == arp.ARP_REQUEST:
            #2.Static routing
            if packet_arp.dst_ip == '192.168.1.1': #Left LAN
                self.arp_reply(
                    action_list = [datapath.ofproto_parser.OFPActionOutput(msg.in_port)],
                    dp=datapath,
                    source_mac="00:00:00:00:01:01",
                    dest_mac=src,
                    source_ip=packet_arp.dst_ip,
                    dest_ip=packet_arp.src_ip
                )
            elif packet_arp.dst_ip == '192.168.2.1': #Right LAN
                self.arp_reply(
                    action_list = [datapath.ofproto_parser.OFPActionOutput(msg.in_port)],
                    dp=datapath,
                    source_mac="00:00:00:00:02:01",
                    dest_mac=src,
                    source_ip=packet_arp.dst_ip,
                    dest_ip=packet_arp.src_ip
                )
            else:
                return

        return
```

Figure 13: ARP Replies for two LANs

```
elif eth.ethertype == ether_types.ETH_TYPE_IP: # this packet is IP packet
    """
    fill in the code here for the IP packets operation
    You must i) handle the packets coming to the controller with a packet_out message and then
    ii) add an appropriate flow, modifying and using the add_flow function, in order the controller to
    """
    #2.Static routing
    dest_ip = packet_ip.dst
    if '192.168.1.' in dest_ip:
        self.logger.info("Packet in left LAN: %s ----> %s\n", packet_ip.src, dest_ip)
        self.ip_match_and_forward(datapath, "00:00:00:00:01:01", ARP_table[dest_ip], dest_ip, 1, pkt)

    elif '192.168.2.' in dest_ip:
        self.logger.info("Packet in right LAN: %s ----> %s\n", packet_ip.src, dest_ip)
        self.ip_match_and_forward(datapath, "00:00:00:00:02:01", ARP_table[dest_ip], dest_ip, 2, pkt)

    return
return
```

Figure 14: The forwarding in the correct LAN

```python
#Function that is called for IP packets to fill the flow table
def ip_match_and_forward(self, dp, source_mac, dest_mac, dest_ip, out_port, pkt):
    #Get the OpenFlow protocol object for the datapath (switch)
    ofproto = dp.ofproto

    #Define the actions to be taken on the packet
    actions = [
        dp.ofproto_parser.OFPActionSetDlSrc(source_mac),
        dp.ofproto_parser.OFPActionSetDlDst(dest_mac),
        dp.ofproto_parser.OFPActionOutput(out_port)
    ]

    #Mask is 32 because the packets go to a specific host and not in other router
    ip_mask = 32

    #Create a PacketOut message to send the packet immediately
    out = dp.ofproto_parser.OFPPacketOut(
        datapath=dp,
        buffer_id=ofproto.OFP_NO_BUFFER,
        in_port=ofproto.OFPP_CONTROLLER,
        actions=actions,
        data=pkt.data
        )
    #Send the PacketOut message to the datapath (switch)
    dp.send_msg(out)

    #Create a match object to match IP packets with the specified destination IP and mask
    match = dp.ofproto_parser.OFPMatch(
        dl_type=ether_types.ETH_TYPE_IP,
        nw_dst=dest_ip,
        nw_dst_mask=ip_mask
        )
    #Add the flow entry to the switch with the specified match and actions
    self.add_flow(dp, match, actions)
```

Figure 15: `ip_match_and_forward` function

The `ip_match_and_forward` function is responsible for handling IP packets by updating the flow table of an OpenFlow-compatible switch and forwarding the packet immediately. It first retrieves the OpenFlow protocol object from the datapath (switch). Then, it defines a list of actions to be performed on the packet, which includes setting the source MAC address, setting the destination MAC address, and specifying the output port. A packet out message is created using these actions, which is sent to the datapath to forward the packet immediately. The function also creates a match object to match IP packets with a specified destination IP and a 32-bit mask (indicating that the packets go to a specific host and not in other router). Finally, it adds a flow entry to the switch with the specified match and actions by calling the `add_flow` method, ensuring that future packets with the same destination IP are handled according to the defined actions.

## 2.3 Examples

The first example is `h1 ping h3` that has also an ICMP Reply from h3 to h1 as said above so the arp tables of the two hosts are created and the records are below.

```
mininet> h1 ping h3
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=37.3 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=64 time=0.470 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=64 time=0.087 ms
64 bytes from 192.168.2.2: icmp_seq=4 ttl=64 time=0.069 ms
^C
--- 192.168.2.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3050ms
rtt min/avg/max/mdev = 0.069/9.476/37.281/16.053 ms
mininet> h1 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:00:00:00:01:01   C                     h1-eth1
mininet> h3 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.1              ether   00:00:00:00:02:01   C                     h3-eth1
```

Figure 16: `h1 ping h3` and ARP tables

```
packet in 0x2 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x1 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=1
The ARP Reply is: SourceMAC: 00:00:00:00:01:01 SourceIP: 192.168.1.1 to DestinationMAC: 00:00:00:00:01:02 DestinationIP: 192.168.1.2]

packet in 0x2 0x806 00:00:00:00:01:01 00:00:00:00:01:02 in_port=1
packet in 0x2 0x800 00:00:00:00:01:02 00:00:00:00:01:01 in_port=2
packet in 0x1 0x800 00:00:00:00:01:02 00:00:00:00:01:01 in_port=1
Packet in right LAN: 192.168.1.2 ----> 192.168.2.2

packet in 0x3 0x800 00:00:00:00:02:01 00:00:00:00:02:02 in_port=1
packet in 0x3 0x806 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x1 0x806 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:02:01 SourceIP: 192.168.2.1 to DestinationMAC: 00:00:00:00:02:02 DestinationIP: 192.168.2.2]

packet in 0x3 0x806 00:00:00:00:02:01 00:00:00:00:02:02 in_port=1
packet in 0x3 0x800 00:00:00:00:02:02 00:00:00:00:02:01 in_port=2
packet in 0x1 0x800 00:00:00:00:02:02 00:00:00:00:02:01 in_port=2
Packet in left LAN: 192.168.2.2 ----> 192.168.1.2
```

Figure 17: Controller output

The second example, is `h2 ping h4` that has also an ICMP Reply from h4 to h2 so the arp tables of the two hosts are created and the records are below.

```
mininet> h2 ping h4
PING 192.168.2.3 (192.168.2.3) 56(84) bytes of data.
64 bytes from 192.168.2.3: icmp_seq=1 ttl=64 time=30.1 ms
64 bytes from 192.168.2.3: icmp_seq=2 ttl=64 time=0.321 ms
64 bytes from 192.168.2.3: icmp_seq=3 ttl=64 time=0.065 ms
64 bytes from 192.168.2.3: icmp_seq=4 ttl=64 time=0.110 ms
64 bytes from 192.168.2.3: icmp_seq=5 ttl=64 time=0.101 ms
^C
--- 192.168.2.3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4077ms
rtt min/avg/max/mdev = 0.065/6.132/30.066/11.967 ms
mininet> h2 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:00:00:00:01:01   C                     h2-eth1
mininet> h4 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.1              ether   00:00:00:00:02:01   C                     h4-eth1
```

Figure 18: `h2 ping h4` and ARP tables

```
packet in 0x2 0x806 00:00:00:00:01:03 ff:ff:ff:ff:ff:ff in_port=3
packet in 0x1 0x806 00:00:00:00:01:03 ff:ff:ff:ff:ff:ff in_port=1
The ARP Reply is: SourceMAC: 00:00:00:00:01:01 SourceIP: 192.168.1.1 to DestinationMAC: 00:00:00:00:01:03 DestinationIP: 192.168.1.3]

packet in 0x2 0x806 00:00:00:00:01:01 00:00:00:00:01:03 in_port=1
packet in 0x2 0x800 00:00:00:00:01:03 00:00:00:00:01:01 in_port=3
packet in 0x1 0x800 00:00:00:00:01:03 00:00:00:00:01:01 in_port=1
Packet in right LAN: 192.168.1.3 ----> 192.168.2.3

packet in 0x3 0x800 00:00:00:00:02:01 00:00:00:00:02:03 in_port=1
packet in 0x3 0x806 00:00:00:00:02:03 ff:ff:ff:ff:ff:ff in_port=3
packet in 0x1 0x806 00:00:00:00:02:03 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:02:01 SourceIP: 192.168.2.1 to DestinationMAC: 00:00:00:00:02:03 DestinationIP: 192.168.2.3]

packet in 0x3 0x806 00:00:00:00:02:01 00:00:00:00:02:03 in_port=1
packet in 0x3 0x800 00:00:00:00:02:03 00:00:00:00:02:01 in_port=3
packet in 0x1 0x800 00:00:00:00:02:03 00:00:00:00:02:01 in_port=2
Packet in left LAN: 192.168.2.3 ----> 192.168.1.3
```

Figure 19: Controller output

The third example, is `h1 ping h4` and `h2 ping h3` with the ICMP Replies also so the arp tables of all hosts remain the same like the previous 2 examples and the records are below.

```
mininet> h1 ping h4
PING 192.168.2.3 (192.168.2.3) 56(84) bytes of data.
64 bytes from 192.168.2.3: icmp_seq=1 ttl=64 time=0.242 ms
64 bytes from 192.168.2.3: icmp_seq=2 ttl=64 time=0.093 ms
^C
--- 192.168.2.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1015ms
rtt min/avg/max/mdev = 0.093/0.167/0.242/0.074 ms
mininet> h2 ping h3
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=0.342 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=64 time=0.090 ms
^C
--- 192.168.2.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1027ms
rtt min/avg/max/mdev = 0.090/0.216/0.342/0.126 ms
mininet> h1 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:00:00:00:01:01   C                     h1-eth1
mininet> h2 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:00:00:00:01:01   C                     h2-eth1
mininet> h3 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.1              ether   00:00:00:00:02:01   C                     h3-eth1
mininet> h4 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.1              ether   00:00:00:00:02:01   C                     h4-eth1
```

Figure 20: `h1 ping h4`, `h2 ping h3` and ARP tables of all hosts

The flow table of router s1 has 4 records(with source MAC, the two MACs of two ports of the router and destination MAC, the MACs of the hosts) after all runs with pings and is the below.

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1
 cookie=0x0, duration=103.637s, table=0, n_packets=3, n_bytes=294, ip,nw_dst=192.168.2.2
 actions=mod_dl_src:00:00:00:00:02:01,mod_dl_dst:00:00:00:00:02:02,output:"s1-eth2"
 cookie=0x0, duration=103.605s, table=0, n_packets=3, n_bytes=294, ip,nw_dst=192.168.1.2
 actions=mod_dl_src:00:00:00:00:01:01,mod_dl_dst:00:00:00:00:01:02,output:"s1-eth1"
 cookie=0x0, duration=98.408s, table=0, n_packets=4, n_bytes=392, ip,nw_dst=192.168.2.3
actions=mod_dl_src:00:00:00:00:02:01,mod_dl_dst:00:00:00:00:02:03,output:"s1-eth2"
 cookie=0x0, duration=98.383s, table=0, n_packets=4, n_bytes=392, ip,nw_dst=192.168.1.3
actions=mod_dl_src:00:00:00:00:01:01,mod_dl_dst:00:00:00:00:01:03,output:"s1-eth1"
```

Figure 21: Flow table of s1

# 3 Static routing with two routers

## 3.1 Topology

The topology of this part is the below.

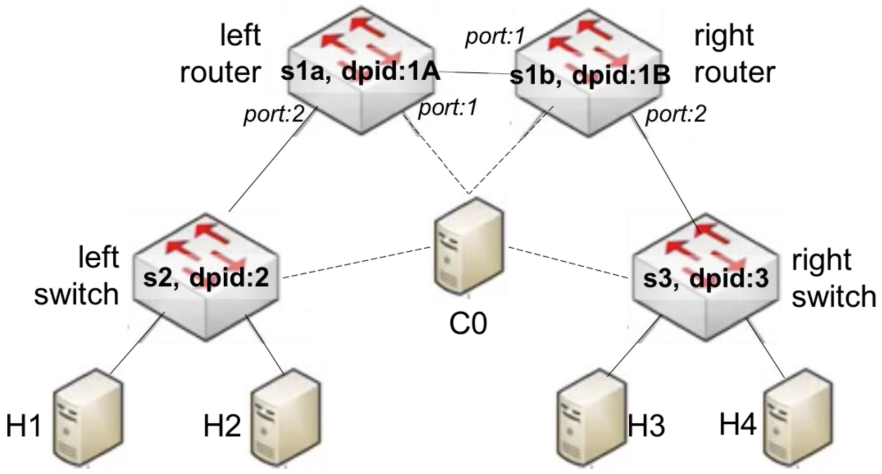


Figure 22: Part 3 Topology

The topology above is created with the script `mininet-router-two.py`.

```
root@mininet-vm:~# chmod 777 mininet-router-two.py
root@mininet-vm:~# ./mininet-router-two.py
```

Figure 23: Commands to create the topology

## 3.2 Explanation of the code

The code is written to the script `ryu-router-two-frame.py`. The variables that are used in the code below are the ARP tables of the left and right LAN(the mappings of the ip addresses to mac addresses).

```
left_LAN_ARP_table = {
    '192.168.1.2': '00:00:00:00:01:02',
    '192.168.1.3': '00:00:00:00:01:03'
}

right_LAN_ARP_table = {
    '192.168.2.2': '00:00:00:00:02:02',
    '192.168.2.3': '00:00:00:00:02:03'
}
```

Figure 24: 2 ARP tables for each LAN

The function `ip_match_and_forward` has changed a little bit, the mask has been added as a parameter as the packet can go to another router so the mask must be 24 (24 bits must be checked) instead of 32 when the whole IP address must be checked to go to a specific host.

```
#Function that is called for IP packets to fill the flow table
def ip_match_and_forward(self, dp, source_mac, dest_mac, dest_ip, mask, out_port, pkt):
    #Get the OpenFlow protocol object for the datapath (switch)
    ofproto = dp.ofproto

    #Define the actions to be taken on the packet
    action_list = [
        dp.ofproto_parser.OFPActionSetDlSrc(source_mac),
        dp.ofproto_parser.OFPActionSetDlDst(dest_mac),
        dp.ofproto_parser.OFPActionOutput(out_port)
    ]

    #Define the IP mask based on the provided subnet mask. Mask is either 24 or 32
    ip_mask = mask

    #Create a PacketOut message to send the packet immediately
    out = dp.ofproto_parser.OFPPacketOut(
        datapath=dp,
        buffer_id=ofproto.OFP_NO_BUFFER,
        in_port=ofproto.OFPP_CONTROLLER,
        actions=action_list,
        data=pkt.data
        )
    #Send the PacketOut message to the datapath (switch)
    dp.send_msg(out)

    #Create a match object to match IP packets with the specified destination IP and mask
    match = dp.ofproto_parser.OFPMatch(
        dl_type=ether_types.ETH_TYPE_IP,
        nw_dst=dest_ip,
        nw_dst_mask=ip_mask
        )
    #Add the flow entry to the switch with the specified match and actions
    self.add_flow(dp, match, action_list)
```

Figure 25: Changed `ip_match_and_forward` function

The routers, identified by dpid 0x1A and 0x1B, are responsible for managing ARP and IP packets, facilitating communication between different LAN segments. The code distinguishes between two routers using the dpid (Datapath ID) values 0x1A and 0x1B. If the packet's ethertype indicates an ARP packet (`ETH_TYPE_ARP`), the code checks if it is an ARP request (`packet_arp.opcode == arp.ARP_REQUEST`). If the destination IP of the ARP request (`packet_arp.dst_ip`) is 192.168.1.1, it sends an ARP reply with the source MAC address 00:00:00:00:01:01, which is the MAC address of router 0x1A on the left LAN. Similarly, if the destination IP is 192.168.2.1, it sends an ARP reply with the source MAC address 00:00:00:00:02:01, which is the MAC address of router 0x1B on the right LAN. If the packet's ethertype indicates an IP packet (`ETH_TYPE_IP`), the code extracts the destination IP address (`packet_ip.dst`). If the destination IP is within the 192.168.1. subnet and is present in the `left_LAN_ARP_table`, it forwards it within the left LAN using the corresponding MAC address. If the destination IP is within the 192.168.2. subnet, it forwards the packet to router 0x1B via the interface with MAC address 00:00:00:00:03:01. If the destination IP is within the 192.168.1. subnet, it forwards the packet to router 0x1A via the interface with MAC address 00:00:00:00:03:02. If the destination IP is within the 192.168.2. subnet and is present in the `right_LAN_ARP_table`, it logs the packet information and forwards it within the right LAN using the corresponding MAC address.

- `self.arp_reply`: Sends an ARP reply with specified source and destination MAC and IP addresses.
- `self.ip_match_and_forward`: Forwards IP packets based on destination IP, source and destination MAC addresses, subnet mask, output port, and the packet itself.
- `datapath.ofproto_parser.OFPActionOutput`: Specifies the output action to send the packet out of a specified port.
- `left_LAN_ARP_table` and `right_LAN_ARP_table`: Static ARP tables mapping IP addresses to MAC addresses for the left and right LANs, respectively.

```
if dpid == 0x1A:
    if ethertype == ether_types.ETH_TYPE_ARP: # this packet is ARP packet
        """
        fill in the code here
        """
        #3.Static routing with two routers
        if packet_arp.opcode == arp.ARP_REQUEST:
            if packet_arp.dst_ip == '192.168.1.1': #Left LAN
                    self.arp_reply(
                        action_list = [datapath.ofproto_parser.OFPActionOutput(msg.in_port)],
                        dp=datapath,
                        source_mac="00:00:00:00:01:01",
                        dest_mac=src,
                        source_ip=packet_arp.dst_ip,
                        dest_ip=packet_arp.src_ip
                    )
            else:
                return

        return
    elif ethertype == ether_types.ETH_TYPE_IP: # this packet is IP packet
        """
        fill in the code here
        """
        #3.Static routing with two routers
        dest_ip = packet_ip.dst
        if '192.168.1.' in dest_ip and dest_ip in left_LAN_ARP_table:
            self.logger.info("Packet in left LAN: %s ----> %s\n", packet_ip.src, dest_ip)
            self.ip_match_and_forward(datapath, "00:00:00:00:01:01", left_LAN_ARP_table[dest_ip], dest_ip, 32, 2, pkt)

        elif '192.168.2.' in dest_ip:
            self.logger.info("Packet in router 1B: %s ----> %s\n", packet_ip.src, dest_ip)
            self.ip_match_and_forward(datapath, "00:00:00:00:03:01", "00:00:00:00:03:02", dest_ip, 24, 1, pkt)

        return
    return
```

Figure 26: Router s1a functionality for ARP Replies and Forwarding

```
if dpid == 0x1B:
    if ethertype == ether_types.ETH_TYPE_ARP: # this packet is ARP packet
        """
        fill in the code here
        """
        #3.Static routing with two routers
        if packet_arp.opcode == arp.ARP_REQUEST:
            if packet_arp.dst_ip == '192.168.2.1': #Left LAN
                    self.arp_reply(
                        action_list = [datapath.ofproto_parser.OFPActionOutput(msg.in_port)],
                        dp=datapath,
                        source_mac="00:00:00:00:02:01",
                        dest_mac=src,
                        source_ip=packet_arp.dst_ip,
                        dest_ip=packet_arp.src_ip
                    )
            else:
                return
        return
    elif ethertype == ether_types.ETH_TYPE_IP: # this packet is IP packet
        """
        fill in the code here
        """
        #3.Static routing with two routers
        dest_ip = packet_ip.dst
        if '192.168.1.' in dest_ip:
            self.logger.info("Packet in router 1A: %s ----> %s\n", packet_ip.src, dest_ip)
            self.ip_match_and_forward(datapath, "00:00:00:00:03:02", "00:00:00:00:03:01", dest_ip, 24, 1, pkt)

        elif '192.168.2.' in dest_ip and dest_ip in right_LAN_ARP_table:
            self.logger.info("Packet in right LAN: %s ----> %s\n", packet_ip.src, dest_ip)
            self.ip_match_and_forward(datapath, "00:00:00:00:02:01", right_LAN_ARP_table[dest_ip], dest_ip, 32, 2, pkt)

        return
    return
```

Figure 27: Router s1b functionality for ARP Replies and Forwarding

## 3.3 Examples

The first example is `h1 ping h3` that has also an ICMP Reply from h3 to h1 as said above so the arp tables of the two hosts are created and the records are below.

```
mininet> h1 ping h3
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=33.4 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=64 time=0.600 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=64 time=0.088 ms
^C
--- 192.168.2.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2032ms
rtt min/avg/max/mdev = 0.088/11.348/33.358/15.564 ms
mininet> h1 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:00:00:00:01:01   C                     h1-eth1
mininet> h3 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.1              ether   00:00:00:00:02:01   C                     h3-eth1
```

Figure 28: `h1 ping h3` and ARP tables

```
packet in 0x2  0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x1a 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:01:01 SourceIP: 192.168.1.1 to DestinationMAC: 00:00:00:00:01:02 DestinationIP: 192.168.1.2]

packet in 0x2  0x806 00:00:00:00:01:01 00:00:00:00:01:02 in_port=1
packet in 0x2  0x800 00:00:00:00:01:02 00:00:00:00:01:01 in_port=2
packet in 0x1a 0x800 00:00:00:00:01:02 00:00:00:00:01:01 in_port=2
Packet in router 1B: 192.168.1.2 ----> 192.168.2.2

packet in 0x1b 0x800 00:00:00:00:03:01 00:00:00:00:03:02 in_port=1
Packet in right LAN: 192.168.1.2 ----> 192.168.2.2

packet in 0x3  0x800 00:00:00:00:02:01 00:00:00:00:02:02 in_port=1
packet in 0x3  0x806 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x1b 0x806 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:02:01 SourceIP: 192.168.2.1 to DestinationMAC: 00:00:00:00:02:02 DestinationIP: 192.168.2.2]

packet in 0x3  0x806 00:00:00:00:02:01 00:00:00:00:02:02 in_port=1
packet in 0x3  0x800 00:00:00:00:02:02 00:00:00:00:02:01 in_port=2
packet in 0x1b 0x800 00:00:00:00:02:02 00:00:00:00:02:01 in_port=2
Packet in router 1A: 192.168.2.2 ----> 192.168.1.2

packet in 0x1a 0x800 00:00:00:00:03:02 00:00:00:00:03:01 in_port=1
Packet in left LAN: 192.168.2.2 ----> 192.168.1.2
```

Figure 29: Controller output

The second example, is `h1 ping h4` that has also an ICMP Reply from h4 to h1 so the arp tables of the two hosts are created and the records are below.

```
mininet> h1 ping h4
PING 192.168.2.3 (192.168.2.3) 56(84) bytes of data.
64 bytes from 192.168.2.3: icmp_seq=1 ttl=64 time=24.9 ms
64 bytes from 192.168.2.3: icmp_seq=2 ttl=64 time=0.336 ms
64 bytes from 192.168.2.3: icmp_seq=3 ttl=64 time=0.126 ms
^C
--- 192.168.2.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2011ms
rtt min/avg/max/mdev = 0.126/8.438/24.854/11.607 ms
mininet> h1 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:00:00:00:01:01   C                     h1-eth1
mininet> h4 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.1              ether   00:00:00:00:02:01   C                     h4-eth1
```

Figure 30: `h1 ping h4` and ARP tables

```
packet in 0x1b 0x800 00:00:00:00:03:01 00:00:00:00:03:02 in_port=1
Packet in right LAN: 192.168.1.2 ----> 192.168.2.3

packet in 0x3  0x800 00:00:00:00:02:01 00:00:00:00:02:03 in_port=1
packet in 0x3  0x806 00:00:00:00:02:03 ff:ff:ff:ff:ff:ff in_port=3
packet in 0x1b 0x806 00:00:00:00:02:03 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:02:01 SourceIP: 192.168.2.1 to DestinationMAC: 00:00:00:00:02:03 DestinationIP: 192.168.2.3]

packet in 0x3  0x806 00:00:00:00:02:01 00:00:00:00:02:03 in_port=1
packet in 0x3  0x800 00:00:00:00:02:03 00:00:00:00:02:01 in_port=3
packet in 0x1a 0x806 00:00:00:00:01:02 00:00:00:00:01:01 in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:01:01 SourceIP: 192.168.1.1 to DestinationMAC: 00:00:00:00:01:02 DestinationIP: 192.168.1.2]
```

Figure 31: Controller output

The third example, is `h2 ping h3` and `h2 ping h4` with the ICMP Replies also so the arp tables of all hosts remain the same like the previous 2 examples and the records are below.

```
mininet> h2 ping h3
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=12.6 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=64 time=0.213 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=64 time=0.118 ms
^C
--- 192.168.2.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2010ms
rtt min/avg/max/mdev = 0.118/4.305/12.584/5.854 ms
mininet> h2 ping h4
PING 192.168.2.3 (192.168.2.3) 56(84) bytes of data.
64 bytes from 192.168.2.3: icmp_seq=1 ttl=64 time=0.287 ms
64 bytes from 192.168.2.3: icmp_seq=2 ttl=64 time=0.073 ms
64 bytes from 192.168.2.3: icmp_seq=3 ttl=64 time=0.064 ms
^C
--- 192.168.2.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2053ms
rtt min/avg/max/mdev = 0.064/0.141/0.287/0.103 ms
mininet> h1 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:00:00:00:01:01   C                     h1-eth1
mininet> h2 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:00:00:00:01:01   C                     h2-eth1
mininet> h3 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.1              ether   00:00:00:00:02:01   C                     h3-eth1
mininet> h4 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.1              ether   00:00:00:00:02:01   C                     h4-eth1
```

Figure 32: `h2 ping h3`, `h2 ping h4` and ARP tables of all hosts

```
packet in 0x2  0x806 00:00:00:00:01:03 ff:ff:ff:ff:ff:ff in_port=3
packet in 0x1a 0x806 00:00:00:00:01:03 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:01:01 SourceIP: 192.168.1.1 to DestinationMAC: 00:00:00:00:01:03 DestinationIP: 192.168.1.3]

packet in 0x2  0x806 00:00:00:00:01:01 00:00:00:00:01:03 in_port=1
packet in 0x2  0x800 00:00:00:00:01:03 00:00:00:00:01:01 in_port=3
packet in 0x1a 0x800 00:00:00:00:03:02 00:00:00:00:03:01 in_port=1
Packet in left LAN: 192.168.2.2 ----> 192.168.1.3

packet in 0x1b 0x806 00:00:00:00:02:02 00:00:00:00:02:01 in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:02:01 SourceIP: 192.168.2.1 to DestinationMAC: 00:00:00:00:02:02 DestinationIP: 192.168.2.2]

packet in 0x1b 0x806 00:00:00:00:02:03 00:00:00:00:02:01 in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:02:01 SourceIP: 192.168.2.1 to DestinationMAC: 00:00:00:00:02:03 DestinationIP: 192.168.2.3]
```

Figure 33: Controller output

The flow table of router s1a has 3 records and router s1b has 3 records(with source MAC, the two MACs of two ports of the router and destination MAC, the MACs of the hosts or the MAC of the other router) after all runs with pings and are the below.

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1a
 cookie=0x0, duration=292.813s, table=0, n_packets=11, n_bytes=1078, ip,nw_dst=192.168.2.0/24
actions=mod_dl_src:00:00:00:00:03:01,mod_dl_dst:00:00:00:00:03:02,output:"s1a-eth1"
 cookie=0x0, duration=292.794s, table=0, n_packets=5, n_bytes=490, ip,nw_dst=192.168.1.2 actio
ns=mod_dl_src:00:00:00:00:01:01,mod_dl_dst:00:00:00:00:01:02,output:"s1a-eth2"
 cookie=0x0, duration=110.009s, table=0, n_packets=5, n_bytes=490, ip,nw_dst=192.168.1.3 actio
ns=mod_dl_src:00:00:00:00:01:01,mod_dl_dst:00:00:00:00:01:03,output:"s1a-eth2"
```

Figure 34: Flow table of s1a

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1b
 cookie=0x0, duration=344.876s, table=0, n_packets=5, n_bytes=490, ip,nw_dst=192.168.2.2 actio
ns=mod_dl_src:00:00:00:00:02:01,mod_dl_dst:00:00:00:00:02:02,output:"s1b-eth2"
 cookie=0x0, duration=263.175s, table=0, n_packets=5, n_bytes=490, ip,nw_dst=192.168.2.3 actio
ns=mod_dl_src:00:00:00:00:02:01,mod_dl_dst:00:00:00:00:02:03,output:"s1b-eth2"
 cookie=0x0, duration=344.863s, table=0, n_packets=11, n_bytes=1078, ip,nw_dst=192.168.1.0/24
actions=mod_dl_src:00:00:00:00:03:02,mod_dl_dst:00:00:00:00:03:01,output:"s1b-eth1"
```

Figure 35: Flow table of s1b

# 4 VLAN with OpenFlow

## 4.1 Topology

The topology of this part is the below.

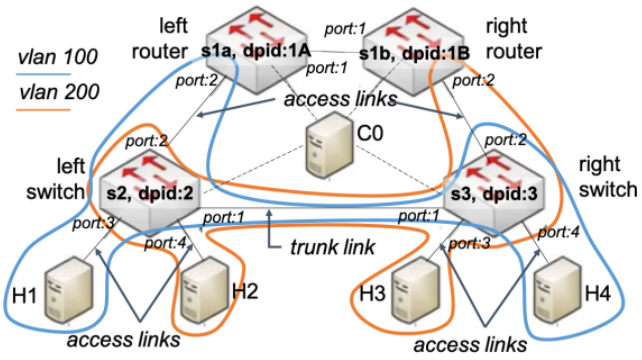


Figure 36: Part 4 Topology

The topology above is created with the script `mininet-router-vlan.py`.

```
root@mininet-vm:~# chmod 777 mininet-router-vlan.py
root@mininet-vm:~# ./mininet-router-vlan.py
```

Figure 37: Commands to create the topology

## 4.2 Explanation of the code

In this exercise, we added and changed code in the `mininet-router-vlan.py` and `ryu-router-two-frame.py` files and we renamed the files to `mininet-router-vlan-extended.py` and `vlan.py` respectively. Specifically, the code we added in the `mininet-router-vlan.py` extends the mininet topology to include an extra link between the routers s1a and s1b, using their port 4, as shown below.

```
#The second link between the routers both with port=4
net.addLink(s1a, s1b, port1=4, port2=4)
```

Figure 38: Extra link between two routers s1a and s1b

In `vlan.py` we have the left and right LAN ARP tables according to the IPs and MACs of the exercise and the s2 and s3 switch ports tables based on the different VLANs (100 and 200) and the trunk port.

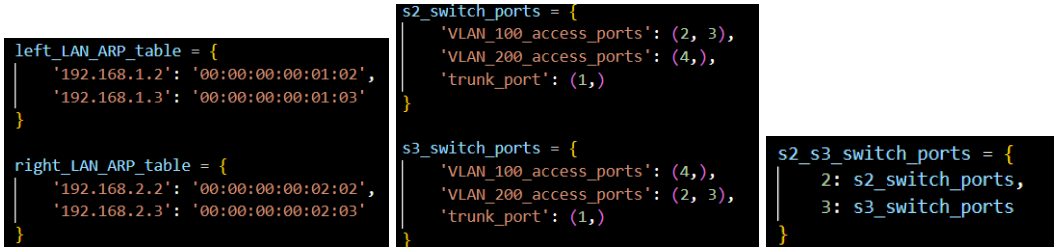


Figure 39: ARP tables and switch ports according the VLAN and trunk port

In the code below, the proactive forwarding of the high-priority traffic is implemented. To be more specific, the function `switch_features_handler` checks whether the packet came from s1a or s1b router and forwards it to the other router through port 4, to use the extra link between the routers s1a and s1b.

```python
#Proactive forwarding of the high-priority traffic(the traffic with non-zero ToS) between the routers
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    dpid = msg.datapath_id

    if dpid == 0x1A:
        self.add_proactive_tos_forwarding(
            dp = datapath,
            dest_ip = '192.168.2.0',
            source_mac = "00:00:00:00:05:01",
            dest_mac = "00:00:00:00:05:02",
            out_port = 4)
    elif dpid == 0x1B:
        self.add_proactive_tos_forwarding(
            dp = datapath,
            dest_ip = '192.168.1.0',
            source_mac = "00:00:00:00:05:02",
            dest_mac = "00:00:00:00:05:01",
            out_port = 4)
```

Figure 40: Proactive handling for high-priority packets in the new link between two routers

The `add_proactive_tos_forwarding` function sets up proactive forwarding rules for high-priority traffic, identified by a specific ToS value in the IP header. By defining these rules in advance, the function ensures that such traffic is handled with minimal delay, optimizing network performance. The function modifies the MAC addresses and directs the traffic to the appropriate output port, facilitating efficient and accurate delivery of high-priority packets.

```python
#Function that handles proactive forwarding for high priority traffic
def add_proactive_tos_forwarding(self, dp, dest_ip, source_mac, dest_mac, out_port):
    #Define the actions to be taken on the packet
    action_list = [
            dp.ofproto_parser.OFPActionSetDlSrc(source_mac),
            dp.ofproto_parser.OFPActionSetDlDst(dest_mac),
            dp.ofproto_parser.OFPActionOutput(out_port)
        ]

    #Create a match object to match IP packets with the specified destination IP and mask
    match = dp.ofproto_parser.OFPMatch(
            dl_type=ether_types.ETH_TYPE_IP,
            nw_dst=dest_ip, nw_dst_mask=24,
            nw_tos=8
        )

    #Add the flow entry to the switch with the specified match and actions
    self.add_flow(dp, match, action_list)
```

Figure 41: `add_proactive_tos_forwarding` function

The code below is explained in section 3. Static routing with two routers.

```python
if dpid == 0x1A:
    if ethertype == ether_types.ETH_TYPE_ARP: # this packet is ARP packet
        """
        fill in the code here
        """
        #3.Static routing with two routers
        if packet_arp.opcode == arp.ARP_REQUEST:
            if packet_arp.dst_ip == '192.168.1.1': #Left LAN
                self.arp_reply(
                    action_list = [datapath.ofproto_parser.OFPActionOutput(msg.in_port)],
                    dp=datapath,
                    source_mac="00:00:00:00:01:01",
                    dest_mac=src,
                    source_ip=packet_arp.dst_ip,
                    dest_ip=packet_arp.src_ip
                )
            else:
                return

        return
    elif ethertype == ether_types.ETH_TYPE_IP: # this packet is IP packet
        """
        fill in the code here
        """
        #3.Static routing with two routers
        dest_ip = packet_ip.dst
        if '192.168.1.' in dest_ip and dest_ip in left_LAN_ARP_table:
            self.logger.info("Packet in left LAN: %s ----> %s\n", packet_ip.src, dest_ip)
            self.ip_match_and_forward(datapath, "00:00:00:00:01:01", left_LAN_ARP_table[dest_ip], dest_ip, 32, 2, pkt)

        elif '192.168.2.' in dest_ip:
            self.logger.info("Packet in router 1B: %s ----> %s\n", packet_ip.src, dest_ip)
            self.ip_match_and_forward(datapath, "00:00:00:00:03:01", "00:00:00:00:03:02", dest_ip, 24, 1, pkt)
```

Figure 42: Same code with section 3

We also added a check if the packet is sent to an unreachable host. Both routers reply with an ICMP type 3, ”Destination Host Unreachable” packet, when they receive a packet that does not belong to the IP networks 192.168.1.0/24 and 192.168.2.0/24. We keep the actual data (payload) of the packet by removing the first 8 bytes and we add the icmp information in the new packet reply created by the function `icmp_reply`.

```
        else:
            #ICMP "Destination Host Unreachable" packet
            if msg.in_port == 1:
                source_mac = '00:00:00:00:03:01'
                dest_mac = '00:00:00:00:03:02'
            elif msg.in_port == 2:
                source_mac = '00:00:00:00:01:01'
                dest_mac = eth.src
            elif msg.in_port == 4:
                source_mac = '00:00:00:00:05:01'
                dest_mac = '00:00:00:00:05:02'

            self.logger.info("ICMP Reply Packet for Unreachable Destination: %s ----> %s\n", source_mac, dest_mac)
            data = msg.data[14:]
            #Extract the IP header and the first 8 bytes of the IP payload(info from transport layer)
            ip_header_len = 20  #Typical IP header length
            min_data_len = ip_header_len + 8
            if len(data) > min_data_len:
                data = data[:min_data_len]

            #Keeps the actual data(payload) of the packet and add icmp info in the new packet reply
            self.icmp_reply(
                action_list = [datapath.ofproto_parser.OFPActionOutput(msg.in_port)],
                data = data,
                source_mac = source_mac,
                dest_mac = dest_mac,
                source_ip = '192.168.1.1',
                packet_source_ip = packet_ip.src,
                dp = datapath
            )
            return
        return
    return
```

Figure 43: Functionality for ICMP Reply packet for Destination Host Unreachable

The `icmp_reply` function takes the parameters action_list, data, source_mac, dest_mac, packet_source_ip and dp (datapath). This function handles the creation and sending of an ICMP Destination Host Unreachable message in response to packets destined for an unknown IP address. It creates the icmp_reply_packet and sends it back to the packet_source_ip it came from.

```
#Function that handles the received packets in the routers that are destined to an unknown IP address
def icmp_reply(self, action_list, data, source_mac, dest_mac, source_ip, packet_source_ip, dp):
    #Get the OpenFlow protocol object for the datapath (switch)
    ofproto = dp.ofproto

    #Create a new packet for the ICMP reply
    icmp_reply_packet = packet.Packet()

    #Add an Ethernet frame to the packet with the appropriate ethertype, destination MAC, and source MAC
    icmp_reply_packet.add_protocol(
        ethernet.ethernet(
            ethertype=ether_types.ETH_TYPE_IP,
            dst=dest_mac,
            src=source_mac
        )
    )

    #Add an ARP protocol frame to the packet with the ARP reply opcode and the source and destination MAC and IP addresses
    icmp_reply_packet.add_protocol(
        ipv4.ipv4(
            dst=packet_source_ip,
            src=source_ip,
            proto=inet.IPPROTO_ICMP
        )
    )
```

Figure 44: `icmp_reply` function (1)

```
    #Add an ICMP protocol header to the icmp_reply packet
    icmp_reply_packet.add_protocol(
        icmp.icmp(
            type_=icmp.ICMP_DEST_UNREACH,
            code=icmp.ICMP_HOST_UNREACH_CODE,
            data=icmp.dest_unreach(
                data_len=len(data),
                data=data
            )
        )
    )

    #Serialize the packet to prepare it for sending
    icmp_reply_packet.serialize()

    #Create an OpenFlow PacketOut message to send the ICMP reply packet
    out = dp.ofproto_parser.OFPPacketOut(
        datapath=dp,
        buffer_id=ofproto.OFP_NO_BUFFER,
        in_port=ofproto.OFPP_CONTROLLER,
        actions=action_list,
        data=icmp_reply_packet.data
    )
    dp.send_msg(out)
```

Figure 45: `icmp_reply` function (2)

In this part of code we handle learning the source MAC address (src) to the corresponding switch port (`msg.in_port`) to prevent future flooding of packets. We identify the VLAN ID for the switch port by checking the incoming port against known VLAN configurations in s2_s3_switch_ports. If the destination MAC address (dst) is already known, it determines the output port (out_port) from the learned MAC-to-port mapping. If the packet is a VLAN-tagged packet (ETH_TYPE_8021Q), the VLAN ID is extracted and the VLAN tag is stripped before forwarding. When forwarding packets to trunk ports, the VLAN tag is added back with the appropriate VLAN ID. Finally, the packet is directed to the determined output port with the necessary VLAN encapsulation or decapsulation actions appended to the list of actions, ensuring proper handling of VLAN traffic across trunk and access ports.

```
# learn a mac address to avoid FLOOD next time.
self.mac_to_port[dpid][src] = msg.in_port

#Get vlan id of switch's port
for vlan_key in s2_s3_switch_ports[dpid]:
    if 'VLAN_' in vlan_key and msg.in_port in s2_s3_switch_ports[dpid][vlan_key]:
        vlanId = int(vlan_key.split('_')[1])  #Extract VLAN ID from the key
        break

actions = []
if dst in self.mac_to_port[dpid]:
    out_port = self.mac_to_port[dpid][dst]
    #Check if packet came from trunk port and extract vid (decapsulate)
    if ethertype == ether_types.ETH_TYPE_8021Q:
        vlanPacket = pkt.get_protocol(vlan.vlan)
        vlanId = vlanPacket.vid
        actions.append(datapath.ofproto_parser.OFPActionStripVlan())

    #If to be sent to trunk port tag it (encapsulate)
    if out_port in s2_switch_ports["trunk_port"] or out_port in s3_switch_ports["trunk_port"]:
        actions.append(datapath.ofproto_parser.OFPActionVlanVid(vlanId))
        actions.append(datapath.ofproto_parser.OFPActionOutput(out_port))
    else :
        actions.append(datapath.ofproto_parser.OFPActionOutput(out_port))
```

Figure 46: Functionality for setting the forwarding of switches (1)

On the other hand, if the MAC is unknown to the switch, this flooding mechanism ensures that the packet reaches its intended destination. By broadcasting the packet to all access and trunk ports (except the port it came from), the switch maximizes the chances of reaching the correct destination. The handling of VLAN tags ensures that VLAN-specific traffic is correctly encapsulated or decapsulated as needed when traversing between trunk and access ports.

```
else: #Flooding
    out_port = ofproto.OFPP_FLOOD
    #Flood to all access ports and trunk port but not to in-port
    for out_port_access in s2_s3_switch_ports[dpid]['VLAN_100_access_ports']:
        if out_port_access != msg.in_port:
            #Decapsulate if came from trunk port
            if ethertype == ether_types.ETH_TYPE_8021Q:
                actions.append(datapath.ofproto_parser.OFPActionStripVlan())
            actions.append(datapath.ofproto_parser.OFPActionOutput(out_port_access))

    for out_port_access in s2_s3_switch_ports[dpid]['VLAN_200_access_ports']:
        if out_port_access != msg.in_port:
            #Decapsulate if came from trunk port
            if ethertype == ether_types.ETH_TYPE_8021Q:
                actions.append(datapath.ofproto_parser.OFPActionStripVlan())
            actions.append(datapath.ofproto_parser.OFPActionOutput(out_port_access))

    #Flood to trunk port
    for out_port_trunk in s2_s3_switch_ports[dpid]['trunk_port']:
        if out_port_trunk != msg.in_port:
            #Decapsulate if came from trunk port
            if ethertype == ether_types.ETH_TYPE_8021Q:
                actions.append(datapath.ofproto_parser.OFPActionStripVlan())
            actions.append(datapath.ofproto_parser.OFPActionVlanVid(vlanId))
            actions.append(datapath.ofproto_parser.OFPActionOutput(out_port_trunk))
```

Figure 47: Functionality for setting the forwarding of switches (2)

```
#Function that handles proactive forwarding for high priority traffic
def add_proactive_tos_forwarding(self, dp, dest_ip, source_mac, dest_mac, out_port):
    #Define the actions to be taken on the packet
    action_list = [
            dp.ofproto_parser.OFPActionSetDlSrc(source_mac),
            dp.ofproto_parser.OFPActionSetDlDst(dest_mac),
            dp.ofproto_parser.OFPActionOutput(out_port)
        ]

    #Create a match object to match IP packets with the specified destination IP and mask
    match = dp.ofproto_parser.OFPMatch(
            dl_type=ether_types.ETH_TYPE_IP,
            nw_dst=dest_ip, nw_dst_mask=24,
            nw_tos=8
        )

    #Add the flow entry to the switch with the specified match and actions
    self.add_flow(dp, match, action_list)
```

Figure 48: Functionality for setting the forwarding of switches (3)

## 4.3 Examples

The first example is `h2 ping h4` and the important thing is that we try to move from VLAN 200 to VLAN 100. There is an ICMP Reply from h4 to h2 so the arp tables of the two hosts are created and the records are below.

```
mininet> h2 ping h4
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=187 ms
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=209 ms (DUP!)
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=0.889 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=0.242 ms
64 bytes from 192.168.1.3: icmp_seq=4 ttl=64 time=0.141 ms
^C
--- 192.168.1.3 ping statistics ---
4 packets transmitted, 4 received, +1 duplicates, 0% packet loss, time 3057ms
rtt min/avg/max/mdev = 0.141/79.391/208.864/96.965 ms
mininet> h2 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.1              ether   00:00:00:00:02:01   C                     h2-eth1
mininet> h4 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:00:00:00:01:01   C                     h4-eth1
```

Figure 49: `h2 ping h4` and ARP tables

```
packet in 0x2  0x806 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=4
packet in 0x3  0x8100 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=1
packet in 0x1a 0x806 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x1b 0x806 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:02:01 SourceIP: 192.168.2.1 to DestinationMAC
: 00:00:00:00:02:02 DestinationIP: 192.168.2.2]

packet in 0x3  0x806 00:00:00:00:02:01 00:00:00:00:02:02 in_port=2
packet in 0x2  0x8100 00:00:00:00:02:01 00:00:00:00:02:02 in_port=1
packet in 0x2  0x800 00:00:00:00:02:02 00:00:00:00:02:01 in_port=4
packet in 0x3  0x8100 00:00:00:00:02:02 00:00:00:00:02:01 in_port=1
packet in 0x1b 0x800 00:00:00:00:02:02 00:00:00:00:02:01 in_port=2
Packet in router 1A: 192.168.2.2 ----> 192.168.1.3

packet in 0x1a 0x800 00:00:00:00:03:02 00:00:00:00:03:01 in_port=1
Packet in left LAN: 192.168.2.2 ----> 192.168.1.3

packet in 0x2  0x800 00:00:00:00:01:01 00:00:00:00:01:03 in_port=2
packet in 0x3  0x8100 00:00:00:00:01:01 00:00:00:00:01:03 in_port=1
packet in 0x2  0x800 00:00:00:00:01:01 00:00:00:00:01:03 in_port=2
packet in 0x3  0x806 00:00:00:00:01:03 ff:ff:ff:ff:ff:ff in_port=4
packet in 0x2  0x8100 00:00:00:00:01:03 ff:ff:ff:ff:ff:ff in_port=1
packet in 0x3  0x8100 00:00:00:00:01:01 00:00:00:00:01:03 in_port=1
packet in 0x1b 0x806 00:00:00:00:01:03 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x1a 0x806 00:00:00:00:01:03 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:01:01 SourceIP: 192.168.1.1 to DestinationMAC
The ARP Reply is: SourceMAC: 00:00:00:00:01:01 SourceIP: 192.168.1.1 to DestinationMAC
: 00:00:00:00:01:03 DestinationIP: 192.168.1.3]

packet in 0x2  0x806 00:00:00:00:01:01 00:00:00:00:01:03 in_port=2
packet in 0x3  0x800 00:00:00:00:01:03 00:00:00:00:01:01 in_port=4
packet in 0x3  0x800 00:00:00:00:01:03 00:00:00:00:01:01 in_port=4
packet in 0x2  0x8100 00:00:00:00:01:03 00:00:00:00:01:01 in_port=1
packet in 0x2  0x8100 00:00:00:00:01:03 00:00:00:00:01:01 in_port=1
packet in 0x1a 0x800 00:00:00:00:01:03 00:00:00:00:01:01 in_port=2
Packet in router 1B: 192.168.1.3 ----> 192.168.2.2

packet in 0x1a 0x800 00:00:00:00:01:03 00:00:00:00:01:01 in_port=2
Packet in router 1B: 192.168.1.3 ----> 192.168.2.2

packet in 0x1b 0x800 00:00:00:00:03:01 00:00:00:00:03:02 in_port=1
Packet in right LAN: 192.168.1.3 ----> 192.168.2.2

packet in 0x1b 0x800 00:00:00:00:03:01 00:00:00:00:03:02 in_port=1
Packet in right LAN: 192.168.1.3 ----> 192.168.2.2
```

Figure 50: Controller output

The flow tables are shown below and as we see the packets from h2 go through the trunk link to s3 (VLAN 200), then up to s1b, s1a, down to s2 to enter VLAN 100 and through the trunk link again to reach h4.

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1a
 cookie=0x0, duration=180.867s, table=0, n_packets=0, n_bytes=0, ip,nw_dst=192.168.2.0/24,nw_tos=8 actions
=mod_dl_src:00:00:00:00:05:01,mod_dl_dst:00:00:00:00:05:02,output:4
 cookie=0x0, duration=170.267s, table=0, n_packets=4, n_bytes=392, ip,nw_dst=192.168.1.3 actions=mod_dl_sr
c:00:00:00:00:01:01,mod_dl_dst:00:00:00:00:01:03,output:"s1a-eth2"
 cookie=0x0, duration=170.158s, table=0, n_packets=3, n_bytes=294, ip,nw_dst=192.168.2.0/24 actions=mod_dl
_src:00:00:00:00:03:01,mod_dl_dst:00:00:00:00:03:02,output:"s1a-eth1"
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1b
 cookie=0x0, duration=199.186s, table=0, n_packets=0, n_bytes=0, ip,nw_dst=192.168.1.0/24,nw_tos=8 actions
=mod_dl_src:00:00:00:00:05:02,mod_dl_dst:00:00:00:00:05:01,output:4
 cookie=0x0, duration=188.776s, table=0, n_packets=4, n_bytes=392, ip,nw_dst=192.168.1.0/24 actions=mod_dl
_src:00:00:00:00:03:02,mod_dl_dst:00:00:00:00:03:01,output:"s1b-eth1"
 cookie=0x0, duration=188.632s, table=0, n_packets=3, n_bytes=294, ip,nw_dst=192.168.2.2 actions=mod_dl_sr
c:00:00:00:00:02:01,mod_dl_dst:00:00:00:00:02:02,output:"s1b-eth2"
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s2
 cookie=0x0, duration=210.711s, table=0, n_packets=5, n_bytes=510, in_port="s2-eth1",dl_vlan=200,dl_dst=00
:00:00:00:02:02 actions=strip_vlan,output:"s2-eth4"
 cookie=0x0, duration=210.591s, table=0, n_packets=3, n_bytes=306, in_port="s2-eth1",dl_vlan=100,dl_dst=00
:00:00:00:01:01 actions=strip_vlan,output:"s2-eth2"
 cookie=0x0, duration=210.703s, table=0, n_packets=3, n_bytes=294, in_port="s2-eth4",dl_dst=00:00:00:00:02
:01 actions=mod_vlan_vid:200,output:"s2-eth1"
 cookie=0x0, duration=210.619s, table=0, n_packets=3, n_bytes=294, in_port="s2-eth2",dl_dst=00:00:00:00:01
:03 actions=mod_vlan_vid:100,output:"s2-eth1"
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s3
 cookie=0x0, duration=212.531s, table=0, n_packets=5, n_bytes=490, in_port="s3-eth2",dl_dst=00:00:00:00:02
:02 actions=mod_vlan_vid:200,output:"s3-eth1"
 cookie=0x0, duration=212.414s, table=0, n_packets=3, n_bytes=294, in_port="s3-eth4",dl_dst=00:00:00:00:01
:01 actions=mod_vlan_vid:100,output:"s3-eth1"
 cookie=0x0, duration=212.509s, table=0, n_packets=3, n_bytes=306, in_port="s3-eth1",dl_vlan=200,dl_dst=00
:00:00:00:02:01 actions=strip_vlan,output:"s3-eth2"
 cookie=0x0, duration=212.447s, table=0, n_packets=5, n_bytes=472, in_port="s3-eth1",dl_vlan=100,dl_dst=00
:00:00:00:01:03 actions=strip_vlan,output:"s3-eth4"
```

Figure 51: Flow tables of s1a, s1b, s2 and s3

The second example, exit from the mininet and create it from the beginning, is `h2 ping h3` where we stay in the same VLAN 200 and has also an ICMP Reply from h3 to h2 so the arp tables of the two hosts are created and the records are below.

```
mininet> h2 ping h3
PING 192.168.2.3 (192.168.2.3) 56(84) bytes of data.
64 bytes from 192.168.2.3: icmp_seq=1 ttl=64 time=63.3 ms
64 bytes from 192.168.2.3: icmp_seq=2 ttl=64 time=0.567 ms
64 bytes from 192.168.2.3: icmp_seq=3 ttl=64 time=0.121 ms
64 bytes from 192.168.2.3: icmp_seq=4 ttl=64 time=0.124 ms
64 bytes from 192.168.2.3: icmp_seq=5 ttl=64 time=0.114 ms
64 bytes from 192.168.2.3: icmp_seq=6 ttl=64 time=0.131 ms
64 bytes from 192.168.2.3: icmp_seq=7 ttl=64 time=0.120 ms
64 bytes from 192.168.2.3: icmp_seq=8 ttl=64 time=0.118 ms
^C
--- 192.168.2.3 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7149ms
rtt min/avg/max/mdev = 0.114/8.071/63.277/20.866 ms
mininet> h2 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.3              ether   00:00:00:00:02:03   C                     h2-eth1
mininet> h3 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.2.2              ether   00:00:00:00:02:02   C                     h3-eth1
```

Figure 52: `h2 ping h3` and ARP tables

```
loading app vlan.py
loading app ryu.controller.ofp_handler
instantiating app vlan.py of SimpleSwitch
instantiating app ryu.controller.ofp_handler of OFPHandler
packet in 0x2  0x806 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=4
packet in 0x1a 0x806 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x3  0x8100 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=1
packet in 0x1b 0x806 00:00:00:00:02:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x3  0x806 00:00:00:00:02:03 00:00:00:00:02:02 in_port=3
packet in 0x2  0x8100 00:00:00:00:02:03 00:00:00:00:02:02 in_port=1
packet in 0x2  0x800 00:00:00:00:02:02 00:00:00:00:02:03 in_port=4
packet in 0x3  0x8100 00:00:00:00:02:02 00:00:00:00:02:03 in_port=1
```

Figure 53: Controller output

The flow tables are shown below and as we see the packets from h2 go through the trunk link to s3 (VLAN 200) and reach h3.

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1a
 cookie=0x0, duration=46.094s, table=0, n_packets=0, n_bytes=0, ip,nw_dst=192.168.2.0/24,nw_tos=8 actions=
mod_dl_src:00:00:00:00:05:01,mod_dl_dst:00:00:00:00:05:02,output:4
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1b
 cookie=0x0, duration=51.986s, table=0, n_packets=0, n_bytes=0, ip,nw_dst=192.168.1.0/24,nw_tos=8 actions=
mod_dl_src:00:00:00:00:05:02,mod_dl_dst:00:00:00:00:05:01,output:4
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s2
 cookie=0x0, duration=52.003s, table=0, n_packets=9, n_bytes=862, in_port="s2-eth1",dl_vlan=200,dl_dst=00:
00:00:00:02:02 actions=strip_vlan,output:"s2-eth4"
 cookie=0x0, duration=51.995s, table=0, n_packets=8, n_bytes=728, in_port="s2-eth4",dl_dst=00:00:00:00:02:
03 actions=mod_vlan_vid:200,output:"s2-eth1"
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s3
 cookie=0x0, duration=53.834s, table=0, n_packets=9, n_bytes=826, in_port="s3-eth3",dl_dst=00:00:00:00:02:
02 actions=mod_vlan_vid:200,output:"s3-eth1"
 cookie=0x0, duration=53.800s, table=0, n_packets=8, n_bytes=760, in_port="s3-eth1",dl_vlan=200,dl_dst=00:
00:00:00:02:03 actions=strip_vlan,output:"s3-eth3"
```

Figure 54: Flow tables of s1a, s1b, s2 and s3

The third example, exit from the mininet and create it from the beginning, is `h1 ping 192.168.1.6` where we try to connect to a host in the same VLAN (100) that does not exist so it will return a ”Host Unreachable” message.

```
mininet> h1 ping 192.168.1.6
PING 192.168.1.6 (192.168.1.6) 56(84) bytes of data.
From 192.168.1.2 icmp_seq=1 Destination Host Unreachable
From 192.168.1.2 icmp_seq=2 Destination Host Unreachable
From 192.168.1.2 icmp_seq=3 Destination Host Unreachable
From 192.168.1.2 icmp_seq=4 Destination Host Unreachable
From 192.168.1.2 icmp_seq=5 Destination Host Unreachable
From 192.168.1.2 icmp_seq=6 Destination Host Unreachable
^C
--- 192.168.1.6 ping statistics ---
7 packets transmitted, 0 received, +6 errors, 100% packet loss, time 6131ms
pipe 4
mininet> h1 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.6                      (incomplete)                              h1-eth1
```

Figure 55: `h1 ping 192.168.1.6` and ARP table

```
loading app vlan.py
loading app ryu.controller.ofp_handler
instantiating app vlan.py of SimpleSwitch
instantiating app ryu.controller.ofp_handler of OFPHandler
packet in 0x2  0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=3
packet in 0x3  0x8100 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=1
packet in 0x1a 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x1b 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x2  0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=3
packet in 0x3  0x8100 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=1
packet in 0x1a 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x1b 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x2  0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=3
packet in 0x1a 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x3  0x8100 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=1
packet in 0x1b 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x2  0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=3
packet in 0x1a 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x3  0x8100 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=1
```

Figure 56: Controller output

The fourth example, exit from the mininet and create it from the beginning, is `h1 ping 192.168.2.6` where we try to connect to a host in the other VLAN (200) that does not exist so it will return a ”Host Unreachable” message.

```
mininet> h1 ping 192.168.2.6
PING 192.168.2.6 (192.168.2.6) 56(84) bytes of data.
From 192.168.2.1 icmp_seq=1 Destination Host Unreachable
From 192.168.2.1 icmp_seq=2 Destination Host Unreachable
From 192.168.2.1 icmp_seq=3 Destination Host Unreachable
From 192.168.2.1 icmp_seq=4 Destination Host Unreachable
^C
--- 192.168.2.6 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3009ms

mininet> h1 arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   00:00:00:00:01:01   C                     h1-eth1
```

Figure 57: `h1 ping 192.168.2.6` and ARP table

```
packet in 0x2  0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=3
packet in 0x3  0x8100 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=1
packet in 0x1a 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
The ARP Reply is: SourceMAC: 00:00:00:00:01:01 SourceIP: 192.168.1.1 to DestinationMAC
: 00:00:00:00:01:02 DestinationIP: 192.168.1.2]

packet in 0x1b 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=2
packet in 0x2  0x806 00:00:00:00:01:01 00:00:00:00:01:02 in_port=2
packet in 0x2  0x800 00:00:00:00:01:02 00:00:00:00:01:01 in_port=3
packet in 0x1a 0x800 00:00:00:00:01:02 00:00:00:00:01:01 in_port=2
Packet in router 1B: 192.168.1.2 ----> 192.168.2.6

packet in 0x1b 0x800 00:00:00:00:03:01 00:00:00:00:03:02 in_port=1
ICMP Reply Packet for Unreachable Destination: 00:00:00:00:03:02 ----> 00:00:00:00:03:
01

packet in 0x1a 0x800 00:00:00:00:03:02 00:00:00:00:03:01 in_port=1
Packet in left LAN: 192.168.2.1 ----> 192.168.1.2

packet in 0x1b 0x800 00:00:00:00:03:01 00:00:00:00:03:02 in_port=1
ICMP Reply Packet for Unreachable Destination: 00:00:00:00:03:02 ----> 00:00:00:00:03:
01

packet in 0x1b 0x800 00:00:00:00:03:01 00:00:00:00:03:02 in_port=1
ICMP Reply Packet for Unreachable Destination: 00:00:00:00:03:02 ----> 00:00:00:00:03:
01
```

Figure 58: Controller output

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1a
 cookie=0x0, duration=105.594s, table=0, n_packets=0, n_bytes=0, ip,nw_dst=192.168.2.0/24,nw_tos=8 actions
=mod_dl_src:00:00:00:00:05:01,mod_dl_dst:00:00:00:00:05:02,output:4
 cookie=0x0, duration=93.859s, table=0, n_packets=3, n_bytes=294, ip,nw_dst=192.168.2.0/24 actions=mod_dl_
src:00:00:00:00:03:01,mod_dl_dst:00:00:00:00:03:02,output:"s1a-eth1"
 cookie=0x0, duration=93.846s, table=0, n_packets=3, n_bytes=210, ip,nw_dst=192.168.1.2 actions=mod_dl_src
:00:00:00:00:01:01,mod_dl_dst:00:00:00:00:01:02,output:"s1a-eth2"
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1b
 cookie=0x0, duration=109.430s, table=0, n_packets=0, n_bytes=0, ip,nw_dst=192.168.1.0/24,nw_tos=8 actions
=mod_dl_src:00:00:00:00:05:02,mod_dl_dst:00:00:00:00:05:01,output:4
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s2
 cookie=0x0, duration=100.388s, table=0, n_packets=4, n_bytes=280, in_port="s2-eth2",dl_dst=00:00:00:00:01
:02 actions=output:"s2-eth3"
 cookie=0x0, duration=100.379s, table=0, n_packets=3, n_bytes=294, in_port="s2-eth3",dl_dst=00:00:00:00:01
:01 actions=output:"s2-eth2"
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s3
mininet@mininet-vm:~$
```

Figure 59: Flow tables of s1a, s1b, s2 and s3

The last example (exit from the mininet and create it from the beginning) is `h2 send traffic with iperf -S 8 to h4` and now we run the `mininet-router-vlan-extended.py` to create an extra link for proactive forwarding of the high-priority traffic between the routers. Also we use the parameter -S 8 of iperf to produce non-zero ToS (high-priority) packets. The running of iperf -c(for client,that is h2) and iperf -s(for server, that is h4) we use the command xterm h2, and xterm h4 to open 2 new windows and run the iperf command.

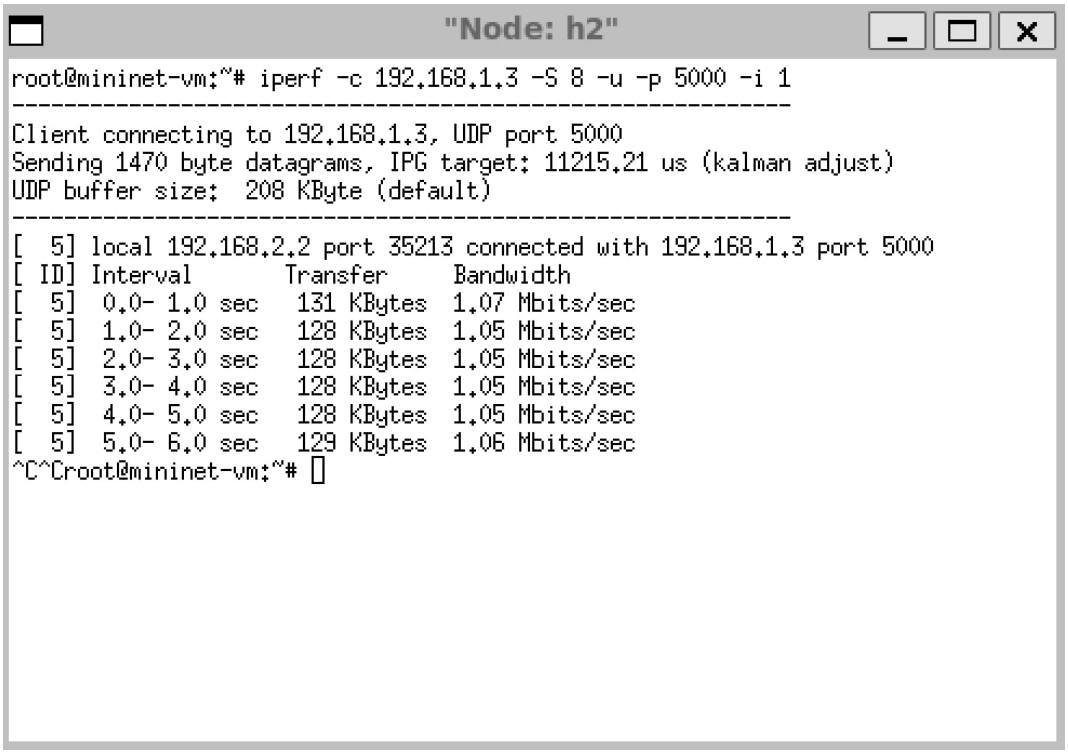


Figure 60: h2 sends traffic with iperf -S 8 to h4(192.168.1.3)

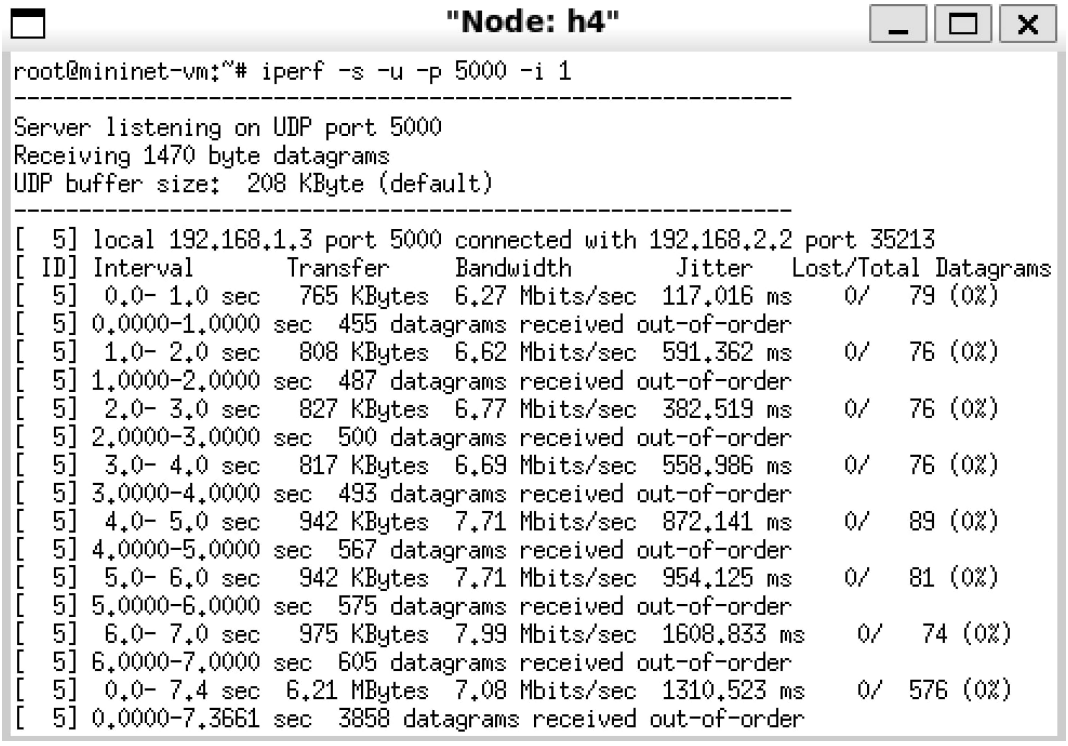


Figure 61: h4 receives traffic with iperf -s from h2

Also, we send traffic to an unknown destination in both VLANs to receive ”Destination Host Unreachable” message from `icmp_reply` function and uses the MAC addresses 00:00:00:00:05:01 and 00:00:00:00:05:02 because of the iperf -S 8(high-priority packets). If the packet goes from 00:00:00:00:05:02 to 00:00:00:00:05:01 the icmp reply will be from 00:00:00:00:05:01 to 00:00:00:00:05:02, vice versa the same applies.

```
root@mininet-vm:~# iperf -c 192.168.2.10 -S 8 -u -p 5000 -i 1
------------------------------------------------------------
Client connecting to 192.168.2.10, UDP port 5000
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  5] local 192.168.1.3 port 53788 connected with 192.168.2.10 port 5000
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 1.0 sec   131 KBytes  1.07 Mbits/sec
[  5]  1.0- 2.0 sec   128 KBytes  1.05 Mbits/sec
^C[  5] WARNING: did not receive ack of last datagram after 10 tries.
[  5]  0.0- 2.2 sec   276 KBytes  1.05 Mbits/sec
[  5] Sent 192 datagrams
root@mininet-vm:~#
```

Figure 62: h4 sends traffic to unknown destination in the other VLAN

```
packet in 0x1b 0x800 00:00:00:00:05:01 00:00:00:00:05:02 in_port=4
ICMP Reply Packet for Unreachable Destination: 00:00:00:00:05:02 ----> 00:00:00:00:05:01

packet in 0x1b 0x800 00:00:00:00:05:01 00:00:00:00:05:02 in_port=4
ICMP Reply Packet for Unreachable Destination: 00:00:00:00:05:02 ----> 00:00:00:00:05:01

packet in 0x1b 0x800 00:00:00:00:05:01 00:00:00:00:05:02 in_port=4
ICMP Reply Packet for Unreachable Destination: 00:00:00:00:05:02 ----> 00:00:00:00:05:01
```

Figure 63: Controller output

```
^C^Croot@mininet-vm:~# iperf -c 192.168.1.10 -S 8 -u -p 5000 -i 1
------------------------------------------------------------
Client connecting to 192.168.1.10, UDP port 5000
Sending 1470 byte datagrams, IPG target: 11215.21 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  5] local 192.168.2.2 port 57370 connected with 192.168.1.10 port 5000
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 1.0 sec   131 KBytes  1.07 Mbits/sec
^C^Croot@mininet-vm:~#
```

Figure 64: h2 sends traffic to unknown destination in the other VLAN

```
packet in 0x1a 0x800 00:00:00:00:05:02 00:00:00:00:05:01 in_port=4
ICMP Reply Packet for Unreachable Destination: 00:00:00:00:05:01 ----> 00:00:00:00:05:02

packet in 0x1a 0x800 00:00:00:00:05:02 00:00:00:00:05:01 in_port=4
ICMP Reply Packet for Unreachable Destination: 00:00:00:00:05:01 ----> 00:00:00:00:05:02

packet in 0x1a 0x800 00:00:00:00:05:02 00:00:00:00:05:01 in_port=4
ICMP Reply Packet for Unreachable Destination: 00:00:00:00:05:01 ----> 00:00:00:00:05:02

packet in 0x1a 0x800 00:00:00:00:05:02 00:00:00:00:05:01 in_port=4
ICMP Reply Packet for Unreachable Destination: 00:00:00:00:05:01 ----> 00:00:00:00:05:02
```

Figure 65: Controller output

The flow tables are shown below and as we see now the packets go through the new link that connects ports s1a-eth4 and s1b-eth4. The new link is created and the forwarding functionality proactively is added in the flow tables as explained previously.

```
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1a
 cookie=0x0, duration=343.394s, table=0, n_packets=0, n_bytes=0, ip,nw_dst=192.168.2.0/24,nw_t
os=8 actions=mod_dl_src:00:00:00:00:05:01,mod_dl_dst:00:00:00:00:05:02,output:"s1a-eth4"
 cookie=0x0, duration=8.128s, table=0, n_packets=2147, n_bytes=3240732, ip,nw_dst=192.168.1.3
actions=mod_dl_src:00:00:00:00:01:01,mod_dl_dst:00:00:00:00:01:03,output:"s1a-eth2"
 cookie=0x0, duration=5.015s, table=0, n_packets=0, n_bytes=0, ip,nw_dst=192.168.2.0/24 action
s=mod_dl_src:00:00:00:00:03:01,mod_dl_dst:00:00:00:00:03:02,output:"s1a-eth1"
mininet@mininet-vm:~$ sudo ovs-ofctl dump-flows s1b
 cookie=0x0, duration=380.550s, table=0, n_packets=2144, n_bytes=3241728, ip,nw_dst=192.168.1.
0/24,nw_tos=8 actions=mod_dl_src:00:00:00:00:05:02,mod_dl_dst:00:00:00:00:05:01,output:"s1b-et
h4"
 cookie=0x0, duration=42.192s, table=0, n_packets=0, n_bytes=0, ip,nw_dst=192.168.2.2 actions=
mod_dl_src:00:00:00:00:02:01,mod_dl_dst:00:00:00:00:02:02,output:"s1b-eth2"
 cookie=0x0, duration=42.179s, table=0, n_packets=0, n_bytes=0, ip,nw_dst=192.168.1.0/24 actio
ns=mod_dl_src:00:00:00:00:03:02,mod_dl_dst:00:00:00:00:03:01,output:"s1b-eth1"
```

Figure 66: Flow tables of s1a and s1b