Network Virtualization TP1 - OpenFlow

Lin I-Ping[pg41017], Erikson Tomas[pg44102], Eduardo Cunha[pg45515], and Miguel Peixoto[pg45517]

Engineering of Computer Networks and Telematic Services, School of Engineering, University of Minho, Braga, Portugal

Abstract. This work represents an introduction to the OpenFlow protocol. It focus in the development of Applications on top of Ryu SDN controller. Is composed by two exercises. In the first one, we developed an application that implements a "Layer-2 self-learning switch" on the OVSwitch. In the second exercise, we developed an application that implements a "Layer-3 switch" on the OVSwitch using a static routing approach in Software Defined Networks context.

Keywords: Network Virtualization · Software Defined Networks · SDN · OpenFlow · OVSwitch · Ryu SDN controller.

1 Introduction

To develop this work a Fedora 34 (kernel 5.11) virtual machine was created. To emulate the network the Mininet (version 2.3.0) is used. Among the two options proposed by the professor - Ryu or FloodLight - the Ryu (version 4.34) was chosen by the group due to the knowledge of the Python language among the elements of the group. A Python IDE was used to create and debug the scripts, in our case the Visual Studio Code (version 1.65.0). The OpenFlow version must be 1.3. The last software used is Wireshark (version 3.6.2), used in the assignment on the tests, confirming the results that were expected.

2 Exercise 1

The main goal of this exercise is to create an application that acts as a conventional layer 2 self-learning switch. The topology used is illustrated in Figure 1.

The Mininet topology was defined through a Python script. Firstly, the nodes are created, then the links between the hosts and switches. The topology file for the first exercise of TP1 is available in GitHub and represented in Figure 1.

2.1 Strategy

Ryu SDN controller has an example that meets the goals for this exercise 1 and is located at `~.local/lib/python3.9/site-packages/ryu/app/simple_`

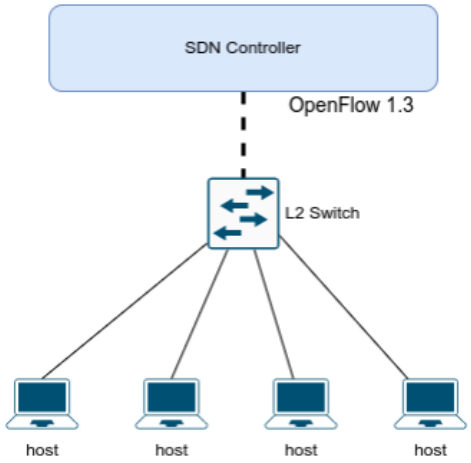

Fig. 1. Network topology

`switch_13.py`. Although we have this Ryu example available, we have made some changes to it to ensure we understand all the code and simplify the switch logic based on the proposed objective.

The **simple_switch_13** available on Ryu is an application that runs on the controller that controls the OVSSwitch. It is composed basically of one class and three primitives. The class has one object with - at this part - one attribute called **mac_to_port** that represents the MAC table of the switch.

The first primitive **switch_features_handler** is used during the handshake between the controller and the switch to configure the communication[1]. In this part installed a flow with the lowest priority possible - zero - to send the packet to the controller. When the packet arrives at the switch and there are no matches, it is sent to the controller to handle the packet correctly. We added an injection in the switch a flow to drop IPv6 packets at this part of the code.

The following one id the **add_flow** that is a primitive to inject a flow in the table of the switch. To inject a flow is necessary to inform the priority, the datapath to the switch extracted from the packet, the match expected and the action to be taken if the match occurs.

The last one is **_packet_in_handler** that handles packets that comes from the switch to the controller that is explained below.

The switch must learn the port for the source MAC address whenever an **Packet In** event occurs (the switch asks the controller what to do with the packet). If the destination MAC address is not inside the MAC table (python dictionary), the controller must trigger a flood to all ports on that switch. Otherwise, if the destination MAC address is inside the MAC table, the switch must forward the packet to the specific port related to the destination and the controller adds a flow in the switch so for the next time this source-destination

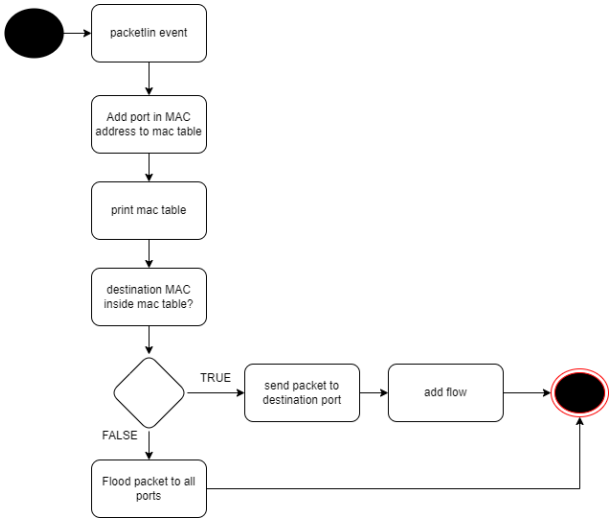

Fig. 2. Packet-In event

packet occurs, the Packet-In event will not be triggered again. The behavior could be easier understandable by looking at Figure 2.

2.2 Dry-Run

The first step is to ensure that we do not have any other containers running on the system:

```
sudo mn --clean
```

With this command, Mininet will stop any controller that is running and release any network interface used before.

Second step is to start the Ryu controller invoking ryu-manager with our code:

```
ryu-manager ryu.app.simple_switch_13
```

Then we are ready to start the Mininet with our topology with the command:

```
sudo mn --custom topology.py --topo mytopo --controller=remote
```

The **- - custom** indicates that a custom topology will be called and is necessary to pass the path to the topology script. The **- -mac** starts the hosts using its MAC addresses in ascending order, it is easier to understand the packets like that. The last argument **- - controller** equal to remote means that the controller will be initialized remotely. With no port being passed as a complementary argument, the Mininet expects the controller to run at the default port 6653.

2.3 Tests

With topology and controller running we have started Wireshark with the root user to be able to analyze the traffic in the links of the created topology.

Figure 3 shows the controller and the mininet running and figure 4 shows the initial entries in the flow table of the switch. As expected, there are two entries, one to drop the IPv6 packets and the other one to send packets to controllers when there are no matches.

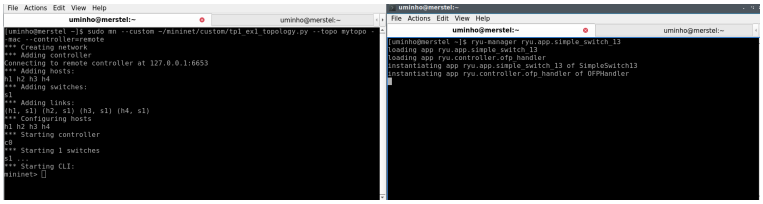

Fig. 3. Starting tests Exercise 1

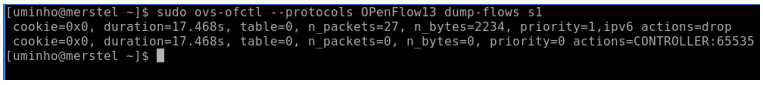

Fig. 4. Initial flow table s1 - Exercise 1

The interface to link the controller and the switch is the loopback (127.0.0.1). A filter is used to display only the OpenFlow version 1.3 protocol packets. The next figure shows all the OpenFlow packets captured during an execution of a ping between host H1 and the H2.

The sequence of the packets is exactly how was expected. Firstly, the H1 sends an Address Resolution Protocol request to try to find out what is the MAC address of the H2. The switch has nothing in the routing table, so the packet is forward to the controller (packet Nº 2). The controller first updates the MAC table with the in port and MAC address of H1, then check if the MAC address of H2 is known, as it is the first time the packets are being exchanged, the controller sends a packet to order the switch to flood the packet seeking to discover the MAC address(Nº 2). Then, the Address Resolution Protocol reply comes to the controller from H2 (Nº 3). Then, the controller updates the MAC table again and after that, a FLOW_MOD to add a new entry for the switch table with the pair **src - dst == H1 port - H2 port** is sent (Nº 4). The next packet (Nº 5) is the forward from the Address Resolution Protocol reply from the controller to the switch.

Then, with the Address Resolution Protocol accomplished, the ICMP packets start to flow. It happened in the next packet of the sequence (Nº 15) with an

ICMP request from H1 to H2 and a new entry is added to the switch table (Nº 16) with the pair **src - dst == H1 port - H2 port**. Then the reply (Nº 17) is forwarded to the H1. From now, packets will flow without any interference from the controller. The **.pcap** file is available and attached with this report to allow those who would like to see the packet's structure and order in more detail.

Checking the flow table again - figure 5 - there are two new entries that represent the flows from H1 to H2 and vice-versa. It shows that the L2 self-learning switch is working exactly as expected.

Fig. 5. Ping test Exercise 1

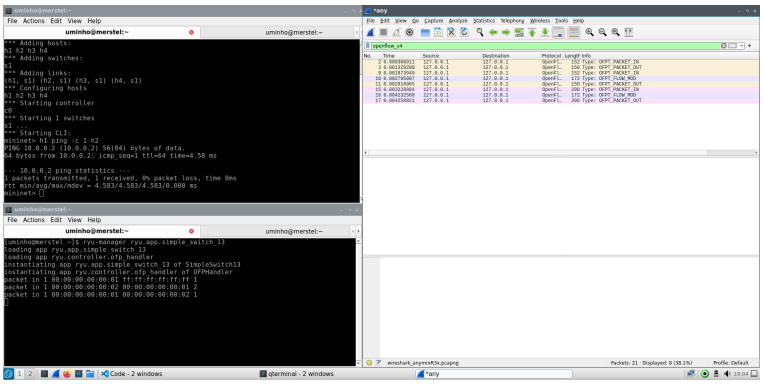

Fig. 6. Ping test on WireShark

3 Exercise 2

The main objective of this exercise is to create a Layer 3 forwarder/switch, which is not exactly a router as it does not decrement the IP Time To Live(TTL) and does not recalculate the checksum at each hop, therefore traceroute does not work. However, this 'router' can be either completely static or dynamic. Figure 10 shows the topology that this exercise must follow: use the Layer 2 self-learning application created in the previous exercise for the L2 switch and develop only a new application for the L3 switch. This exercise demonstrates that the network can be virtually layered by using OpenFlow-enabled transport devices. We can combine the functionality of switches, routers, and higher layers. If a packet goes to an IP address that recognizes the next hop, the Layer 2 destination needs to be changed to forward the packet to the correct port. The Topology follows Figure 10.

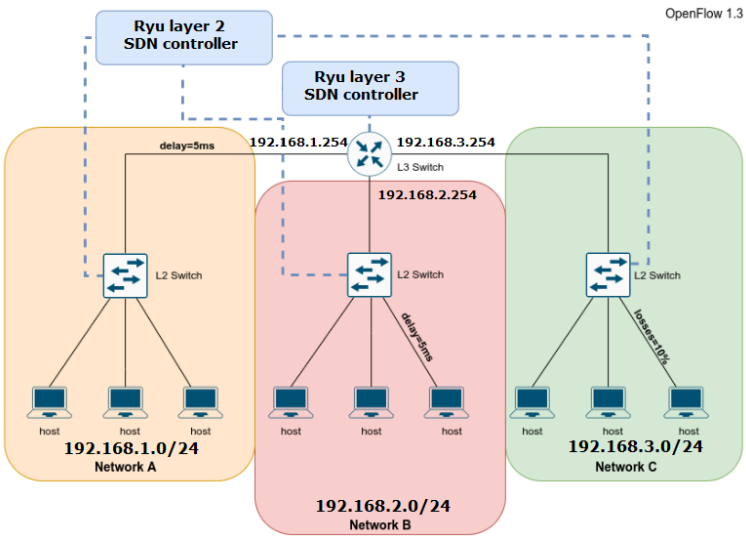

Fig. 7. Exercise 2 topology

A Python script is used to define the Mininet topology. First, nodes are created. Next, links are created and switched between hosts as was done in exercise 1.

However, this part is necessary to define the gateway for the hosts and the IPv4 address for the L3 switch ports. Another difference is that our group decides to work with 2 controllers, one only to control L2 switches - on port 6633 - and the other one to control the L3 - on port 6655.

The MAC address to each switch port and to each host is defined in the topology script as well, to make it easier to debug and understand the flows of the packets.

Another implementation in this topology is the addition of losses (h3_C - S3) and delay(R1 - S1 and h3_B - S2) on some links. The topology script is attached to this report and must be checked for more details.

Normally, the router has to respond to Address Resolution Protocol requests. An Ethernet broadcast is forwarded to the controller at least initially. The application needs to produce an Address Resolution Protocol response and forward it to the appropriate port. Required structure.

- Routing table (a structure where all information is statically allocated).
- Content Addressable Memory table - associative memory table
- Message queue (while the router waits for the Address Resolution Protocol response)

In addition, the application may receive Internet Control Message Protocol Echo (ping) requests to the router, to which it must respond. Finally, packets on unreachable subnets must be responded to with Internet Control Message Protocol Network Unreachable messages.

3.1 Strategy

To implement an L3 self-learning switch is necessary to implement fundamental changes in the previous application. Firstly, it is necessary to add an attribute to hold the pairs IP-MAC, we call it **ip_to_mac**. This variable is essentially a dictionary of a dictionary that follows this logic: {DPID:{IP: MAC}}, it saves the IP table to each L3 that the controller is linked.

It is necessary to define the IPv4 associated with each L3 switch port, and it follows the same logical approach: {DPID:{PORT: IP}}. The relation of IPs are defined as a global variable named **interface_port_to_ip**.

More two variables were created as attributes of the object. These variables are named as **L3_mac_to_port** and **L3_ip_to_port**, which as the names suggest holds the relation MAC - PORT and IP - PORT of each switch itself. The **L3_mac_to_port** is populated dynamically, in the **switch_features_handler** and routine - called **send_port_desc_stats_request** - that send to all the switches a request to get information about the ports, including the MAC address. Then, another routine - **port_desc_stats_reply_handler** - that is triggered when the reply is sent by the router is received by the controller populates the dictionary. the stats reply handler the **L3_ip_to_port** is populated as well using the global variable **interface_port_to_ip**. The global variable could be used, but to keep the standard used until the moment, our group decided that this approach was better and easier once the code is more complex.

An attribute to save the packets in the queue was created. It is named **queue** and it saves the packets as {DPID:{DST_IP:[in_port, src_mac, src_ip, msg]}}. The list inside the dictionary contains all the information already extracted that needs to forward to the packet later.

With these attributes implemented is already possible to implement the L3 switch needed to execute the exercise. The flow below explains in more detail how the controller handles the packets that the switches send to it.

When the switch sends a packet to the controller, the first verification to check if is an LLDP packet, if it is, the controller ignores it. However, if it is not, then the condition check if it is an ARP is done. If the packet is an ARP packet, then a routine called **arp_handler** is triggered. The ARP handler will be explained later in this section.

If it is not an ARP packet, then the controller verifies if it is an IP packet, if it is not, just ignore it. However, if it is true, then the MAC and IP table are updated with the information from the packet. The next step is to verify if the destination IP is known. If it is true, a new flow is added to the flow table and the packet is forward to the right port.

But, if the destination IP is not in the IP table, then the controller checks if the destination is for the L3 switch itself. If the statement is true, then it is necessary to check if it is an ICMP request, if it is then the controller sends an ICMP reply to the host that sent the ICMP request, if not, then the packet is ignored.

However, if the destination IP is not for the L3 switch or is not even inside the IP table, then the controller starts a routine called **flood_arp**. This routine enqueues the packet and searches the right port to send the ARP request to search for the destination in the right subnet. This routine will be explained separately later in this section as well.

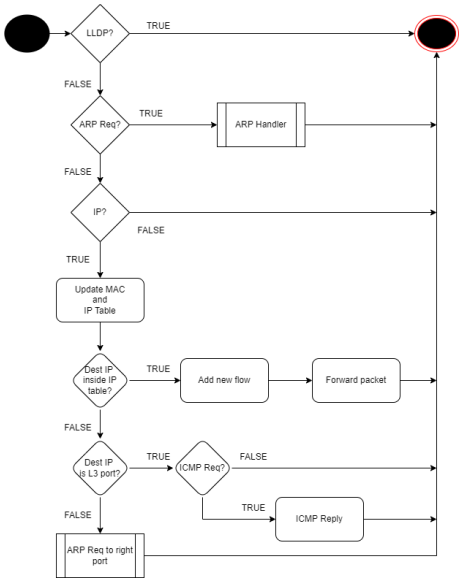

Fig. 8. Exercise 2 Packet-In event

The ARP handler is the routine that handles ARP packets. The first statement to check is if the packet is an ARP request. If it is and it is destined to

L3, then the controller prepares the reply and sends it. If it is not to some L3 switch port, then the packet is ignored.

However, if it is not an ARP request but an ARP reply, then the controller checks if the queue has any packets in it, if there are no packets, ignore it. Then the controller checks if there are some packet in the queue that matches the reply that just came, if no matches are found, the packet is ignored (a warning is printed in the console as a security message because a reply comes without any request). If the queue is empty, the controller prints a warning message as well for the same reason.

But, if the match is true, then the reply was requested by the controller, then the MAC and IP table is updated, the flow is added to the flow table, and the packet in the queue is forward to the original destination (updating the destination MAC address in the packet) and the packet is removed from the queue.

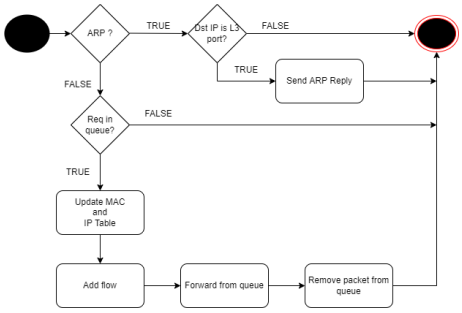

Fig. 9. ARP handler function

The **flood_arp** is quite simple, but it is an important part of the process. Firstly, the packet that the destination is unknown is queued. Then, the controller checks if the subnet of the destination is known. If the subnet is known, forward the ARP request to the right port, updating the MAC address of the source.

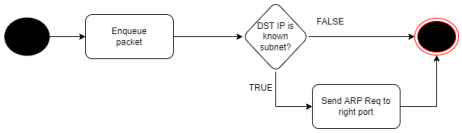

Fig. 10. ARP Request function

3.2 Dry-Run

The first step is to ensure that we don't have any other containers running on the system:

```
sudo mn --clean
```

With this command, Mininet will stop any running controller and release any network interface used before.

The second step is to start the Ryu controller on port 6633 for layer 2 switches invoking ryu-manager with our previous exercise code:

```
ryu-manager tp1_ex1_controller.py --ofp-tcp-listen-port 6633
```

The third step is to start the Ryu controller on port 6655 for the layer 3 switch invoking ryu-manager with our new exercise code:

```
ryu-manager ryu.app.L3_TP1 --ofp-tcp-listen-port 6655
```

Then we are ready to start the Mininet with our topology with the command:

```
sudo python3 VR/topology/tp1_ex2_topology.py
```

3.3 Tests

It runs on both topologies and controllers. Start Wireshark with root privileges so that traffic on the links of the created topology can be analysed.

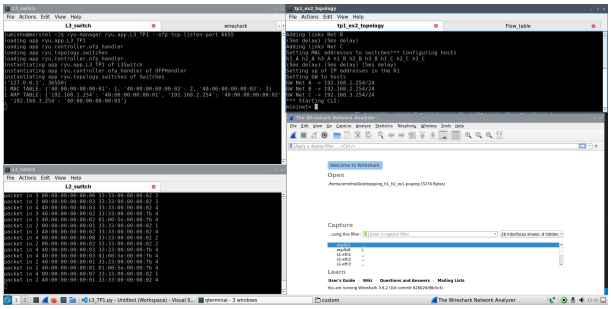

Fig. 11. Test start exercise 2

The interface between the controller and the switch is the loopback. A filter is applied to display only packets of the OpenFlow version 1.3 protocol that are ICMP, ARP, or OpenFlowMod. The following figure 12 shows all OpenFlow packets captured in ping between hosts H1_A and H1_B. The order of the packets is exactly how was expected and the **.pcap** file is attached with this report in case the reader is interested to see it in detail.

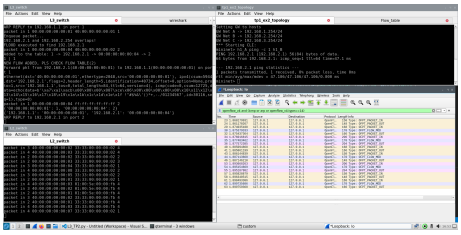

Fig. 12. Ping test on WireShark - exercise 2

The flow table shows that the entries were added to the flow table - figure 13 as expected and it means that new packets that match the flows from H1_A to H1_B will not be handled by the controller anymore - in our case as we apply the matches based on the entire network itself and not only in the IPv4 of the hosts, any flow from net A to net B will flow without any interference of the controller anymore.

Fig. 13. Flow table R1 - exercise 2

So, the forwarding, the queue, and the addition of entries in the flow table are working properly. The last test is to check if the controller responds to the ICMP requests to its port. The test is executed by running the ping test to each port of the L3 switch - figure 14. As it is shown, the ICMP requests always are replied to.

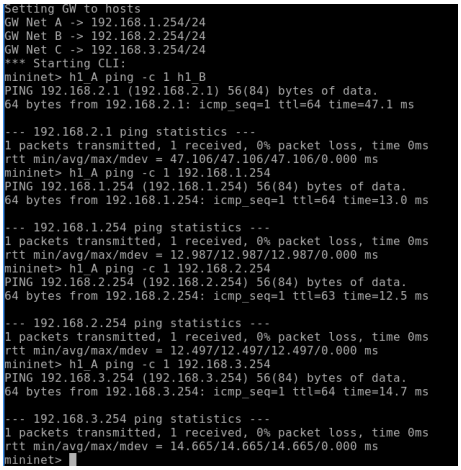

Fig. 14. Ping test to L3's ports - exercise 2

4 Conclusion

Carrying out this work we have faced some difficulties, namely OpenFlow and the API that Ryu makes available to interact with it, mainly due to the poor documentation available. The Ryu website provides an API reference, but with almost no examples, we knew which primitives we needed to use, but couldn't find implementation examples. This trial-and-error approach led us to spend a lot of time on exercise 2 and resulted in the greatest difficulty encountered in carrying out this practical work. On the other hand, having an example available for exercise 1 facilitated the process of self-learning and introduction to OpenFlow with the Ryu controller.

The controller replies to the ICMP requests and we think that injecting a flow in the L3 switch to create a reply packet for pings to its port could be an improvement to be implemented in the future. However, due to the lack of examples in documentation, we could not find out how to implement it.

Another important aspect of our controller to be mentioned is that Idle timeout and hard time out are not set, this approach makes the entries in the flow permanent. As this exercise is a simple one in our opinion it was not important to handle with these timers. However, we do know how to implement it. It is only necessary to set **dle_timeout** and **hard_timeout** in the OFPFlowMod routine while the packet is being created to inject a new entry in the switch.

References

[1] R. team. *RYU SDN Framework - English Edition*. Release 1.0. RYU project team, 2014. URL: `https://books.google.pt/books?id=JC3rAgAAQBAJ` (cit. on p. 2).