Networks II: Final Project

Gavouras Dimitrios, Kafantaris Konstantinos, Moysis Moysis

July 2024

Networks II 2023-2024
Department of Electrical & Computer Engineering
University of Thessaly, Volos
email: {dgavouras, kkafantaris, mmoysis} @ e-ce.uth.gr

Abstract

This project involves the development of an OpenFlow controller using the Ryu framework to manage a network with two VLANs, interconnected through two static routers. The primary objectives include implementing ARP spoofing to respond to ARP requests, establishing static routing between subnets, and handling VLAN-tagged traffic between switches and routers. A critical feature implemented is the generation of ICMP "Destination Host Unreachable" messages for packets destined to unknown IP addresses. The project ensures that the network adheres to dynamic traffic management and routing protocols, providing a robust and scalable network infrastructure. The functionalities are tested and validated in a Mininet environment, ensuring practical application and reliability.

Contents

1 Introduction

This report presents the implementation and evaluation of an OpenFlow controller using the Ryu framework to address various networking tasks as part of a final project. The project is divided into four distinct parts, each focusing on different aspects of network management and control using OpenFlow v1.0 messages and structures. These parts include ARP spoofing, static routing, static routing with two routers, and VLAN configuration with OpenFlow.

2 ARP Spoofing

In the first part, an OpenFlow controller is developed to handle ARP requests within a network. The controller is designed to reply to ARP requests directly without forwarding them to the intended recipient. This is achieved by constructing ARP replies that mimic the responses from the target hosts, effectively spoofing the ARP responses. The implementation involves using a Mininet environment with specified MAC addresses for hosts and ensuring the controller can manage ARP tables and respond to ARP requests appropriately.

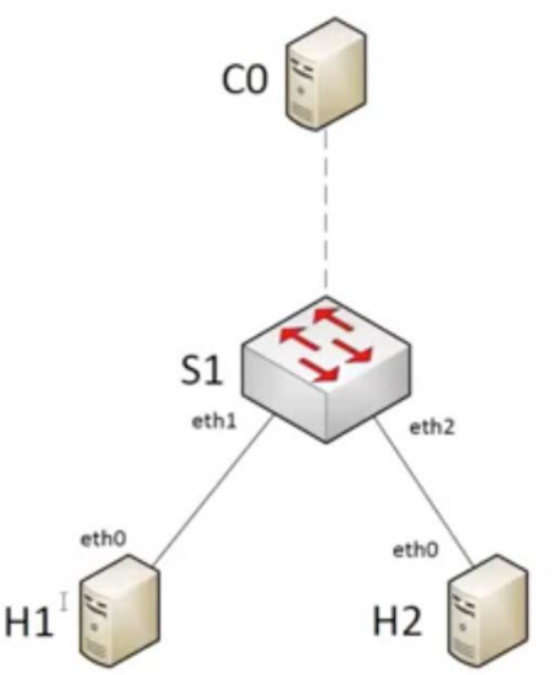

Figure 1: Mininet Network Topology

After we created a specific function to achieve the ARP Spoofing we call it in the **packet_in_handler** in the part where switch is used.

```
    if ethertype == 0x86dd:
            return

        if dpid == 1:
            if ethertype == ether_types.ETH_TYPE_ARP: # this packet is ARP packet
                arp_pkt = pkt.get_protocol(arp.arp)
                if arp_pkt.opcode == arp.ARP_REQUEST:
                    destinationMAC = MAC_TO_IP.get(arp_pkt.dst_ip)

                    self.send_arp_reply(datapath, msg.in_port, destinationMAC, "00:00:
                                                                    00:00:00:01", arp_pkt.
                                                                    dst_ip, arp_pkt.src_ip)
                return
```

The **send_arp_reply** function we created is implemented using the Ryu framework. This function constructs and sends ARP reply messages in response to ARP requests, effectively enabling ARP spoofing as required in this step. Below is a detailed explanation of the code and its role in the controller.

```
    def send_arp_reply(self, datapath, port, src_mac, dst_mac, src_ip, dst_ip):
       self.logger.info("[ARP Reply: [%s %s] -> [%s %s]] ",
     src_mac, src_ip, dst_mac, dst_ip)

       arp_reply = packet.Packet()
       arp_reply.add_protocol(ethernet.ethernet(
           ethertype=ether.ETH_TYPE_ARP,
           dst=dst_mac,
           src=src_mac
       ))
       arp_reply.add_protocol(arp.arp(
           opcode=arp.ARP_REPLY,
           src_mac=src_mac,
           src_ip=src_ip,
           dst_mac=dst_mac,
           dst_ip=dst_ip
       ))
       arp_reply.serialize()

       actions = [datapath.ofproto_parser.OFPActionOutput(port)]
       out = datapath.ofproto_parser.OFPPacketOut(
           datapath=datapath,
           buffer_id=datapath.ofproto.OFP_NO_BUFFER,
           in_port=datapath.ofproto.OFPP_CONTROLLER,
           actions=actions,
           data=arp_reply.data
       )
       datapath.send_msg(out)
```

- **Function Definition**

  `def send_arp_reply(self, datapath, port, src_mac, dst_mac, src_ip, dst_ip):`

  - **Parameters:**
    - `datapath`: Represents the switch datapath to which the message will be sent.
    - `port`: The output port number where the ARP reply will be sent.
    - `src_mac`: The source MAC address to be used in the ARP reply.
    - `dst_mac`: The destination MAC address (i.e., the MAC address of the host that sent the ARP request).
    - `src_ip`: The source IP address to be used in the ARP reply.
    - `dst_ip`: The destination IP address (i.e., the IP address of the host that sent the ARP request).

- **Creating the ARP Reply Packet**

```
arp_reply = packet.Packet()
arp_reply.add_protocol(ethernet.ethernet(
    ethertype=ether.ETH_TYPE_ARP,
    dst=dst_mac,
    src=src_mac
))
arp_reply.add_protocol(arp.arp(
    opcode=arp.ARP_REPLY,
    src_mac=src_mac,
    src_ip=src_ip,
    dst_mac=dst_mac,
    dst_ip=dst_ip
))
arp_reply.serialize()
```

  - `packet.Packet()`: Initializes a new packet object.
  - **Adding Ethernet Header**:
    - `ethertype=ether.ETH_TYPE_ARP`: Specifies that this packet is an ARP packet.
    - `dst=dst_mac`: Sets the destination MAC address.
    - `src=src_mac`: Sets the source MAC address.
  - **Adding ARP Header**:
    - `opcode=arp.ARP_REPLY`: Specifies that this is an ARP reply.
    - `src_mac=src_mac`: The MAC address of the host sending the ARP reply.
    - `src_ip=src_ip`: The IP address of the host sending the ARP reply.
    - `dst_mac=dst_mac`: The MAC address of the host that sent the ARP request.
    - `dst_ip=dst_ip`: The IP address of the host that sent the ARP request.
  - `arp_reply.serialize()`: Serializes the packet, preparing it for transmission.

- **Defining Actions and Sending the Packet Out**

```
actions = [datapath.ofproto_parser.OFPActionOutput(port)]
out = datapath.ofproto_parser.OFPPacketOut(
    datapath=datapath,
    buffer_id=datapath.ofproto.OFP_NO_BUFFER,
    in_port=datapath.ofproto.OFPP_CONTROLLER,
    actions=actions,
    data=arp_reply.data
)
datapath.send_msg(out)
```

  - **Actions**:
    - `datapath.ofproto_parser.OFPActionOutput(port)`: Specifies the action to output the packet to the specified port.
  - **Creating PacketOut Message**:
    - `datapath=datapath`: The datapath (switch) object.
    - `buffer_id=datapath.ofproto.OFP_NO_BUFFER`: Indicates that no buffer is used.
    - `in_port=datapath.ofproto.OFPP_CONTROLLER`: Indicates that the packet is sent from the controller.
    - `actions=actions`: The actions to be applied to the packet (i.e., send it out of the specified port).
    - `data=arp_reply.data`: The serialized ARP reply packet data.

- `datapath.send_msg(out)`: Sends the constructed PacketOut message to the datapath, which in turn sends the ARP reply out of the specified port.

- Simulation Example

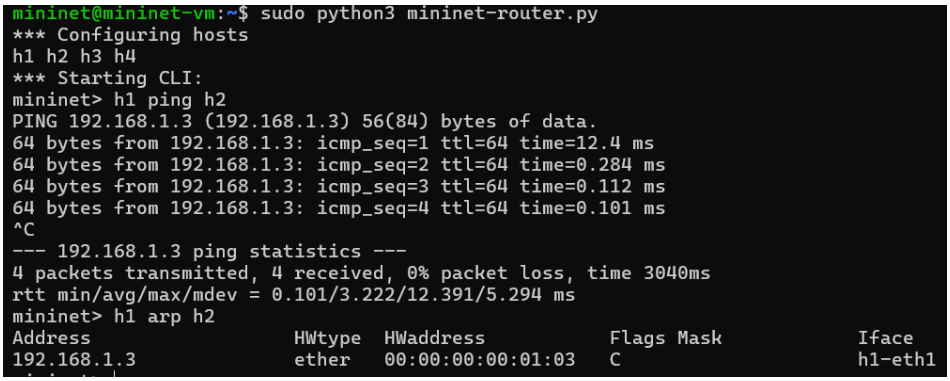

Figure 2: Example of the first task

3 Static Routing

The second part extends the controller's functionality to implement static routing between two LANs connected via a single router. The router, acting as a gateway, responds to ARP requests for its interfaces and forwards packets between the LANs based on their IP addresses. This involves setting up reactive flow rules and modifying Ethernet headers to enable seamless packet forwarding across the network.

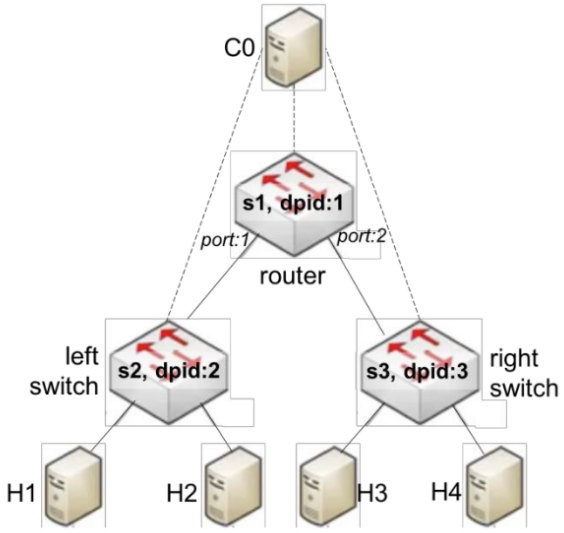

Figure 3: Mininet Network Topology

After we created a specific function to achieve the IP packet handling we call it in the **packet_in_handler** in the part where switch is used. In this part we check whether the packet type is ARP or IP and then call the functions we created.

```
    #  Exercise: if ethertype = 0x86dd return.
        if ethertype == 0x86dd:
            return

        if dpid == 1:
            # ARP Packet.
            if ethertype == ether_types.ETH_TYPE_ARP:
                arp_pkt = pkt.get_protocol(arp.arp)
                if arp_pkt.opcode == arp.ARP_REQUEST:
                    self.handle_arp(pkt, datapath, in_port, eth)
                return
            # IP packet.
            elif ethertype == ether_types.ETH_TYPE_IP:
                self.handle_ip(pkt, datapath, in_port, eth)
                return
            return
```

The `handle_ip` function is designed to manage the routing of IP packets within the network by making decisions based on the destination IP addresses and ensuring proper forwarding.

```
    # Handle IP packets.
    def handle_ip(self, pkt, datapath, in_port, eth):
        ip_pkt = pkt.get_protocol(ipv4.ipv4)
        print("ARP packet from: " + str(ip_pkt.src) + " to: " + str(ip_pkt.dst))

        # Get the destination MAC from the table.
        dst_mac = MAC_TO_IP.get(ip_pkt.dst)

        # Determine output port based on the destination IP
        if ip_pkt.dst.startswith('192.168.1'):
            # print("Left LAN " + dst_mac)
            eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:01:01",
                                                    ethertype=eth.ethertype)
            src_mac = "00:00:00:00:01:01"
            out_port = 1  # Assuming port 1 connects to 192.168.1.0/24 network
        elif ip_pkt.dst.startswith('192.168.2'):
            # print("Right LAN " + dst_mac)
            eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:02:01",
                                                    ethertype=eth.ethertype)
            out_port = 2  # Assuming port 2 connects to 192.168.2.0/24 network
            src_mac = "00:00:00:00:02:01"
        else:
            self.logger.info("Invalid")
            return

        ipv4_pkt = ipv4.ipv4(dst=ip_pkt.dst, src=ip_pkt.src, proto=ip_pkt.proto)

        # Add protocols to packet header.
        pkt.add_protocol(eth_pkt)
        pkt.add_protocol(ipv4_pkt)
        pkt.serialize()

        # Send packet.
        actions = [datapath.ofproto_parser.OFPActionSetDlSrc(src_mac),
                               datapath.ofproto_parser.OFPActionSetDlDst(dst_mac),
                               datapath.ofproto_parser.OFPActionOutput(out_port, 0)]
        match = datapath.ofproto_parser.OFPMatch(
            dl_type=ether_types.ETH_TYPE_IP,
            nw_src=ip_pkt.src,
            nw_dst=ip_pkt.dst
        )

        self.add_flow(datapath, match, actions)

        out = datapath.ofproto_parser.OFPPacketOut(
            datapath=datapath, buffer_id=datapath.ofproto.OFP_NO_BUFFER, in_port=
                                                    datapath.ofproto.
                                                    OFPP_CONTROLLER,
            actions=actions, data=pkt.data)
        datapath.send_msg(out)
```

- **Function Definition**

```
def handle_ip(self, pkt, datapath, in_port, eth):
```

  - **Parameters:**
    - `pkt`: The packet received by the controller.
    - `datapath`: The switch datapath to which the message will be sent.
    - `in_port`: The port on which the packet was received.
    - `eth`: The Ethernet frame containing the packet.

- **Extracting IP Packet Information**

```
ip_pkt = pkt.get_protocol(ipv4.ipv4)
print("ARP packet from: " + str(ip_pkt.src) + " to: " + str(ip_pkt.dst))
```

  - Retrieves the IPv4 protocol from the received packet.
  - Logs the source and destination IP addresses for monitoring purposes.

- **Determining Destination MAC Address and Output Port**

```
dst_mac = MAC_TO_IP.get(ip_pkt.dst)
```

  - Looks up the destination MAC address corresponding to the destination IP address from a predefined mapping (`MAC_TO_IP`).

- **Handling Different IP Ranges**

```
if ip_pkt.dst.startswith('192.168.1'):
    eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:01:01", ethertype=
                                                    eth.ethertype)
    src_mac = "00:00:00:00:01:01"
    out_port = 1  # Assuming port 1 connects to 192.168.1.0/24 network
elif ip_pkt.dst.startswith('192.168.2'):
    eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:02:01", ethertype=
                                                    eth.ethertype)
    out_port = 2  # Assuming port 2 connects to 192.168.2.0/24 network
    src_mac = "00:00:00:00:02:01"
else:
    self.logger.info("Invalid")
    return
```

  - Checks if the destination IP address belongs to the `192.168.1.0/24` or `192.168.2.0/24` subnet.
  - Sets the source MAC address and output port based on the destination subnet.
  - Constructs an Ethernet frame with the appropriate source and destination MAC addresses.

- **Creating and Adding Protocols to the Packet**

```
ipv4_pkt = ipv4.ipv4(dst=ip_pkt.dst, src=ip_pkt.src, proto=ip_pkt.proto)

# Add protocols to packet header.
pkt.add_protocol(eth_pkt)
pkt.add_protocol(ipv4_pkt)
pkt.serialize()
```

  - Constructs a new IPv4 packet header with the source and destination IP addresses and protocol type.
  - Adds the Ethernet and IPv4 headers to the packet and serializes it for transmission.

- **Defining Actions and Sending the Packet Out**

```
actions = [datapath.ofproto_parser.OFPActionSetDlSrc(src_mac),
           datapath.ofproto_parser.OFPActionSetDlDst(dst_mac),
           datapath.ofproto_parser.OFPActionOutput(out_port, 0)]
```

  - **Actions**:
    - `datapath.ofproto_parser.OFPActionSetDlSrc(src_mac)`: Sets the source MAC address for the outgoing packet.
    - `datapath.ofproto_parser.OFPActionSetDlDst(dst_mac)`: Sets the destination MAC address for the outgoing packet.
    - `datapath.ofproto_parser.OFPActionOutput(out_port, 0)`: Specifies the output port for the packet.

- **Matching and Adding Flow**

```
match = datapath.ofproto_parser.OFPMatch(
    dl_type=ether_types.ETH_TYPE_IP,
    nw_src=ip_pkt.src,
    nw_dst=ip_pkt.dst
)

self.add_flow(datapath, match, actions)
```

  - Creates a match object to define the flow rule, matching on the Ethernet type (IPv4) and the source and destination IP addresses.
  - Adds the flow to the switch, instructing it on how to handle similar packets in the future.

- **Sending the Packet Out**

```
out = datapath.ofproto_parser.OFPPacketOut(
    datapath=datapath, buffer_id=datapath.ofproto.OFP_NO_BUFFER, in_port=datapath.
                                                    ofproto.OFPP_CONTROLLER,
    actions=actions, data=pkt.data)
datapath.send_msg(out)
```

  - Constructs a PacketOut message to send the packet immediately.
  - Specifies the buffer ID, input port, actions, and the serialized packet data.
  - Sends the PacketOut message to the datapath, which forwards the packet out of the specified port.

- Simulation Example

```
mininet@mininet-vm:~$ sudo python3 mininet-router.py
*** Configuring hosts
h1 h2 h3 h4
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

Figure 4: Example of execution

```
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
ARP packet from: 192.168.1.2 to: 192.168.1.3
packet in 0x2 0x806 00:00:00:00:01:03 00:00:00:00:01:02 in_port=3
packet in 0x2 0x806 00:00:00:00:01:02 ff:ff:ff:ff:ff:ff in_port=1
packet in 0x2 0x800 00:00:00:00:01:02 00:00:00:00:01:03 in_port=2
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
ARP packet from: 192.168.1.2 to: 192.168.1.1
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
ARP packet from: 192.168.1.2 to: 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
ARP packet from: 192.168.2.2 to: 192.168.2.1
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
```

Figure 5: Result of execution

4 Static Routing with Two Routers

Building on the previous part, the third section introduces two routers to interconnect two LANs. Each router serves as a gateway for its respective LAN, handling ARP requests and forwarding packets between the LANs. The controllers are designed to set up flows that match network addresses and handle packet forwarding based on the destination IP addresses. The routers also manage inter-router communication to ensure efficient routing across the entire network.

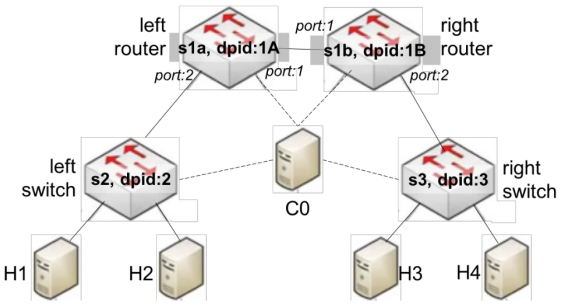

Figure 6: Mininet Network Topology

In this part we check in the **packet_in_handler** function the datapath ID (dpid) and the Ethernet type of incoming packets. If the dpid is 0x1A (left router) or 0x1B (right router), it processes ARP and IP packets by invoking the appropriate handler functions, with IP packets being routed differently for dpid 0x1A and 0x1B, by using the two different functions we created.

```
    if dpid == 0x1A:
        if ethertype == ether_types.ETH_TYPE_ARP: # this packet is ARP packet
            self.handle_arp(pkt, datapath, msg.in_port, eth)
            return
        elif ethertype == ether_types.ETH_TYPE_IP: # this packet is IP packet
            self.handle_ip(pkt, datapath, msg.in_port, eth)
            return
        return
    if dpid == 0x1B:
        if ethertype == ether_types.ETH_TYPE_ARP: # this packet is ARP packet
            self.handle_arp(pkt, datapath, msg.in_port, eth)
            return
        elif ethertype == ether_types.ETH_TYPE_IP: # this packet is IP packet
            self.handle_ip_right(pkt, datapath, msg.in_port, eth)
            return
        return
```

```
    # Handle IP packets.
    def handle_ip(self, pkt, datapath, in_port, eth):
        ip_pkt = pkt.get_protocol(ipv4.ipv4)
        print("ARP packet from: " + str(ip_pkt.src) + " to: " + str(ip_pkt.dst))

        # Get the destination MAC from the table.
        dst_mac = MAC_TO_IP.get(ip_pkt.dst)

        # Determine output port based on the destination IP
        if ip_pkt.dst.startswith('192.168.1'):
            # print("Left LAN " + dst_mac)
            eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:01:01",
                                        ethertype=eth.ethertype)
            src_mac = "00:00:00:00:01:01"
            out_port = 2  # Assuming port 1 connects to 192.168.1.0/24 network
        elif ip_pkt.dst.startswith('192.168.2'):
            # print("Right LAN " + dst_mac)
            eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:02:01",
                                        ethertype=eth.ethertype)
            out_port = 1  # Assuming port 2 connects to 192.168.2.0/24 network
            src_mac = "00:00:00:00:02:01"
```

```
        else:
            self.logger.info("Invalid")
            return

        ipv4_pkt = ipv4.ipv4(dst=ip_pkt.dst, src=ip_pkt.src, proto=ip_pkt.proto)

        # Add protocols to packet header.
        pkt.add_protocol(eth_pkt)
        pkt.add_protocol(ipv4_pkt)
        pkt.serialize()

        # Send packet.
        actions = [datapath.ofproto_parser.OFPActionSetDlSrc(src_mac),
                                datapath.ofproto_parser.OFPActionSetDlDst(dst_mac),
                                datapath.ofproto_parser.OFPActionOutput(out_port, 0)]
        match = datapath.ofproto_parser.OFPMatch(
            dl_type=ether_types.ETH_TYPE_IP,
            nw_src=ip_pkt.src,
            nw_dst=ip_pkt.dst
        )

        self.add_flow(datapath, match, actions)

        out = datapath.ofproto_parser.OFPPacketOut(
            datapath=datapath, buffer_id=datapath.ofproto.OFP_NO_BUFFER, in_port=
                                                    datapath.ofproto.
                                                    OFPP_CONTROLLER,
            actions=actions, data=pkt.data)
        datapath.send_msg(out)
```

```
    # Handle IP packets.
    def handle_ip_right(self, pkt, datapath, in_port, eth):
        ip_pkt = pkt.get_protocol(ipv4.ipv4)
        print("ARP packet from: " + str(ip_pkt.src) + " to: " + str(ip_pkt.dst))

        # Get the destination MAC from the table.
        dst_mac = MAC_TO_IP.get(ip_pkt.dst)

        # Determine output port based on the destination IP
        if ip_pkt.dst.startswith('192.168.1'):
            # print("Left LAN " + dst_mac)
            eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:01:01",
                                                    ethertype=eth.ethertype)
            src_mac = "00:00:00:00:01:01"
            out_port = 1  # Assuming port 1 connects to 192.168.1.0/24 network
        elif ip_pkt.dst.startswith('192.168.2'):
            # print("Right LAN " + dst_mac)
            eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:02:01",
                                                    ethertype=eth.ethertype)
            out_port = 2  # Assuming port 2 connects to 192.168.2.0/24 network
            src_mac = "00:00:00:00:02:01"
        else:
            self.logger.info("Invalid")
            return

        ipv4_pkt = ipv4.ipv4(dst=ip_pkt.dst, src=ip_pkt.src, proto=ip_pkt.proto)

        # Add protocols to packet header.
        pkt.add_protocol(eth_pkt)
        pkt.add_protocol(ipv4_pkt)
        pkt.serialize()

        # Send packet.
        actions = [datapath.ofproto_parser.OFPActionSetDlSrc(src_mac),
                                datapath.ofproto_parser.OFPActionSetDlDst(dst_mac),
                                datapath.ofproto_parser.OFPActionOutput(out_port, 0)]
        match = datapath.ofproto_parser.OFPMatch(
            dl_type=ether_types.ETH_TYPE_IP,
            nw_src=ip_pkt.src,
            nw_dst=ip_pkt.dst
        )
```

```
        self.add_flow(datapath, match, actions)

        out = datapath.ofproto_parser.OFPPacketOut(
            datapath=datapath, buffer_id=datapath.ofproto.OFP_NO_BUFFER, in_port=
                                                    datapath.ofproto.
                                                    OFPP_CONTROLLER,
            actions=actions, data=pkt.data)
        datapath.send_msg(out)
```

- **Function Definitions:**
  - `handle_ip`: Handles IP packets and routes them based on their destination IP addresses in the left router.
  - `handle_ip_right`: Similar to `handle_ip`, but handles routing logic for right router.
- **Similarities between `handle_ip` and `handle_ip_right`:**
  - **Extracting IP Packet Information:**

```
        ip_pkt = pkt.get_protocol(ipv4.ipv4)
        print("ARP packet from: " + str(ip_pkt.src) + " to: " + str(ip_pkt.
                                                    dst))
```

    * Retrieves the IPv4 protocol from the received packet.
    * Logs the source and destination IP addresses for monitoring purposes.
  - **Determining Destination MAC Address and Output Port:**

```
        dst_mac = MAC_TO_IP.get(ip_pkt.dst)
```

    * Looks up the destination MAC address corresponding to the destination IP address from a predefined mapping (`MAC_TO_IP`).
  - **Creating and Adding Protocols to the Packet:**

```
        ipv4_pkt = ipv4.ipv4(dst=ip_pkt.dst, src=ip_pkt.src, proto=ip_pkt.
                                                    proto)

        # Add protocols to packet header.
        pkt.add_protocol(eth_pkt)
        pkt.add_protocol(ipv4_pkt)
        pkt.serialize()
```

    * Constructs a new IPv4 packet header with the source and destination IP addresses and protocol type.
    * Adds the Ethernet and IPv4 headers to the packet and serializes it for transmission.
  - **Defining Actions and Sending the Packet Out:**

```
        actions = [datapath.ofproto_parser.OFPActionSetDlSrc(src_mac),
                   datapath.ofproto_parser.OFPActionSetDlDst(dst_mac),
                   datapath.ofproto_parser.OFPActionOutput(out_port, 0)]
```

    * **Actions:**
      - `datapath.ofproto_parser.OFPActionSetDlSrc(src_mac)`: Sets the source MAC address for the outgoing packet.
      - `datapath.ofproto_parser.OFPActionSetDlDst(dst_mac)`: Sets the destination MAC address for the outgoing packet.
      - `datapath.ofproto_parser.OFPActionOutput(out_port, 0)`: Specifies the output port for the packet.

– **Matching and Adding Flow:**

```
match = datapath.ofproto_parser.OFPMatch(
    dl_type=ether_types.ETH_TYPE_IP,
    nw_src=ip_pkt.src,
    nw_dst=ip_pkt.dst
)

self.add_flow(datapath, match, actions)
```

* Creates a match object to define the flow rule, matching on the Ethernet type (IPv4) and the source and destination IP addresses.
* Adds the flow to the switch, instructing it on how to handle similar packets in the future.

– **Sending the Packet Out:**

```
out = datapath.ofproto_parser.OFPPacketOut(
    datapath=datapath, buffer_id=datapath.ofproto.OFP_NO_BUFFER,
                                                    in_port=datapath.
                                                    ofproto.
                                                    OFPP_CONTROLLER,
    actions=actions, data=pkt.data)
datapath.send_msg(out)
```

* Constructs a PacketOut message to send the packet immediately.
* Specifies the buffer ID, input port, actions, and the serialized packet data.
* Sends the PacketOut message to the datapath, which forwards the packet out of the specified port.

- **Differences between `handle_ip` and `handle_ip_right`:**
  – **Determining Output Port Based on Destination IP:**

```
if ip_pkt.dst.startswith('192.168.1'):
    eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:01:01",
                                                     ethertype=eth.
                                                    ethertype)
    src_mac = "00:00:00:00:01:01"
    out_port = 2  # Assuming port 1 connects to 192.168.1.0/24
                                                    network
elif ip_pkt.dst.startswith('192.168.2'):
    eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:02:01",
                                                     ethertype=eth.
                                                    ethertype)
    out_port = 1  # Assuming port 2 connects to 192.168.2.0/24
                                                    network
    src_mac = "00:00:00:00:02:01"
else:
    self.logger.info("Invalid")
    return
```

* In `handle_ip`, if the destination IP is in the `192.168.1.0/24` subnet, the packet is sent out of port 2, and for the `192.168.2.0/24` subnet, it is sent out of port 1.
* In `handle_ip_right`, if the destination IP is in the `192.168.1.0/24` subnet, the packet is sent out of port 1, and for the `192.168.2.0/24` subnet, it is sent out of port 2.

– Simulation Example

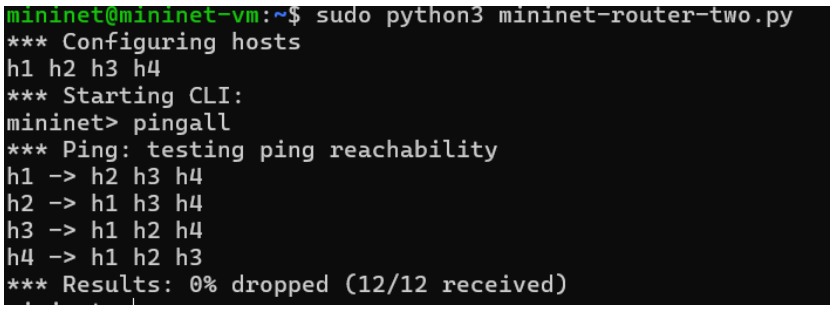

Figure 7: Execution example

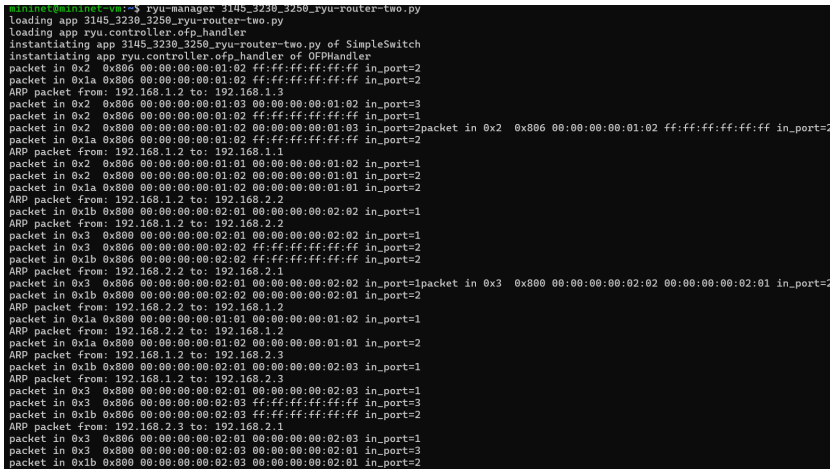

Figure 8: Executions Results(1)

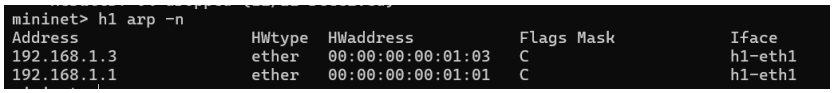

Figure 9: Executions Results(2)

5 VLAN with OpenFlow

The final part focuses on creating a network with VLAN segmentation using two interconnected switches and routers. The network is divided into two VLANs, each associated with a different IP network. The implementation ensures that packets are correctly encapsulated and decapsulated based on VLAN IDs as they pass through access and trunk links. Additionally, the routers are configured to handle high-priority traffic using a dedicated link and to reply with ICMP "Destination Host Unreachable" messages for unknown IP addresses.

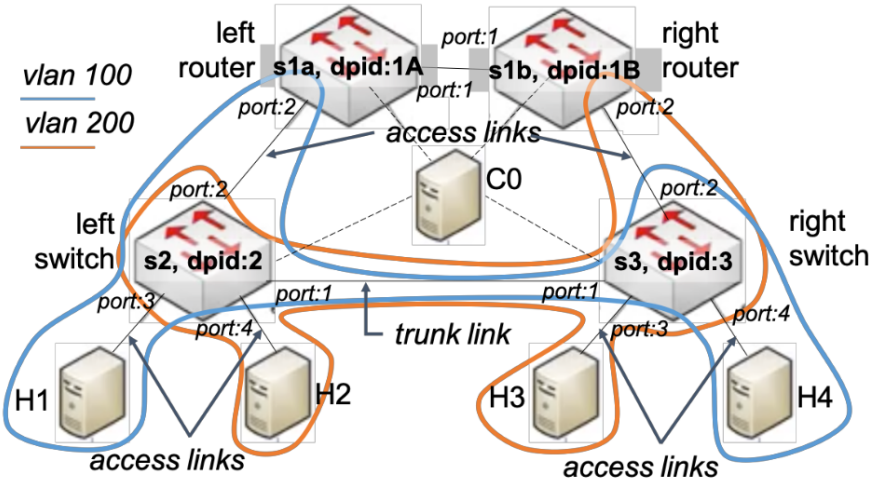

Figure 10: Mininet Network Topology

After we created a specific function to achieve the IP packet handling we call it in the **packet_in_handler** in the part where switch is used, like we said in the previous part. In this part we check whether the packet type is ARP or IP and then call the functions we created. Also for this part of the project we updated the `mininet_router_vlan.py` to succeed new connection between on the two router on port 4.

```
    # New link between the routers on port 4 (for priority packets).
net.addLink(s1a, s1b, port1=4, port2=4)
```

```
    if dpid == 0x1A:
            print("Router left")
            if ethertype == ether_types.ETH_TYPE_ARP: # this packet is ARP packet
                print("HANDLE ARP FROM 0X1A")
                arp_pkt = pkt.get_protocol(arp.arp)
                if arp_pkt.opcode == arp.ARP_REQUEST:
                    # ARP Packet must be destined for router
                    if arp_pkt.dst_ip != '192.168.1.1':
                        return
                    actions.append(datapath.ofproto_parser.OFPActionOutput(msg.in_port
                                                                           ))
                    self.handle_arp(pkt, datapath, msg.in_port, eth)
                return
            elif ethertype == ether_types.ETH_TYPE_IP: # this packet is IP packet
                print("HANDLE IP FROM 0X1A")
                self.handle_ip_left(pkt, datapath, msg.in_port, eth)
                return
            return
        if dpid == 0x1B:
            print("Router right")
            if ethertype == ether_types.ETH_TYPE_ARP: # this packet is ARP packet
                print("HANDLE ARP FROM 0X1B")
                arp_pkt = pkt.get_protocol(arp.arp)
                if arp_pkt.opcode == arp.ARP_REQUEST:
                    if arp_pkt.dst_ip != '192.168.2.1':
                        return
                    actions.append(datapath.ofproto_parser.OFPActionOutput(msg.in_port
                                                                           ))
                    self.handle_arp(pkt, datapath, msg.in_port, eth)
                return
            elif ethertype == ether_types.ETH_TYPE_IP: # this packet is IP packet
                print("HANDLE IP FROM 0X1B")
                self.handle_ip_right(pkt, datapath, msg.in_port, eth)
                return
            return
```

To correctly implement the Vlan function in our network we have to check if the packet flows in the same Vlan or it need send the packet to the appropriate port. Once more we are going to explain the functions that we used and implemented.

- **Function Definition**

  The basic function is that the switches handle the packets based on the Vlans from the ports that they are coming and to the ports that the packet needs to go. For example if the packet needs to travel in the same port the goes directly into the trunk link , but if the packet need to change vlan in needs to be send on the router to manage that.
  First we will talk about `handle_ip_left` and `handle_ip_right`. Both functions ultimately ensure that packets are correctly forwarded or replied to, depending on their destination and source, but they do so with different parameters and configurations appropriate to the left and right routers respectively.

  – `handle_ip_left`: This function handles the IP packets that come from the left router. It checks with suitable if and else where the packet should go to. The implementation was made basically by using the ports of its switch.

```
                # If packet goes to left router and comes from VLAN 100.
                if ip_pkt.dst.startswith('192.168.1') and ip_pkt.dst in
                                                        vlan_100_mappings:
                    eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00
                                                        :01:01",
                                                        ethertype=eth.
                                                        ethertype)
                    src_mac = "00:00:00:00:01:01"
                    out_port = 2
                #  If packet goes to right router.
                elif ip_pkt.dst.startswith('192.168.2'):
                    eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00
                                                        :02:01",
                                                        ethertype=eth.
                                                        ethertype)
                    src_mac = "00:00:00:00:02:01"
                    out_port = 1
                # We don't know the packet destination.
                else:
                    # Packet came from the other router.
                    if in_port == 1:
                        src_mac = '00:00:00:00:01:01'
                        dst_mac = '00:00:00:00:02:01'
                        src_ip = '192.168.1.1'
                        out_port = in_port
                    # Priority packets.
                    elif in_port == 4:
                        src_mac = '00:00:00:00:05:01'
                        dst_mac = '00:00:00:00:05:02'
                        src_ip = '192.168.1.1'
                        out_port = in_port
                    # Packet came from switches.
                    else:
                        print(eth.src)
                        src_mac = '00:00:00:00:01:01'
                        dst_mac = eth.src
                        src_ip = '192.168.1.1'
                        out_port = 2
```

  – `handle_ip_right`: This function handles the IP packets that come from the right router. It checks with suitable if and else where the packet should go to. The implementation was made with the exact same way.

```
            # If packet goes to left switch (so left router).
        if ip_pkt.dst.startswith('192.168.1'):
            eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:01:01",
                                                    ethertype=eth.
                                                    ethertype)
            src_mac = "00:00:00:00:01:01"
```

```
            out_port = 1
        # If the packet comes for VLAN 200 and comes from VLAN 200.
        elif ip_pkt.dst.startswith('192.168.2') and ip_pkt.dst in
                                                vlan_200_mappings:
            print("Mas gamas to paketo IP_right 192.168.2")
            eth_pkt = ethernet.ethernet(dst=dst_mac, src="00:00:00:00:02:01",
                                                    ethertype=eth.
                                                   ethertype)
            src_mac = "00:00:00:00:02:01"
            out_port = 2
        else:
            # If packet comes from the other router.
            if in_port == 1:
                src_mac = '00:00:00:00:02:01'
                dst_mac = '00:00:00:00:01:01'
                src_ip = '192.168.2.1'
                out_port = in_port
            # Priority packets.
            elif in_port == 4:
                src_mac = '00:00:00:00:05:02'
                dst_mac = '00:00:00:00:05:01'
                src_ip = '192.168.2.1'
                out_port = in_port
            # Packet comes from switches.
            else:
                src_mac = '00:00:00:00:02:01'
                dst_mac = eth.src
                src_ip = '192.168.2.1'
                out_port = 2
```

- Basic differences of the two IP handle functions
  - **Destination IP Handling:**
    - `Hanlde_ip_left`:
      - Handles IP packets destined for 192.168.1.* and 192.168.2.*.
      - If the destination IP starts with 192.168.2, it sets the source MAC to 00:00:00:00:02:01 and output port to 1.
      - If the packet's destination IP starts with 192.168.1 and is in `vlan_100_mappings`, it sets the source MAC to 00:00:00:00:01:01 and output port to 2.
    - `Handle_ip_right`:
      - Handles IP packets destined for 192.168.1.* and 192.168.2.*.
      - If the packet's destination IP starts with 192.168.1, it sets the source MAC to 00:00:00:00:01:01 and output port to 1.
      - If the destination IP starts with 192.168.2 and is in `vlan_200_mappings`, it sets the source MAC to 00:00:00:00:02:01 and output port to 2.
  - **Handling Unknown Destinations:**
    - `Hanlde_ip_left`:
      - If the packet is unknown, the source MAC and destination MAC are set based on whether it came from another router (in_port == 1), is a priority packet (in_port == 4), or came from switches. It defaults to sending ICMP unreachable messages with 00:00:00:00:01:01 as the source MAC.
    - `Handle_ip_right`:
      - If the packet is unknown, similar logic applies but with different MAC addresses and output ports. If the packet came from another router, it sets the source MAC to 00:00:00:00:02:01. Priority packets use MAC addresses 00:00:00:00:05:02 and 00:00:00:00:05:01.

- **Source and Destination MAC Addresses:**
  - `Hanlde_ip_left`:
    - Uses 00:00:00:00:01:01 as the source MAC for packets going to the left router and switches.
    - Uses 00:00:00:00:02:01 as the source MAC for packets going to the right router.
  - `Handle_ip_right`:
    - Uses 00:00:00:00:02:01 as the source MAC for packets going to the right router and switches.
    - Uses 00:00:00:00:01:01 as the source MAC for packets going to the left router.

- Switch for handling packet with priority

```
    def add_flow(self, datapath, match, actions):
        ofproto = datapath.ofproto

        mod = datapath.ofproto_parser.OFPFlowMod(
            datapath=datapath, match=match, cookie=0,
            command=ofproto.OFPFC_ADD, idle_timeout=0, hard_timeout=0,
            priority=ofproto.OFP_DEFAULT_PRIORITY,
            flags=ofproto.OFPFF_SEND_FLOW_REM, actions=actions)
        datapath.send_msg(mod)

    # -- HANDLE PACKET WITH PRIORITY --
    # Add new flows in order to handle future packets with priority (-S 8 ToS)
    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        dpid = datapath.id

        match, actions = None, []

        if dpid in [0x1A, 0x1B]:
            if dpid == 0x1A:
                match = self.create_ip_match(datapath, '192.168.2.0', 24, 8)
                actions = self.create_flow_actions(datapath, "00:00:00:00:05:01",
                                                   "00:00:00:00:05:02", 4
                                                   )
            elif dpid == 0x1B:
                match = self.create_ip_match(datapath, '192.168.1.0', 24, 8)
                actions = self.create_flow_actions(datapath, "00:00:00:00:05:02",
                                                   "00:00:00:00:05:01", 4
                                                   )

            self.add_flow(datapath, match, actions)

    def create_ip_match(self, datapath, nw_dst, nw_dst_mask, nw_tos):
        return datapath.ofproto_parser.OFPMatch(
            dl_type=ether_types.ETH_TYPE_IP, nw_dst=nw_dst, nw_dst_mask=
                                                  nw_dst_mask, nw_tos=nw_tos
                                                  )

    def create_flow_actions(self, datapath, src_mac, dst_mac, out_port):
        return [
            datapath.ofproto_parser.OFPActionSetDlSrc(src_mac),
            datapath.ofproto_parser.OFPActionSetDlDst(dst_mac),
            datapath.ofproto_parser.OFPActionOutput(out_port)
        ]
```

The code is used to prioritize certain IP packets by adding specific flow entries to the switches. Here's a detailed breakdown:

- When a switch connects, the `switch_features_handler` is triggered.

- If the switch has a DPID of 0x1A or 0x1B, flow entries are created to handle packets with a destination in the specified subnet and ToS 8.
- The `create_ip_match` function defines the match criteria for these packets.
- The `create_flow_actions` function specifies the actions to be taken, such as modifying the MAC addresses and forwarding the packet to a specific port.
- The `add_flow` function installs these flow entries in the switch.

This setup ensures that packets with a specific ToS value are handled with higher priority, allowing for differentiated services in the network.

- Handle_Vlan

```
    # Indicates we don't know packet destination port so it floods.
        flood_pkt = False
        # Indicates that packet came from trunk link.
        trunk_link_pkt = False

        vlan_100_ports = network_switch_ports[dpid]['vlan_100_mappings']
        vlan_200_ports = network_switch_ports[dpid]['vlan_200_mappings']
        handle_trunk_ports = network_switch_ports[dpid]['trunk_ports']

        # Learn MAC address for VLAN 100 if it doesn't exist
        if msg.in_port in vlan_100_ports and eth.src not in self.
                                                    mac_to_port_vlan_100[dpid]:
            self.mac_to_port_vlan_100[dpid][eth.src] = msg.in_port

        # Learn MAC address for VLAN 200 if it doesn't exist
        elif msg.in_port in vlan_200_ports and eth.src not in self.
                                                    mac_to_port_vlan_200[dpid]:
            self.mac_to_port_vlan_200[dpid][eth.src] = msg.in_port

        # Handle VLAN-tagged packets
        else:
            if eth.ethertype == ether_types.ETH_TYPE_8021Q:
                vlan_pkt = pkt.get_protocol(vlan.vlan)

                # Learn MAC address for VLAN 100 if VLAN ID is 100
                if vlan_pkt.vid == 100:
                    self.mac_to_port_vlan_100[dpid][eth.src] = msg.in_port
                # Learn MAC address for VLAN 200 if VLAN ID is 200
                else:
                    self.mac_to_port_vlan_200[dpid][eth.src] = msg.in_port

        # Handle packets from VLAN 100 access ports
        if msg.in_port in vlan_100_ports:
            vlan_id = 100

            if eth.dst in self.mac_to_port_vlan_100[dpid]:
                out_ports = [self.mac_to_port_vlan_100[dpid][eth.dst]]
            else:
                for access_port in vlan_100_ports:
                    if access_port != msg.in_port:
                        out_ports.append(access_port)

                for trunk_port in handle_trunk_ports:
                    out_ports.append(trunk_port)

                flood_pkt = True

        # Handle packets from VLAN 200 access ports
        elif msg.in_port in vlan_200_ports:
            vlan_id = 200
            if eth.dst in self.mac_to_port_vlan_200[dpid]:
                out_ports = [self.mac_to_port_vlan_200[dpid][eth.dst]]
            else:
                for access_port in vlan_200_ports:
                    if access_port != msg.in_port:
                        out_ports.append(access_port)
```

```
            for trunk_port in handle_trunk_ports:
                out_ports.append(trunk_port)

            flood_pkt = True

        # Handle packets from trunk ports
        else:
            if eth.ethertype == ether_types.ETH_TYPE_8021Q:
                # Packet came from trunk link so it goes to access link.
                # So the VLAN header must be removed.
                trunk_link_pkt = True

                vlan_pkt = pkt.get_protocol(vlan.vlan)
                vlan_id = vlan_pkt.vid

                if vlan_pkt.vid == 100:
                    if eth.dst in self.mac_to_port_vlan_100[dpid]:
                        out_ports = [self.mac_to_port_vlan_100[dpid][eth.dst]]
                    else:
                        out_ports.extend(vlan_100_ports)
                        if msg.in_port not in MAIN_TRUNK_PORT:
                            out_ports.extend([1])
                        flood_pkt = True

                elif vlan_pkt.vid == 200:
                    if eth.dst in self.mac_to_port_vlan_200[dpid]:
                        out_ports = [self.mac_to_port_vlan_200[dpid][eth.dst]]
                    else:
                        out_ports.extend(vlan_200_ports)
                        if msg.in_port not in MAIN_TRUNK_PORT:
                            out_ports.extend([1])
                        flood_pkt = True

            # Handle non-VLAN tagged packets
            else:
                match = datapath.ofproto_parser.OFPMatch(in_port=msg.in_port,
                                                         dl_dst=eth.dst)
                self.add_flow(datapath, match, [])
                return

        # Data to send
        data = None
        if msg.buffer_id == datapath.ofproto.OFP_NO_BUFFER:
            data = msg.data

        actions = []
        for out_port in out_ports:
            # Remove VLAN header.
            if trunk_link_pkt:
                actions.append(datapath.ofproto_parser.OFPActionStripVlan())

            # Add VLAN ID if it goes to trunk link.
            if out_port in MAIN_TRUNK_PORT:
                actions.append(datapath.ofproto_parser.OFPActionVlanVid(vlan_id))

            actions.append(datapath.ofproto_parser.OFPActionOutput(out_port))

        # Send packet out
```

The function is designed to handle and forward VLAN-tagged packets in an SDN network. Here is a detailed explanation of the key parts:

- **Initial Setup:**
  - `flood_pkt` is a flag to indicate whether the packet should be flooded.
  - `trunk_link_pkt` is a flag to indicate whether the packet came from a trunk link.
  - `vlan_100_ports`, `vlan_200_ports`, and `handle_trunk_ports` retrieve the respective

port mappings from the `network_switch_ports` dictionary.

- **Learning MAC Addresses:**
  - The function learns and stores the source MAC address for VLAN 100 and VLAN 200 if it is not already known.
  - It handles both tagged and untagged packets to ensure the MAC address is correctly learned.
- **Handling Packets from Access Ports:**
  - For packets coming from VLAN 100 or VLAN 200 access ports, it checks if the destination MAC address is known and sends the packet to the appropriate port.
  - If the destination MAC address is unknown, the packet is flooded to all access and trunk ports of the VLAN.
- **Handling Packets from Trunk Ports:**
  - For packets coming from trunk ports, it checks the VLAN ID and processes the packet accordingly.
  - The VLAN header is stripped if the packet is destined for an access port.
  - If the destination MAC address is unknown, the packet is flooded.
- **Sending the Packet:**
  - The function prepares the actions to be taken, including stripping the VLAN header if necessary and adding the VLAN ID for trunk ports.
  - The packet is then sent out to the appropriate ports using the `OFPPacketOut` message.

- Simulation Example

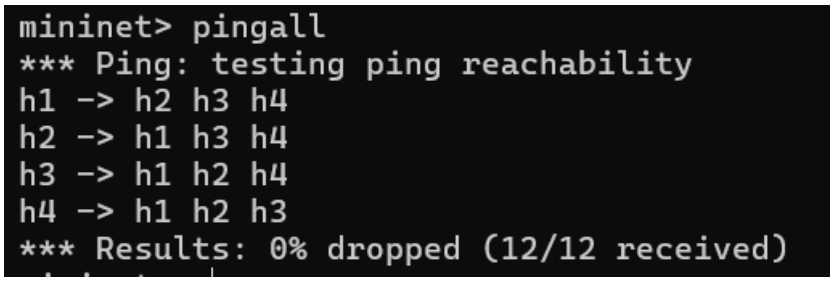

Figure 11: Execution example

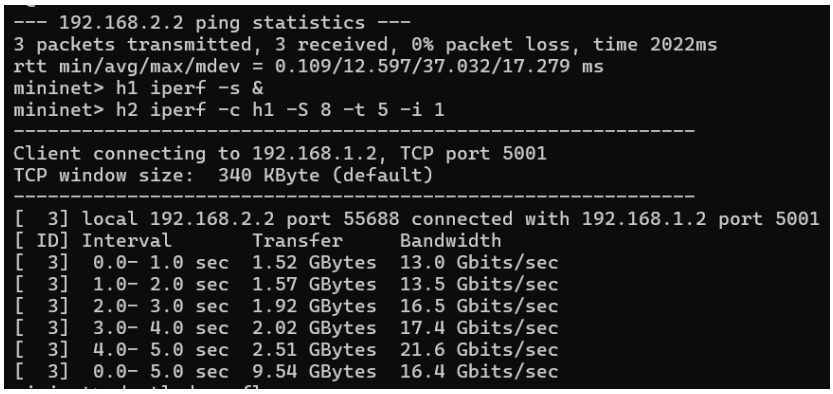

Figure 12: Priority packet execution from h1 to h2

The **dump-flows** command is typically used in the context of network devices and controllers to display the current flow entries in the flow tables of a switch. In the realm of Software-Defined Networking (**SDN**), particularly when using the OpenFlow protocol, flow entries dictate how packets should be handled by the switch.

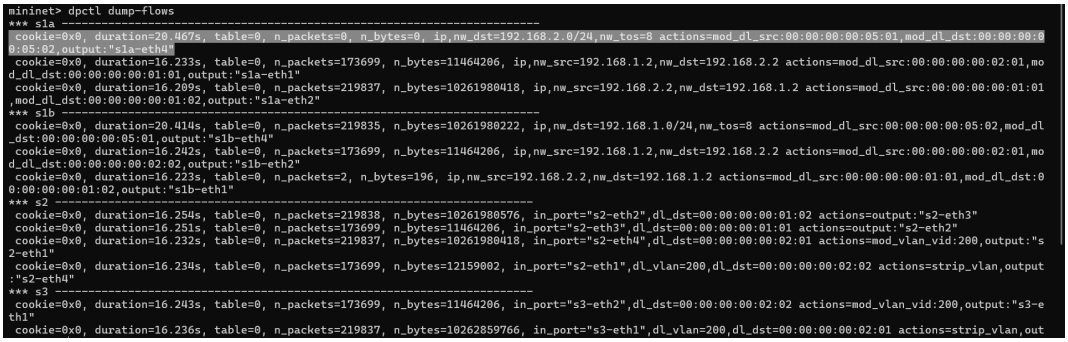

Figure 13: Dump flow execution (dpctl dump-flows)