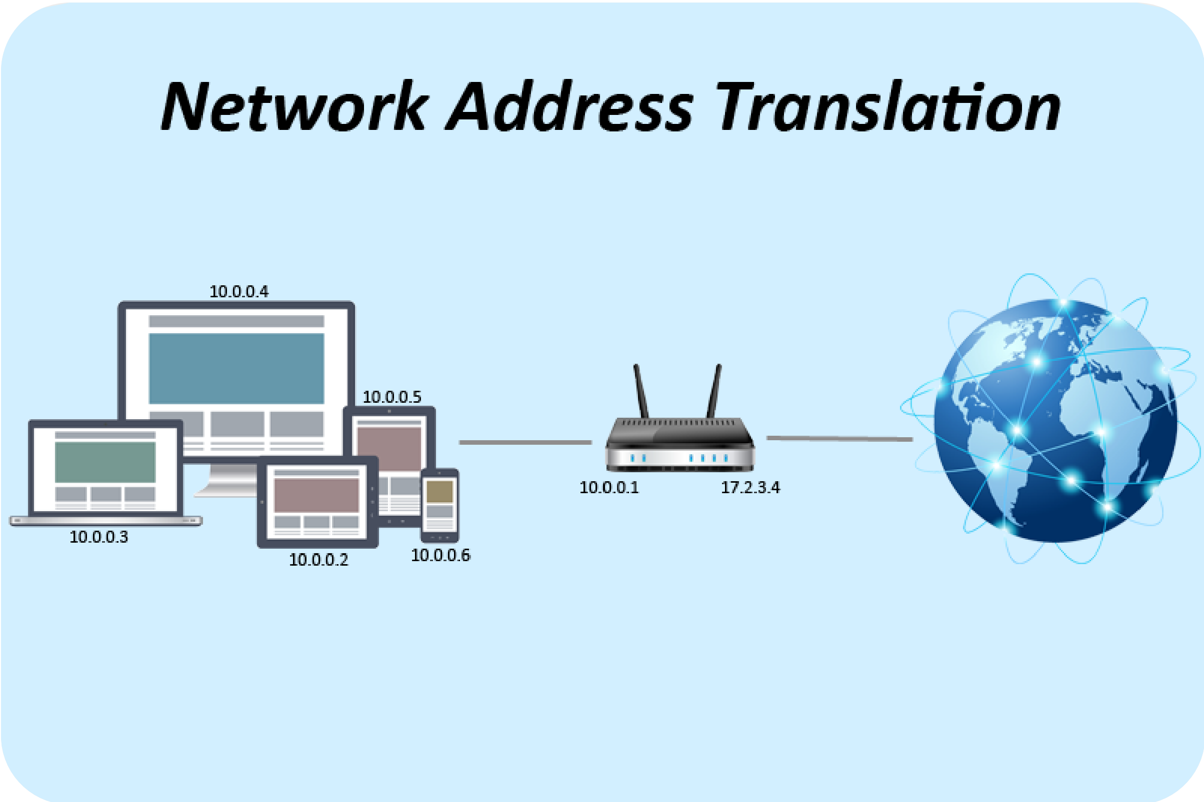
Network Address Translation (NAT):



Group 1
Kathrina Arrocha Umpiérrez
Edgar Mesa Santana
Jorge Gutiérrez Reyes
Sergio Ravelo Vegino

Index:

Introduction ---------- 2
Downloads ---------- 2
Software Defined Networking SDN ---------- 3
OpenFlow Switch ---------- 3

Comparison of controllers ---------- 4
Controller selection ---------- 5

Functional description of the device ---------- 5
Flowchart ---------- 6

Implementation ---------- 7
Topology ---------- 7
Switch ---------- 8

Testing in Mininet and Wireshark ---------- 9

Annexed ---------- 11
Gantt diagram ---------- 11

Introduction:

The objective of this practice is to understand how NAT works and how we can implement it. To fulfill our goal we will have to use some programs such as Mininet.

- **Downloads:**

First at all, we need to install mininet if we don't have it, so here are the steps that we will need to the installation:

There are 3 options that we can use, but we will define the opcion of *native installation from source:*

1) To install natively from source, first you need to get the source code:

```
git clone git://github.com/mininet/mininet
```

2) Next, we have to select the version we want to install:

```
cd mininet
git tag # list available versions
git checkout -b 2.2.1 2.2.1  # or whatever version you wish to install
cd ..
```

3) Once you have the source tree, the command to install Mininet is:

```
mininet/util/install.sh -a
```

We use -a because we want to install everything. You can find more information about the different parameters we can use in this link: http://mininet.org/download/

To execute Mininet we can use the next command: *sudo mn*

Once we have installed Mininet we can start looking for the best controller that suits our preferences. There are a lot of good options, but we will be picking Ryu. Ryu it's a controller based on Python, so we need to install python if we haven't yet. We will see in more detail later on why we decided to use this controller. To install python 3 use:

```
sudo apt install python3
Now we have to use Ryu so we have to use:
sudo pip install ryu
sudo apt-get install python-ryu
```

Last, but not least, we use the next command to turn on our controller. Example, if our project's name is Nat.py and it is inside Desktop:

```
~$ cd Desktop
~/Desktop$ ryu-manager Nat.py –verbose
--verbose: Show debug output.
```

Now, we have all that we needed.

- **Software Defined Networking SDN:**

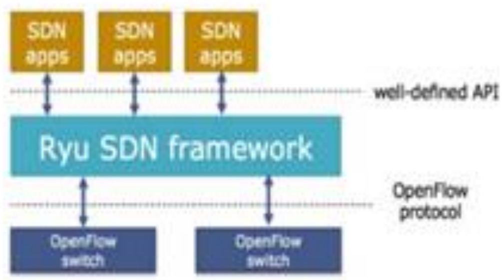
The SDN networks have as a global objective to open the control of the flows of the networks towards a software, independently of the specific hardware in each one. SDN separates the control plane from the data plane, which does not occur in traditional data networks. The objective is to dissociate the network infrastructure of the application and the services present in it, which see the network as a virtual or logical entity. SDN covers multiple types of network technologies designed to make the network more flexible and agile to support the virtualized server and storage infrastructure of the modern data center. The software defined network originally defined an approach to design, build and manage networks that separates the network's control or SDN network policy (brains) and forwarding (muscle) planes thus enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services for applications as SDN cloud computing or mobile networks.

SDN Controller is the central core of the architecture SDN, since it is the one that has all the logic of the network. The controller defines the rules for managing the flow of data on the network. This allows for faster configuration than traditional networks, where the manufacture is expected to do so, as well as the implementation of new applications and services.

This project focuses on the study of OpenFlow, which is explained in detail below.

- **OpenFlow switch:**

OpenFlow is the first standard that has been developed for an SDN. This protocol allows the manipulation of the data plane, by specifying the basic primitives used by the applications. It is used between the network and the SDN controller, to manage the flows that are transmitted in the network, indicating an appropriate action, by means of the



predefined rules. OpenFlow allows you to use remote applications to access the flow tables and thus control traffic from an external terminal.

An OpenFlow controller uses the OpenFlow protocol to connect and configure network devices (routers, switches, etc.) to determine the best route for application traffic. In particular, OpenFlow controllers create a central control point to monitor a variety of network components enabled for OpenFlow. An OpenFlow switch consists of 3 different parts:

- Flow table (flow-table), with an action associated to each entry (flow-entry), in which the node is indicated the way to manage the corresponding flow.
- Communications channel. This channel connects the controller to the switch, through the use of the OpenFlow protocol.
- OpenFlow protocol. Enables communication between the controller and the switch. This prevents programming the switch directly. The driver is responsible for sending the rules, and the nodes store them in their flow-table.

In our project, we must to use an OpenFlow switch that works as NAT. This means that if we send a packet since a private network to a public network, the origin IP must change so, in this way, the other net is not going to know the real IP.

Comparison of controllers:

The core of the OpenFlow network is its controller, described above. The controllers work as a network operating system, have a general view of it and the applications that are executed in the controller are what determine the behaviour of the flows through the network. There are numerous drivers, in different programming languages, such as:

- NOX.
- POX.
- OpenDaylight.
- Floodlight.
- RYU.

We have created a table where we can see some differences between them:

Name of controller:	Programming languages	Description/Objectives	Support for OpenFlow
NOX	C++	Provide a platform that allows developers to make innovations in the management and control of the network. Have a centralized management of connectivity devices, such as switches, a user-level admission control and a security policy engine for the network	OF v1.0

POX	Python	POX, is a controller similar to NOX, but they differ in that POX uses the Python language to program the rules, and NOX uses both Python and C ++	OF v1.0
OpenDaylight	Java	Driver written in Java and that needs a virtual machine, in order to be executed.	OF v1.0
Floodlight	Java	It is a modular open platform for customizing and automating networks of any size and scale. The OpenDaylight Project arose out of the SDN movement, with a clear focus on network programmability. It was designed from the outset as a foundation for commercial solutions that address a variety of use cases in existing network environments.	OF v1.0
RYU	Python	Ryu is a component-based software defined networking framework. Ryu provides software components with well-defined API that make it easy for developers to create new network management and control applications.	OF v1.0, v1.2, v1.3 and Nicira extensions

- **Controller selection:**

We decided to choose Ryu's controller because we thought that it was more complete than the others. Besides, python as the programming language seemed friendlier than, for example, java. The last thing that sold us Ryu was that it supports more versions of OpenFlow than the other options.

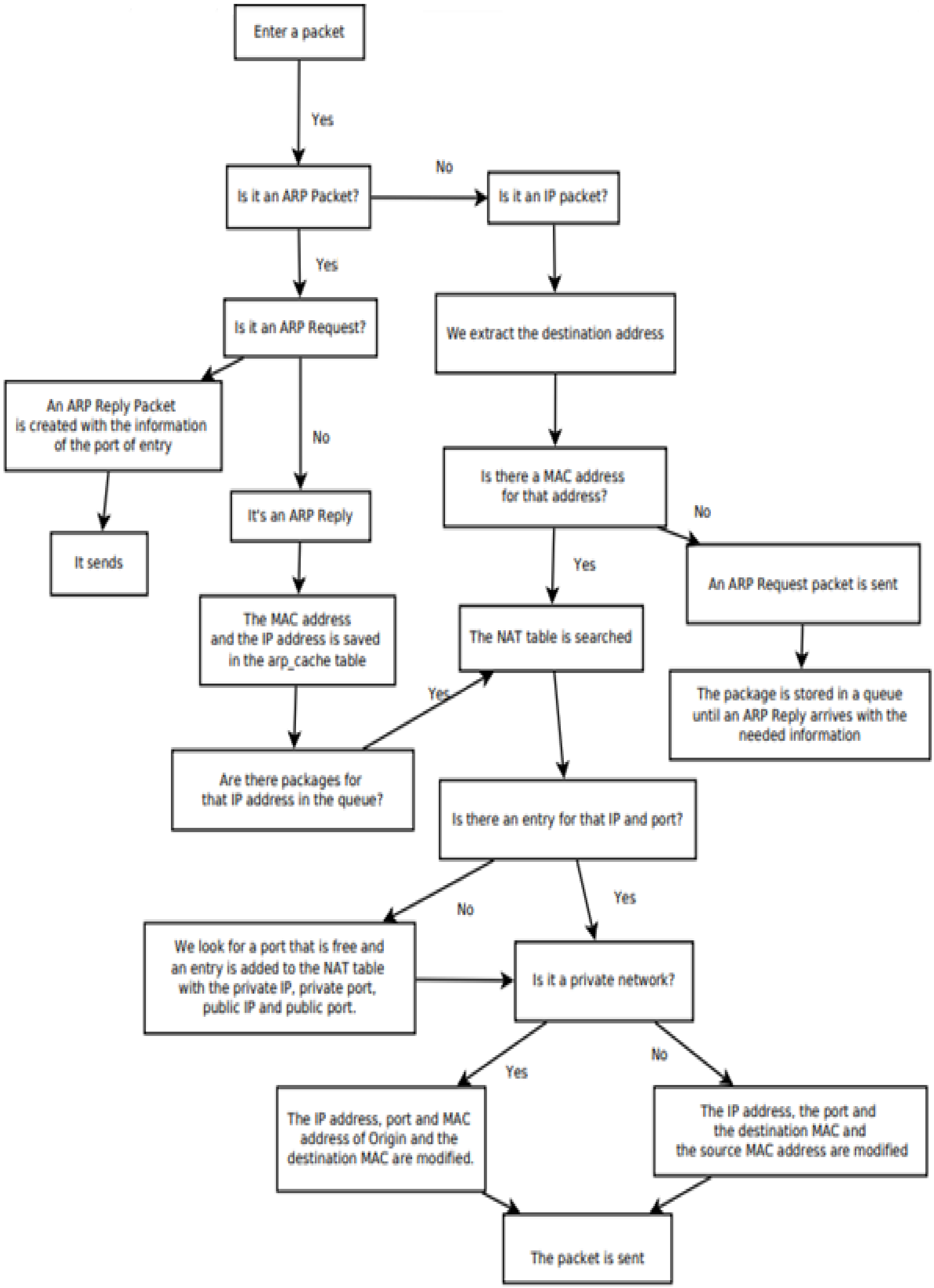
Functional description of the device:

Our device takes the role of a NAT router that translates private addresses into public addresses and vice versa.

There are 2 types of NAT:

- **Source NAT or srcnat:** This type of NAT is performed on packets that are originated from a natted network (private). A NAT router replaces the private source address of an IP packet with a new public IP address as it travels through the router. A reverse operation is applied to the reply packets travelling in the other direction.
- **Destination NAT or dstnat:** This type of NAT is performed on packets that are destined to the natted network. It is most commonly used to make hosts on a private network to be accessible from the Internet. A NAT router performing dstnat replaces the destination IP address of an IP packet as it travels through the router towards a private network.

- **Flowchart:**

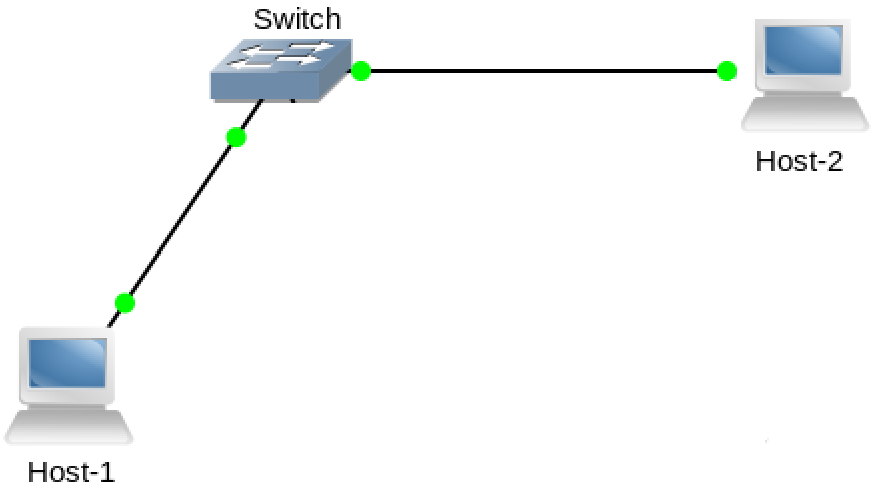


Enter a packet
Yes
Is it an ARP Packet?
No
Is it an IP packet?
Yes
Is it an ARP Request?
We extract the destination address
An ARP Reply Packet is created with the information of the port of entry
No
It sends
It's an ARP Reply
Is there a MAC address for that address?
No
Yes
An ARP Request packet is sent
The MAC address and the IP address is saved in the arp_cache table
The NAT table is searched
The package is stored in a queue until an ARP Reply arrives with the needed information
Yes
Are there packages for that IP address in the queue?
Is there an entry for that IP and port?
Yes
No
We look for a port that is free and an entry is added to the NAT table with the private IP, private port, public IP and public port.
Is it a private network?
No
Yes
The IP address, port and MAC address of Origin and the destination MAC are modified.
The IP address, the port and the destination MAC and the source MAC address are modified
The packet is sent

Implementation:

- **Topology:**

In the picture below we can see the topology that we will be using:



As we can see, we have 2 host. One of them (Host-1) is going to be the external host and the other one is going to be the private/internal host. We do not need more than 1 switch connected to both for this exactly example.

H1 (eth0) ---------- (eth1) S1 (eth2) ---------- (eth0) H2

Remember that the function of NAT is to translate private addresses into public addresses and vice versa.

To actually do this topology we need to create a custom topology file like the one shown below (topo.py)

```
""" Custom topology
host--switch--host
"""
from mininet.topo import Topo
from mininet.net import Mininet
class MyTopo (Topo):
    def __init__(self) :
        Topo.__init__(self)

        h1 = self.addHost('h1', ip="192.168.1.2/24",defaultRoute='via 192.168.1.1')
```

```
        h2 = self.addHost('h2', ip="10.0.0.2/24", defaultRoute='via 10.0.0.69')
        switch = self.addSwitch('s1')

        self.addLink(h1, switch)
        self.addLink(switch, h2)
topos = { 'mytopo': (lambda: MyTopo()) }
```

To use the topology, we will use the next command:

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

As you can see we added some extra parameters to the original command (*sudo mn*).

- *--controller=remote:* With this we tell mn to link himself with the controller (that should be running already in another terminal with *sudo ryu-manager controller.py*).
- *--custom topo.py:* We add the classes or params store in the custom file (we should be in the same directory) topo.py.
- *--topo mytopo*: We select the instance of the topology we wish to use (in this case, *mytopo*).

- **Switch:**

First of all, we needed to develop a completely functional version of ARP from where to start building up our NAT router.

ARP (Addresses Resolution Protocol) is used by the host's and switches to learn destination MAC addresses. It's base in requests and replies.

For example, let's say h1 wants to ping h2:

- First h1 would send an ARP request to s1 using eth0 (via gateway).
- Next, s1 would receive the request and then it would reply h1 with the MAC of the interface eth1. After receiving the reply, h1 would learn the address and proceed to send s1 the ICMP packets.
- S1 would start now receiving the ICMP packets from h1 (now h1 knows s1's MAC) sent via gateway. This packet must be send to h2 using the s1 interface eth2, but s1 doesn't know the MAC address of the h2 interface eth0. S1 needs to send an ARP request to h2 to learn the MAC.
- Once h2 receive the request from s1 it sends the MAC with an ARP reply.
- When that reply reaches s1 will learn it and then proceed to send the packets from h1 through him directly to h2.

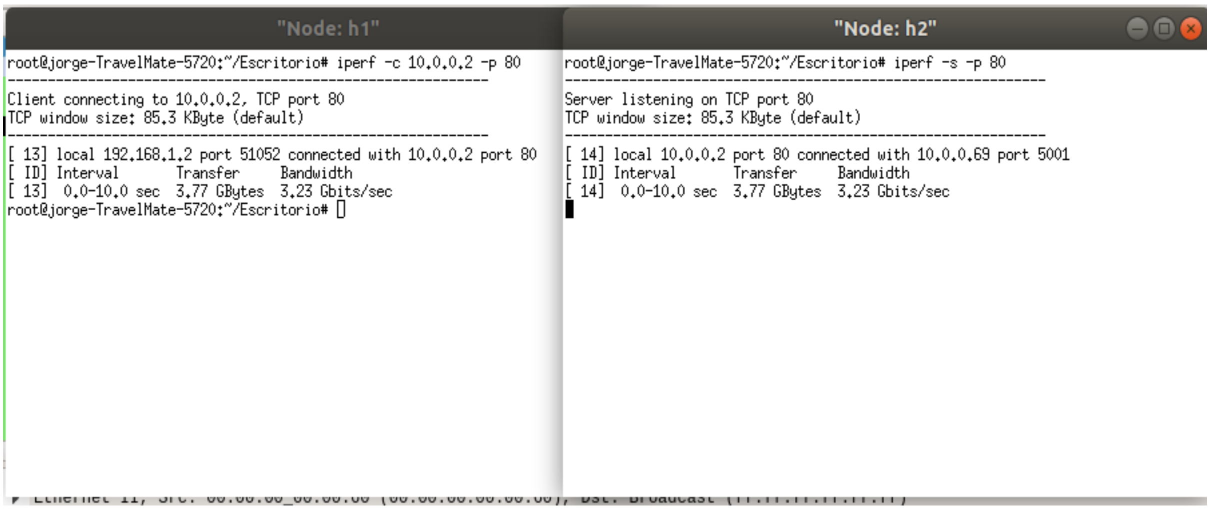
Once we were done with ARP we started to develop our NAT.

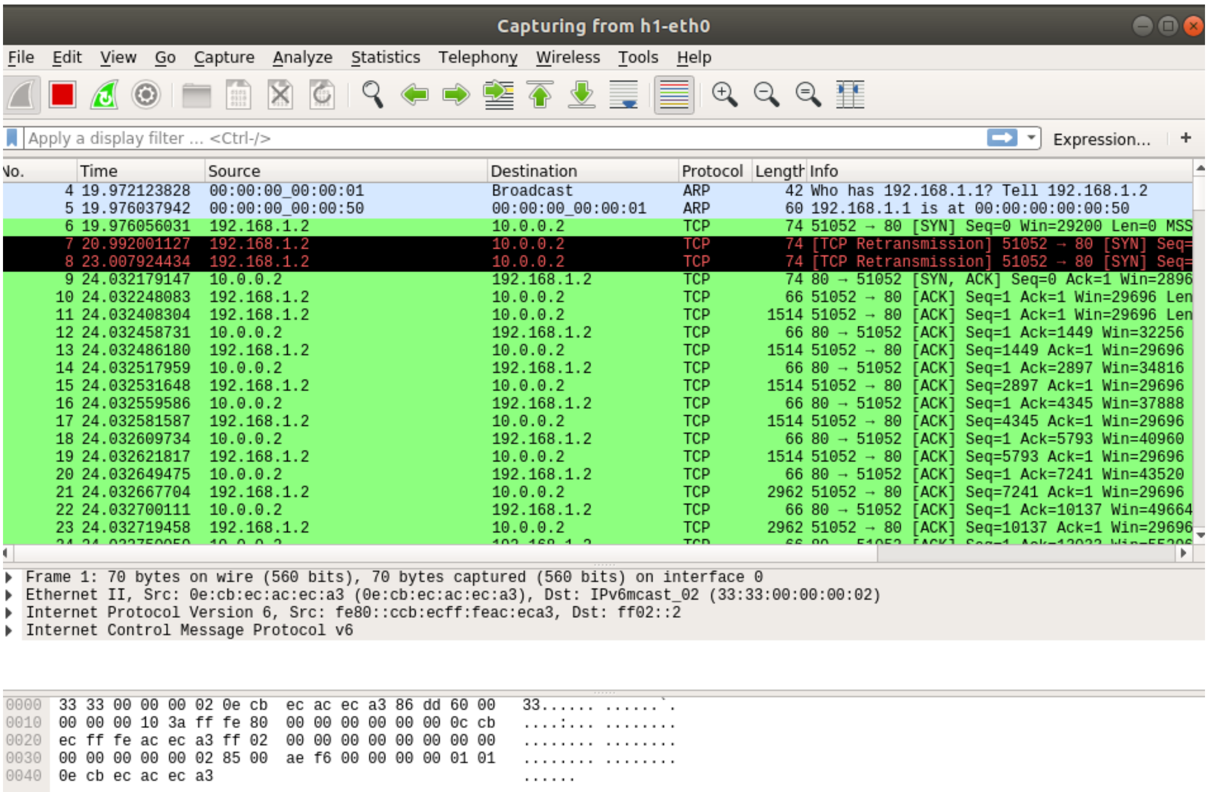
To keep it simple our NAT does the translation like this:

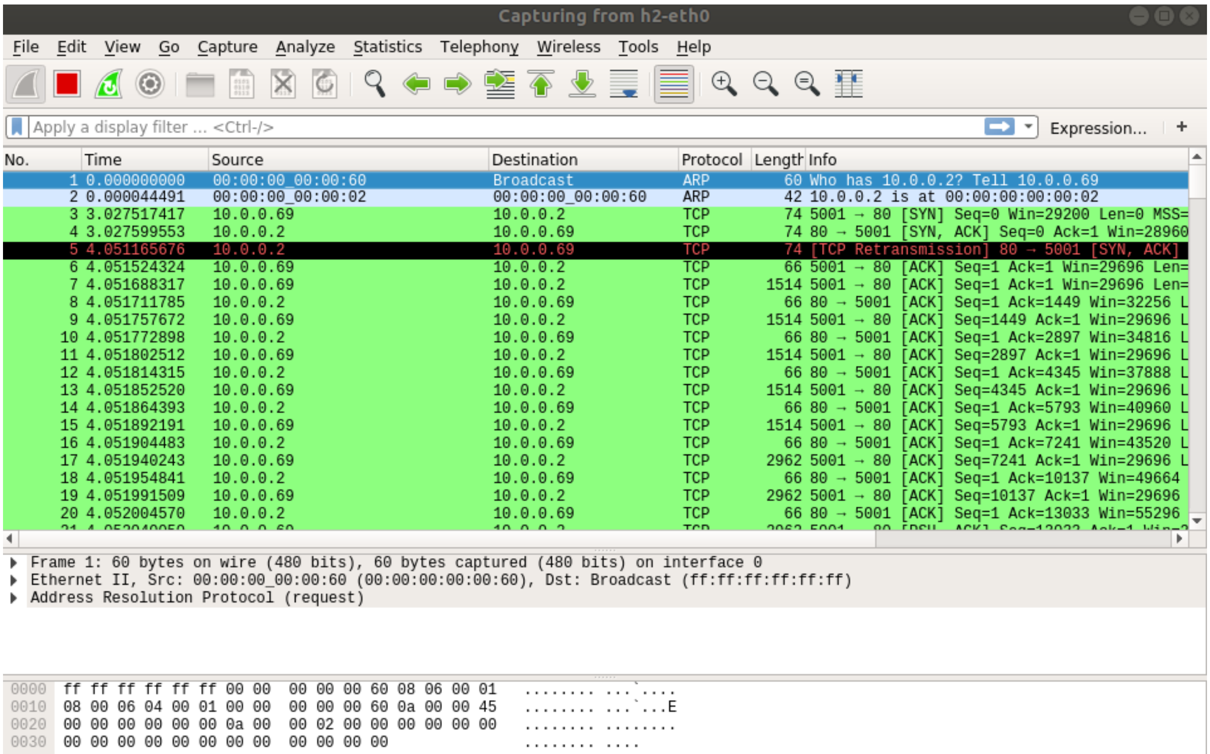
1. Takes incoming forward packets from the internal network and mask the srcIP and srcMAC of h2 into the srcIP and srcMAC of the external interface of s1 (eth1).
2. Take incoming forward external packets and translates the dstIP (that it will always be the IP of the external interface of s1) into the correct dstIP (the h2 eth0 IP) using the in ports to select it.

Testing in Mininet and Wireshark:

We can see in the images of below how it works:




Capturing from h1-eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
No. Time Source Destination Protocol Length Info
4 19.972123828 00:00:00_00:00:01 Broadcast ARP 42 Who has 192.168.1.1? Tell 192.168.1.2
5 19.976037942 00:00:00_00:00:50 00:00:00_00:00:01 ARP 60 192.168.1.1 is at 00:00:00:00:00:50
6 19.976056031 192.168.1.2 10.0.0.2 TCP 74 51052 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS
7 20.992001127 192.168.1.2 10.0.0.2 TCP 74 [TCP Retransmission] 51052 → 80 [SYN] Seq=
8 23.007924434 192.168.1.2 10.0.0.2 TCP 74 [TCP Retransmission] 51052 → 80 [SYN] Seq=
9 24.032179147 10.0.0.2 192.168.1.2 TCP 74 80 → 51052 [SYN, ACK] Seq=0 Ack=1 Win=2896
10 24.032248083 192.168.1.2 10.0.0.2 TCP 66 51052 → 80 [ACK] Seq=1 Ack=1 Win=29696 Len
11 24.032408304 192.168.1.2 10.0.0.2 TCP 1514 51052 → 80 [ACK] Seq=1 Ack=1 Win=29696 Len
12 24.032458731 10.0.0.2 192.168.1.2 TCP 66 80 → 51052 [ACK] Seq=1 Ack=1449 Win=32256
13 24.032486180 192.168.1.2 10.0.0.2 TCP 1514 51052 → 80 [ACK] Seq=1449 Ack=1 Win=29696
14 24.032517959 10.0.0.2 192.168.1.2 TCP 66 80 → 51052 [ACK] Seq=1 Ack=2897 Win=34816
15 24.032531648 192.168.1.2 10.0.0.2 TCP 1514 51052 → 80 [ACK] Seq=2897 Ack=1 Win=29696
16 24.032559586 10.0.0.2 192.168.1.2 TCP 66 80 → 51052 [ACK] Seq=1 Ack=4345 Win=37888
17 24.032581587 192.168.1.2 10.0.0.2 TCP 1514 51052 → 80 [ACK] Seq=4345 Ack=1 Win=29696
18 24.032609734 10.0.0.2 192.168.1.2 TCP 66 80 → 51052 [ACK] Seq=1 Ack=5793 Win=40960
19 24.032621817 192.168.1.2 10.0.0.2 TCP 1514 51052 → 80 [ACK] Seq=5793 Ack=1 Win=29696
20 24.032649475 10.0.0.2 192.168.1.2 TCP 66 80 → 51052 [ACK] Seq=1 Ack=7241 Win=43520
21 24.032667704 192.168.1.2 10.0.0.2 TCP 2962 51052 → 80 [ACK] Seq=7241 Ack=1 Win=29696
22 24.032700111 10.0.0.2 192.168.1.2 TCP 66 80 → 51052 [ACK] Seq=1 Ack=10137 Win=49664
23 24.032719458 192.168.1.2 10.0.0.2 TCP 2962 51052 → 80 [ACK] Seq=10137 Ack=1 Win=29696
Frame 1: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
Ethernet II, Src: 0e:cb:ec:ac:ec:a3 (0e:cb:ec:ac:ec:a3), Dst: IPv6mcast_02 (33:33:00:00:00:02)
Internet Protocol Version 6, Src: fe80::ccb:ecff:feac:eca3, Dst: ff02::2
Internet Control Message Protocol v6
0000 33 33 00 00 00 02 0e cb ec ac ec a3 86 dd 60 00 33...... ......`.
0010 00 00 00 10 3a ff fe 80 00 00 00 00 00 00 0c cb ....:... ....
0020 ec ff fe ac ec a3 ff 02 00 00 00 00 00 00 00 00 ....
0030 00 00 00 00 00 02 85 00 ae f6 00 00 00 00 01 01 ....
0040 0e cb ec ac ec a3 ......


Capturing from h2-eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...
No. Time Source Destination Protocol Length Info
1 0.000000000 00:00:00_00:00:60 Broadcast ARP 60 Who has 10.0.0.2? Tell 10.0.0.69
2 0.000044491 00:00:00_00:00:02 00:00:00_00:00:60 ARP 42 10.0.0.2 is at 00:00:00:00:00:02
3 3.027517417 10.0.0.69 10.0.0.2 TCP 74 5001 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=
4 3.027599553 10.0.0.2 10.0.0.69 TCP 74 80 → 5001 [SYN, ACK] Seq=0 Ack=1 Win=28960
5 4.051165676 10.0.0.2 10.0.0.69 TCP 74 [TCP Retransmission] 80 → 5001 [SYN, ACK]
6 4.051524324 10.0.0.69 10.0.0.2 TCP 66 5001 → 80 [ACK] Seq=1 Ack=1 Win=29696 Len=
7 4.051688317 10.0.0.69 10.0.0.2 TCP 1514 5001 → 80 [ACK] Seq=1 Ack=1 Win=29696 Len=
8 4.051711785 10.0.0.2 10.0.0.69 TCP 66 80 → 5001 [ACK] Seq=1 Ack=1449 Win=32256 L
9 4.051757672 10.0.0.69 10.0.0.2 TCP 1514 5001 → 80 [ACK] Seq=1449 Ack=1 Win=29696 L
10 4.051772898 10.0.0.2 10.0.0.69 TCP 66 80 → 5001 [ACK] Seq=1 Ack=2897 Win=34816 L
11 4.051802512 10.0.0.69 10.0.0.2 TCP 1514 5001 → 80 [ACK] Seq=2897 Ack=1 Win=29696 L
12 4.051814315 10.0.0.2 10.0.0.69 TCP 66 80 → 5001 [ACK] Seq=1 Ack=4345 Win=37888 L
13 4.051852520 10.0.0.69 10.0.0.2 TCP 1514 5001 → 80 [ACK] Seq=4345 Ack=1 Win=29696 L
14 4.051864393 10.0.0.2 10.0.0.69 TCP 66 80 → 5001 [ACK] Seq=1 Ack=5793 Win=40960 L
15 4.051892191 10.0.0.69 10.0.0.2 TCP 1514 5001 → 80 [ACK] Seq=5793 Ack=1 Win=29696 L
16 4.051904483 10.0.0.2 10.0.0.69 TCP 66 80 → 5001 [ACK] Seq=1 Ack=7241 Win=43520 L
17 4.051940243 10.0.0.69 10.0.0.2 TCP 2962 5001 → 80 [ACK] Seq=7241 Ack=1 Win=29696 L
18 4.051954841 10.0.0.2 10.0.0.69 TCP 66 80 → 5001 [ACK] Seq=1 Ack=10137 Win=49664
19 4.051991509 10.0.0.69 10.0.0.2 TCP 2962 5001 → 80 [ACK] Seq=10137 Ack=1 Win=29696
20 4.052004570 10.0.0.2 10.0.0.69 TCP 66 80 → 5001 [ACK] Seq=1 Ack=13033 Win=55296
Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:60 (00:00:00:00:00:60), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)
0000 ff ff ff ff ff ff 00 00 00 00 00 60 08 06 00 01 ....`....
0010 08 00 06 04 00 01 00 00 00 00 00 60 0a 00 00 45 ....`...E
0020 00 00 00 00 00 00 0a 00 00 02 00 00 00 00 00 00 ....
0030 00 00 00 00 00 00 00 00 00 00 00 00 ....

Annexed:

- **Grantt diagram:**

We know that a Grantt diagram is not exactly like we did but there is a why. This is because the most part of tasks are things that we do all time, so we think that it is going to be easier look at the table that we did. Time is the numbers of hours that we dedicated without count the hours that we invert in the classroom. Also, with respect to the code, we started with three versions and we chose the one that went ahead and worked better, that is why we have many hours of implementation each one of us. Further, there are so many hours searching information because it was new for all of us, so we had to search about how Python and Mininet work, and then how can we do the code using RYU for NAT. We have a lot of hours testing too because this project was a perfect example of "try and failure" so we had to prove when we change a line of code.

Hours that one of each dedicate:

TASKS	Start day	Sergio	Kathrina	Jorge	Edgar
Donwloads and installs	18 OF APRIL	2	3	3	3
Searching information	18 OF APRIL	8	8	9	9
Programming	19 OF APRIL	16	14	18	13
Testing	19 OF APRIL	7	6	7	7
Report	26 OF APRIL	3	5	2	2
Presentation	13 OF MAY	1	1	1	1