UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

SDN-sniff: monitor multi-tenant traffic in virtualized network infrastructure

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Luis Fernández Sánchez

In partial fulfilment

of the requirements for the degree in

***TELECOMMUNICATON SYSTEMS* ENGINEERING**

Director: Franco Callegati

Advisor: Josep Solé Pareta

Bologna, Italy, March 2018

Abstract

The Software Defined Networking (SDN) is an emerging paradigm in network design and management that enables the optimization of the use of network resources. This paradigm is being explored and developed from several different angles.

The scope of interest of this thesis is the security applications on SDN, specifically the possibility of the packet interception in this kind of networks. A proof of concept was developed in a previous work and the goal of this thesis is to contribute by developing a more advanced testing environment in which run deeper tests to understand the underlying behaviors and message flow during the monitoring.

Resum

Les Xarxes Definides per Software (SDN, per les seves sigles en angles) és un paradigma emergent en el disseny i la gestió de xarxes que permet la optimització en l'ús dels recursos de les mateixes. Aquest paradigma està sent explorat i desenvolupat des de diferents orientacions.

L'àmbit d'interès d'aquest treball de fi de grau són les aplicacions de seguretat en SDN, específicament la possibilitat d'intercepció de paquets en aquest tipus de xarxes. En un treball previ es va desenvolupar una prova de concepte i l'objectiu d'aquest treball es el de contribuir desenvolupant un entorn de proves més avançat en el que poder realitzar proves més completes per entendre els comportaments subjacents i els fluxos de missatges durant el monitoratge de la xarxa.

Resumen

Las Redes Definidas por Software (SDN, por sus siglas en inglés) es un paradigma emergente en el diseño i la gestión de las redes que permite la optimización en el uso de los recursos de las mismas. Este paradigma está siendo explorado y desarrollado en sus diferentes vertientes.

El ámbito de interés de este trabajo de fin de grado son las aplicaciones en seguridad en SDN, específicamente la posibilidad de intercepción de paquetes en este tipo de redes. En un trabajo previo se ha desarrollado una prueba de concepto y el objetivo de este trabajo es el de contribuir desarrollando un entorno de pruebas más avanzado en el que poder realizar pruebas más completas para entender los comportamientos subyacentes y los flujos de mensajes durante la monitorización de la red.

Revision history and approval record

Revision	Date	Purpose
0	27/02/2018	Document creation
1	16/03/2018	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Luis Fernández Sánchez	luis.fernandez.sanchez@estudiant.upc.edu
Franco Callegati	franco.callegati@unibo.it
Josep Solè Pareta	pareta@ac.upc.edu
Walter Cerroni	walter.cerroni@unibo.it

Written by:		Reviewed and approved by:	
Date	27/02/2018	Date	16/03/2018
Name	Luis Fernández	Name	Franco Callegati
Position	Project Author	Position	Project Supervisor

Table of contents

The table of contents must be detailed. Each chapter and main section in the thesis must be listed in the “Table of Contents” and each must be given a page number for the location of a particular text.

List of Figures

List of Tables:

1. Introduction

1.1. Statement of purpose

The present thesis has the objective of contributing to the investigations in the fields of security and user monitoring in virtualized networking infrastructures. For doing so a new testing environment will be implemented in order to help to test and improve the software developed previously by Giulio Crestani in his Master Thesis "*Implementazione di intercetazione telematica in reti virtuali definite via software".* In order to set the path towards this objective, we will be studying different networking technologies as well as various network management tools.

Software Defined Networking (SDN) is a relatively new paradigm for network management that is based on the principle of detaching the packet process (Data plane) from the routing process (Control Plane). As the use of SDNs expands, new needs are arising and so are doing new opportunities to fulfil them. The creation of new tools to cover necessities in the field of security and monitoring was the original purpose of this investigation line. In his original work, Crestani developed a sniffer software that use the properties of SDN in order to intercept the packages between two hosts without the knowledge of any of them. Due to the time constraint, he was just able to test it in the limited conditions of a virtual setting in just one physical machine.

The purpose of the present thesis is to **contribute to the investigations in security and user monitoring in virtualized networking infrastructures.**

The main objectives are:

- Understand the Software Defined Networking technology
- Get acquainted with network virtualization tools like Mininet and with the OpenFlow protocol
- Understand network slicing techniques and implement an experimental network slicing scenario using a slicing tool like OpenVirtex
- Investigate architectural choices to guarantee full virtual network infrastructure control for security and monitoring reasons
- Implement a distributed proof of concept with in depth analysis of the protocol behaviors and message flow during the traffic monitoring

1.2. Requirements and specifications

Project requirements:

- Understand and experiment the SDN main concepts, using the Mininet platform and the Mininet built in controllers

- Learn how to connect external controllers to the Mininet platform and how to analyze controller/switches traffic
- Deploy a SDN network virtualization platform
- Experiment the platform functionalities
- Verify the message flows and the various building blocks interactions
- Possibly expand the platform to accept external commands to implement networking conditions that may be used to monitor the user traffic

1.3. Methodology and Previous Works

The idea of this project originates on a previous Master thesis (*“Implementazione di intercettazione telematica in reti virtuali definite via software*” by Giulio Crestani) about the development of a network sniffer for SDNs. The main goal is to understand the technology of SDN in order to be able to implement a new benchmarking/testing environment.

In order to do so, the main tools used will be Mininet, OpenVirteX (OVX), Python as the main programming language for the scripts in them, Floodlight as the SDN controller and Wireshark as the main network analysis tool.

Using this tools, an environment test will be developed in order to help achieve deeper understanding of the performance of the sniffer as well as to contribute to its future growth.

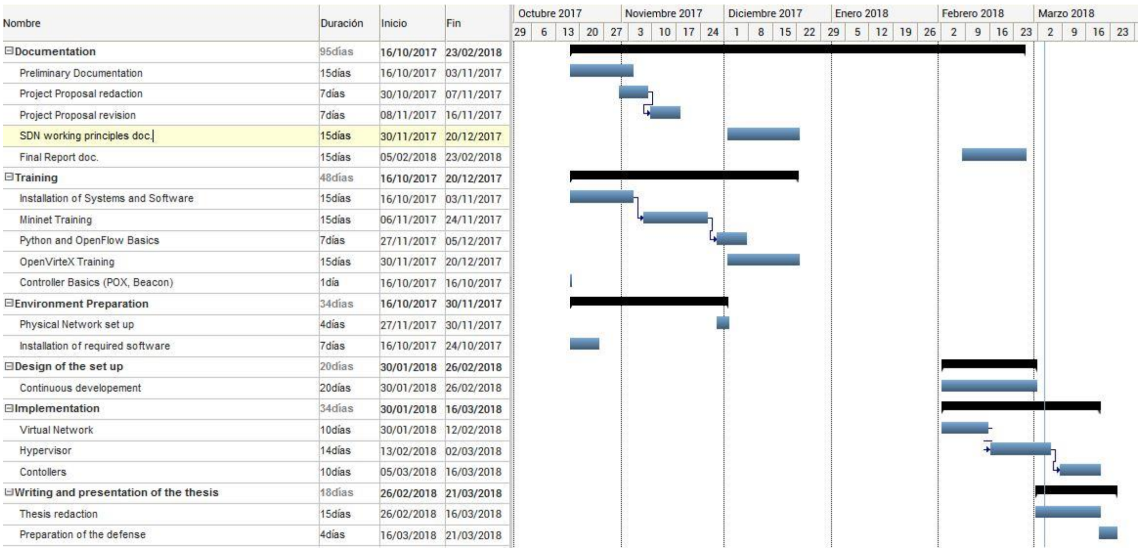
1.4. Work Plan

The project work plan has been largely modified due to the initial optimism in which it was defined. The final work structure and time distribution between the different Work Packages (WP) is shown in the following sections.

Work plan packages

- WP 1: Documentation
- WP 2: Training
- WP 3: Environment preparation
- WP 4: Design of the set up
- WP 5: Implementation
- WP 6: Writing and presentation of the thesis

Gantt Diagram



Nombre	Duración	Inicio	Fin
⊟**Documentation**	95días	**16/10/2017**	**23/02/2018**
Preliminary Documentation	15días	16/10/2017	03/11/2017
Project Proposal redaction	7días	30/10/2017	07/11/2017
Project Proposal revision	7días	08/11/2017	16/11/2017
SDN working principles doc.	15días	30/11/2017	20/12/2017
Final Report doc.	15días	05/02/2018	23/02/2018
⊟**Training**	48días	**16/10/2017**	**20/12/2017**
Installation of Systems and Software	15días	16/10/2017	03/11/2017
Mininet Training	15días	06/11/2017	24/11/2017
Python and OpenFlow Basics	7días	27/11/2017	05/12/2017
OpenVirteX Training	15días	30/11/2017	20/12/2017
Controller Basics (POX, Beacon)	1día	16/10/2017	16/10/2017
⊟**Environment Preparation**	34días	**16/10/2017**	**30/11/2017**
Physical Network set up	4días	27/11/2017	30/11/2017
Installation of required software	7días	16/10/2017	24/10/2017
⊟**Design of the set up**	20días	**30/01/2018**	**26/02/2018**
Continuous developement	20días	30/01/2018	26/02/2018
⊟**Implementation**	34días	**30/01/2018**	**16/03/2018**
Virtual Network	10días	30/01/2018	12/02/2018
Hypervisor	14días	13/02/2018	02/03/2018
Contollers	10días	05/03/2018	16/03/2018
⊟**Writing and presentation of the thesis**	18días	**26/02/2018**	**21/03/2018**
Thesis redaction	15días	26/02/2018	16/03/2018
Preparation of the defense	4días	16/03/2018	21/03/2018

Figure 1. Gantt Diagram of the Degree Thesis

The current thesis had to be adapted and suffered several changes from the initial proposal due to the excess of optimism of the author, being the main topics outside of his field of expertise. The original approach was to develop a new test environment, be able to study and understand Crestani's software, design some new and deeper tests, run them and get to improve the software. Obviously this ended being not feasible without having any previous knowledge on virtual networking and SDN. That is why the project has been redefined to something coherent with a Bachelor Thesis.

2. State of the art of the technology used or applied in this thesis:

2.1. Network Simulation

Software Defined Networking investigation often requires experimentation but usually we do not get to have a physical infrastructure required. That is the point where simulated networks get relevant by allowing experimentation at minimum cost (usually no cost at all) [1]. Even if there are some flaws with simulated networks, they are a valuable resource as they can be fully customized, setting the topology, traffic patterns and dynamic events. This can be done in a scalable way, as shown in [2]. By doing so we can visualize and monitor all the traffic and interactions in our system and even test its resilience to failure.

2.2. Software Defined Networking (SDN)

Software Defined Networks, also known as SDNs, are a new paradigm in network management in which the main concern is to extract as much logic as possible from the hardware. The main goal is to be able to centralize all the “intelligence” of the network in a single controller instead of having it divided between all the switches and/or routers.

This segregation between hardware (data plane) and software (control plane) is meant to increase the efficiency by reducing process times in the intermediate nodes using different flow protocols, usually OpenFlow.

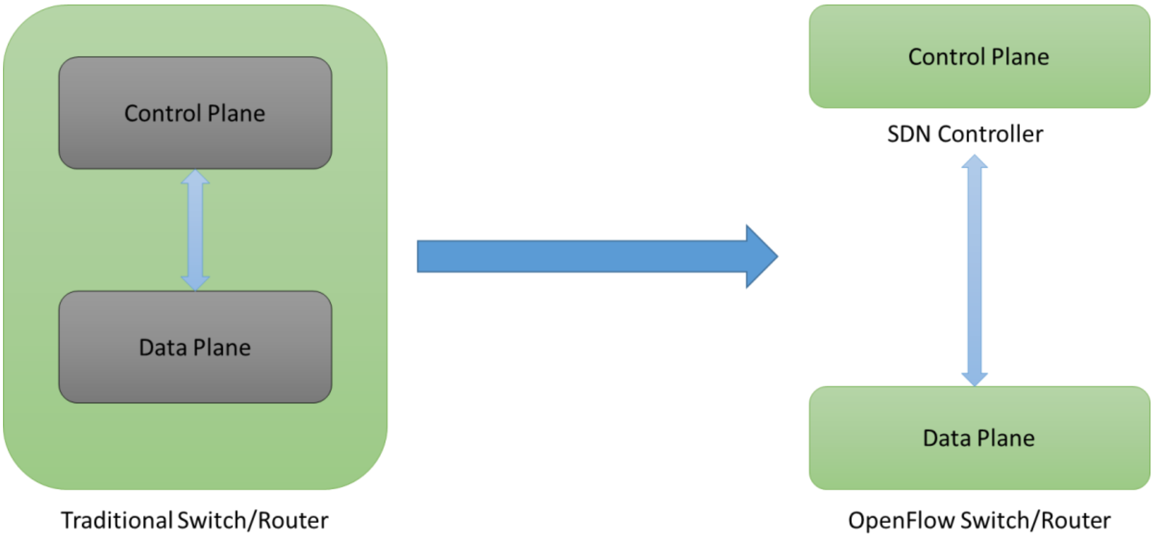


Figure 2. From traditional network to SDN

The way this objective is achieved is by “programing” in the networks some behaviors. By doing so, the nodes just have to look for a few characteristics in the packets and then compare it with a flow table pushed down by the controller to decide what action to carry out.

2.2.1. SDN Architecture

On a traditional network there are a series of computing elements connected that decide individually what to do with a packet they have received based on the resources they have, the kind of packet received and the information contained on it.

On the other hand, on SDN [3], we have an extra control layer above the physical infrastructure. This layer define some simple rules for the switches, centralizing the network intelligence on the SDN Controller.

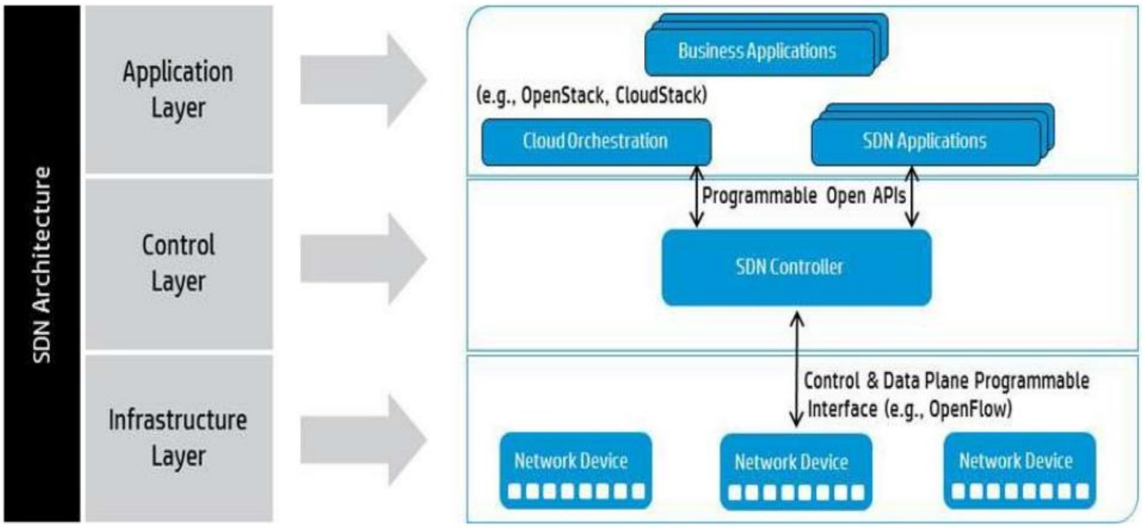


Figure 3. Example of SDN Architecture. Source: SDN 101: Software Defined Networking Course - Sameh Zaghloul/IBM – 2014[1]

As shown in the figure above, the OpenFlow switches communicate with the Control Plane to provide statistics and performing information as well as for receiving instructions from the controller. To do so, it uses the SDN Control-Plane Interface (CDPI), which is the interface between SDN Controller, and SDN Datapath, a logical network device that exposes visibility and uncontested control over its advertised forwarding and data processing capabilities. Finally, the SDN Northbound Interface (NBI) allows the controller to interact with SDN applications that run on top of it by providing abstract network views (instead of showing the direct network behavior).

One of the main advantages of this architecture is that the applications using the network do not need to know the topology nor the hardware in order to run over it, in addition to the others commented above, like networking performance, reliability or resilience.

---

[1] https://www.slideshare.net/SamehZaghloul/iti-sdn-101-v11

2.2.2. Security in SDN

The concern for the security matters in SDNs Is not a new thing and several teams had developed different investigations in the mentioned field. There have been different approaches to the matter. In [4] the authors developed and experimental framework by abstracting the security mechanisms from the hardware layer to the software layer.

The work of [5] builds a layer 2 fire-wall over a simulated SDN in order to provide this protection tool for SDNs. [6] also focus on blocking and protecting from external attacks, specifically DDos botnet-based attacks. Finally in [7] we can observe a more general approach to security using the unique set of properties of the SDNs.

As observable, the main objective of the different works is mainly oriented towards increasing the security of the systems based on SDNs or use some of their properties to create new security architectures. Meanwhile the use of those properties in order to create new tools to intercept information is not being exploited.

2.3. OpenFlow

Born in 2008 in Stanford University, OpenFlow is a communication protocol that enables the administration of a network from a centralized controller. It is the main protocol used in the communication between the Control Plane and the Data Plane. It allows to define the paths that the packets inside the network will follow using simple rules. Those rules are actions that have to be taken over a package if certain conditions are met. Typical conditions can be matching a specific entry port, source or destiny MAC address, protocol, etc.

This rules, called flows, are pushed by the controller into every single switch. Every time a switch gets a packet that does not fit into its flows, it send it to the controller to know what to do about it and the controller gives a new flow as the reply.

It is, essentially, the protocol that defines the behavior of SDN and it is considered by many as one of the firsts SDN standards

3. Methodology / project development:

This chapter describes the methodology used in the development of the project and the processes involved.

3.1. Virtual Machines

Being most of the project outside of the field of specialization of the author, it is a logical conclusion that the training and study must take an important portion of the effort invested in the project. In order to invest these efforts and time in the most efficient way, it is usual that software developers provide tutorial systems encapsulated in customized Virtual Machines (VM).

The possibility of running simulated machines inside of real physical ones is not new, but it still quite useful. By creating a VM image with the OS and software that is required for a specific task, like training into SDN, we can simplify the access to knowledge and increase the efficiency of some processes that have to be replicated in several different machines.

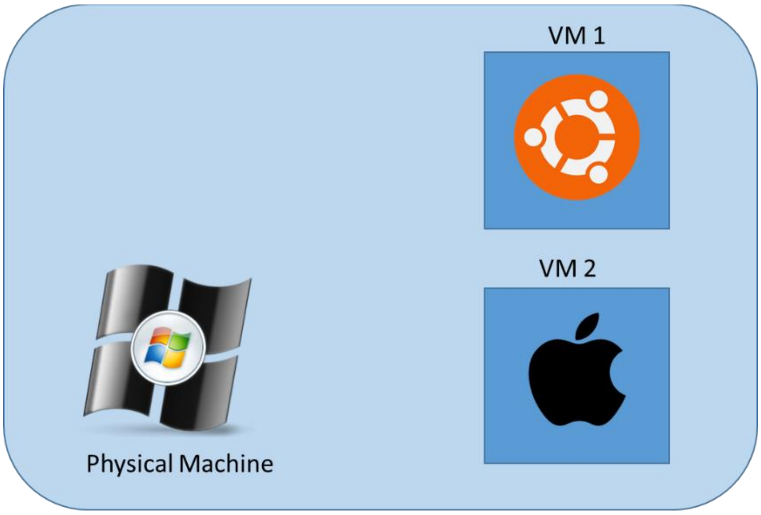


Figure 4. We can run VMs with different OS inside one single physical machine

In the context of this project, the VM have a relevant role in the training part, as they are main tools to learn provided by the developers about both Mininet and OpenVirteX. The chosen tool to run them in the project has been VirtualBox.

VirtualBox is a widely extended platform available for all main OS and used to run VMs via GUI or via command line. Nevertheless, working on a VM inside a remote machine via SSH has proven not be the most reliable nor comfortable way.

3.2. Mininet

To develop a reliable testing environment we needed a stable network simulation platform. A prominent tool in the world of network simulation is Mininet.

Created by Bob Lantz and Brandon Heller, based on the prototype by Bob Lantz[2], Mininet was intended to be a simulation and learning tool for networking, evolving to become one of the main options for SDN and OpenFlow simulations and experiments [8] [9] [10]. It works as an emulation of a network build with Linux based hosts and switches that support OpenFlow in which it is possible to create networks with custom complex topologies allowing to experiment in an inexpensive, fast and scalable way.

As seen in the State of the Art, SDN architecture is built over a physical network used as the infrastructure. In the current project, we have considered more convenient to simulate this physical network using Mininet. Its goal is to provide the basic infrastructure over which the virtualization/emulation layer will be established. For doing so we define a topology that matches the necessities of the project without adding excessive complexity. Those topologies can be defined using different methods, for our set up we have chosen defining it by using the Mininet libraries for Python.

Getting started with Mininet (in Ubuntu) is as simple as installing via command line from the repository and execute *$sudo mn* in a terminal. You can add commands with instructions such as the number of switches, hosts, the kind of topology, etc. but at the end, it come easier to code it into a Python script.

3.3. OpenVirteX and virtual networking

Network virtualization is the process of combine both physical and virtual network elements in order to work as a single entity. These virtual networks (VNs) are useful in many scenarios like training or experimentation of new features reducing the entrance barrier as almost no physical infrastructure is needed. With the evolution of this concept network slicing appeared, allowing to different virtual operators use the same resources independently. [11]

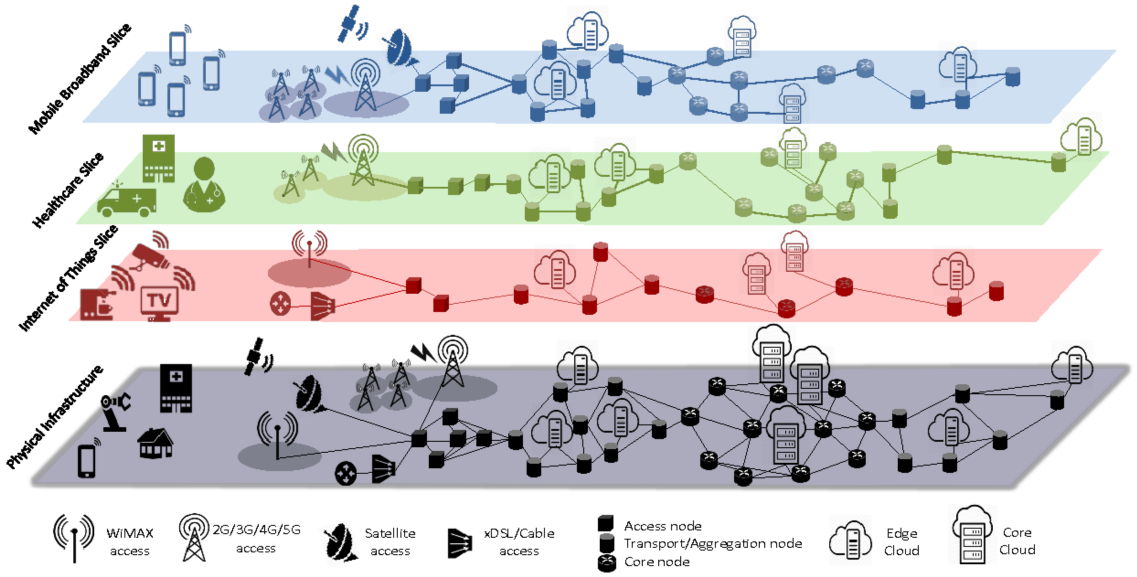


Figure 5. Example of virtual slices over a physical network. SOURCE: Network Slicing for 5G with SDN/NFV: Concepts, Architectures, and Challenges[3]

---

[2] http://mininet.org/credits/

[3] Network Slicing for 5G with SDN/NFV: Concepts, Architectures, and Challenges [9]

OpenVirteX was born in order to exploit the advantages of virtual networking as well as the possibilities that network slicing offered. The platform was conceived as an *Hypervisor,* an intermediate software layer that is able to manage several different virtual systems (even with different OS) in order to allow them to share the same physical resources. This translates to several independent virtual networks on top of it while a single physical network bellow.

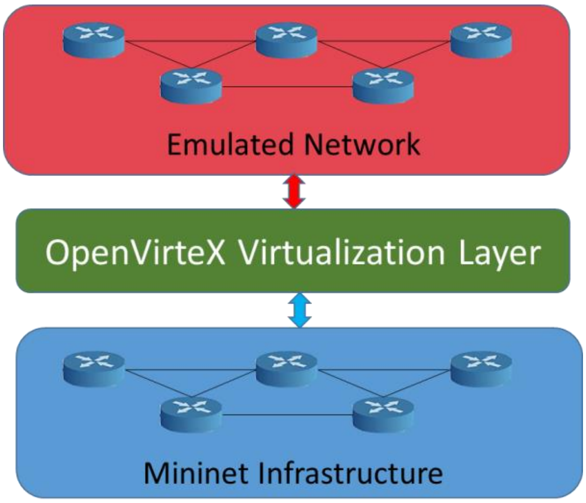


Figure 6. The emulated network created with OVX will be a clone of the one deployed by Mininet

In the context of the project, OpenVirteX is going to be used for approximate the real network conditions, but its features of slicing and network virtualization will not be used. Instead one single emulated network will be created for each Mininet topology by cloning it. This will be achieved by coding bash scripts that use the *ovxclt.py* controller script and the commands provided for it by the ONF. The specific scripts used in each case are in the corresponding annex, in chapter 6.

3.4. Floodlight and Network Controllers

The key element of every SDN architecture is the controller. Is the component in charge of the correct functioning of the network. It defines the rules on which the network is based, the main and back up routes of the packages, and enforces it by pushing flow tables into the networking elements. The controller also collects live information about the status, traffic and performance of the network in order to make it always as efficient as possible and to change the paths in case of link saturation or drop.

For the project set up, we have chosen Floodlight controller for being the one recommended by OpenVirteX developers and being already used in the previous project.

Floodlight is an open source, enterprise-class, Apache-licensed, Java-based OpenFlow controller developed and supported by a community and sponsored by Big Switch Networks Inc.

Its role in the project will be just about acting as a controller but we will not be building anything on top of it, as that would exceed the purpose of this thesis.

We will not have to run any specific configuration, just installing it in the bigTower and change the version of the OpenFlow to 1.0 so it matches the only one supported currently by OpenVirteX.

3.5. Global Set Up

Once we are confident and fluent in the uses of the tools that compose the different layers of the set up, we get to set the whole environment. Let's recap, the elements involved are:

- Physical components:
    - miniTower68 (Physical Machine)
    - bigTower (Physical Machine)
- Software components:
    - Mininet simulated network, hosted on miniTower68
    - OpenVirteX virtualization layer, hosted on miniTower68
    - Floodlight SDN controller, hosted on bigTower

As pointed above, all the systems will be deployed in physical machines, in order to allow achieve one of the main objectives of the project: have separated physical interfaces that can be easily captured by the packet analyzer.

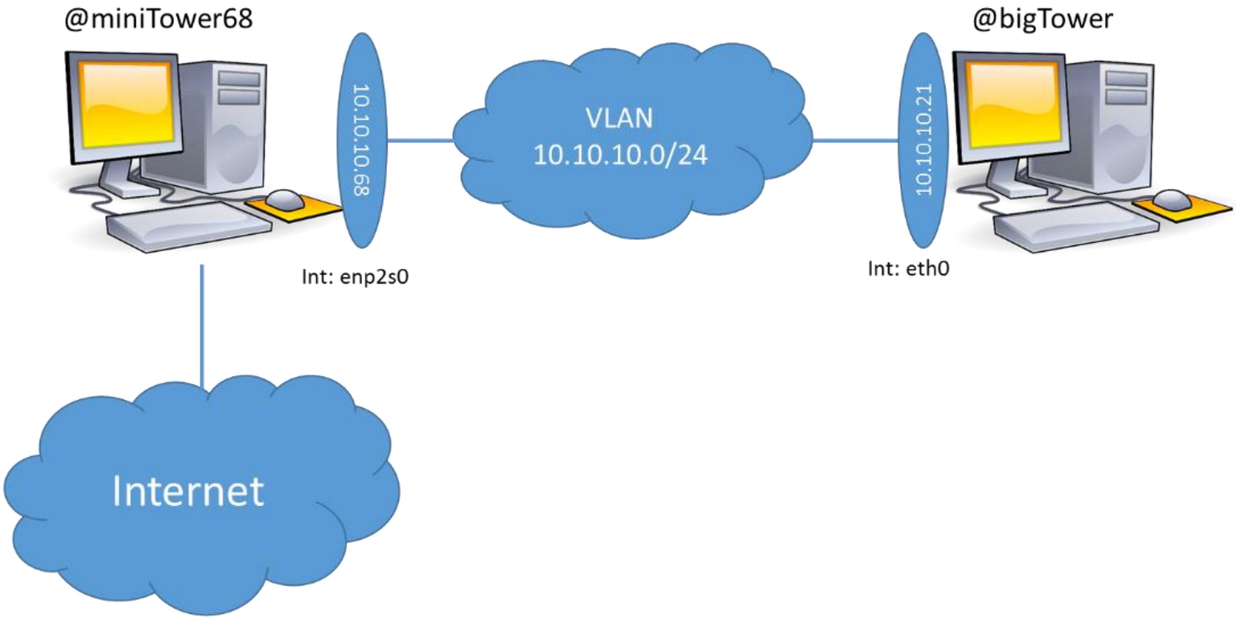


Figure 7. Physical machine configuration

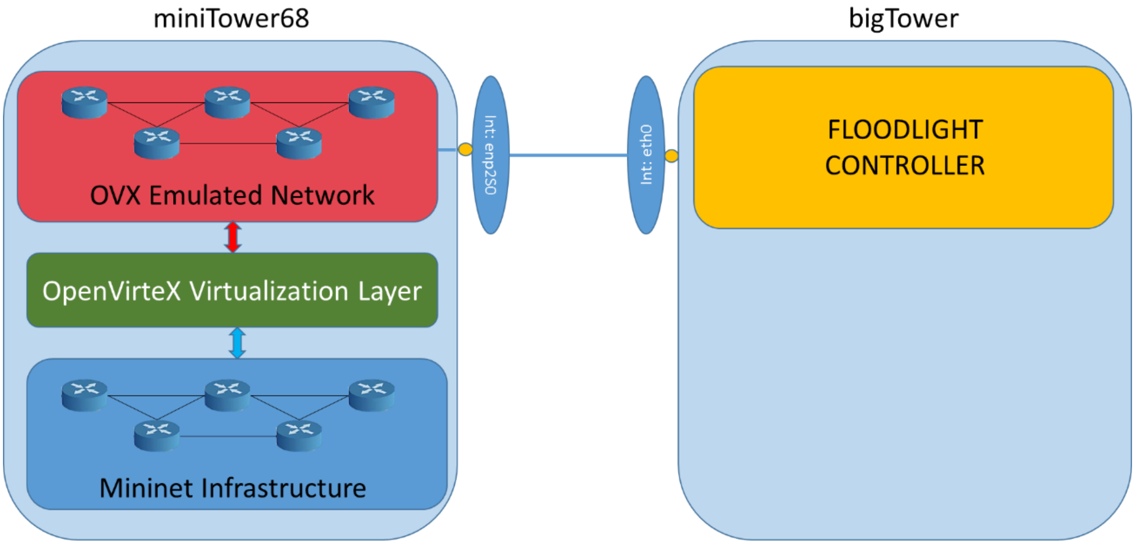


Figure 8. Global test environment set up

3.6. Wireshark and testing phase

When finally the test environment was fully built, we ran some tests (see section 4, Tests and Results). They consisted on defining different topologies and progressive tests adding elements in each successive scenario. The tool used for evaluating their performance has been, mainly, Wireshark[4].

Wireshark is an open source packet analyzer. It allows capturing the traffic in the interface of interest and then visualizing every packet with additional information as the protocol used, the source or the destination.

In this work, Wireshark has been used to understand the underlying behavior of the different elements and the interactions between them.

In order to understand the principles of working of the testing environment, we build different topologies. This topologies will be deployed in the miniTower68 machine. For this tests, we will be capturing the packets with Wireshark and then filtering the OpenFlow packets in three different scenarios. The first scenario is with the Mininet topology running without any controller nor hypervisor, in the second one we will connect the topology to a remote Floodlight controller in bigTower machine and, in the final scenario, the OpenVirteX hypervisor will be added in between them, completing the set up. All the captures will be performed from the bigTower machine over the eth0 interface (See figure 7).

The test will be performed always in the same way:

1. Start Wireshark captures

---

[4] https://www.wireshark.org/

2. Launch required software
3. Mininet > pingall
4. @minitower68:$ sudo ovs-ofctl dump-flows s1
5. Stop Wireshark captures

In the successive section, we will provide the results of the commands listed above as well as the Wireshark captures in order to show and understand the behavior of the different studied scenarios.

4. Tests and Results

4.1. First Test: SimpleTopo

In this first experiment, a minimum topology is build and deployed with Mininet. Composed by only one switch with three hosts connected, its purpose is to be sure that all the parts perform as expected.

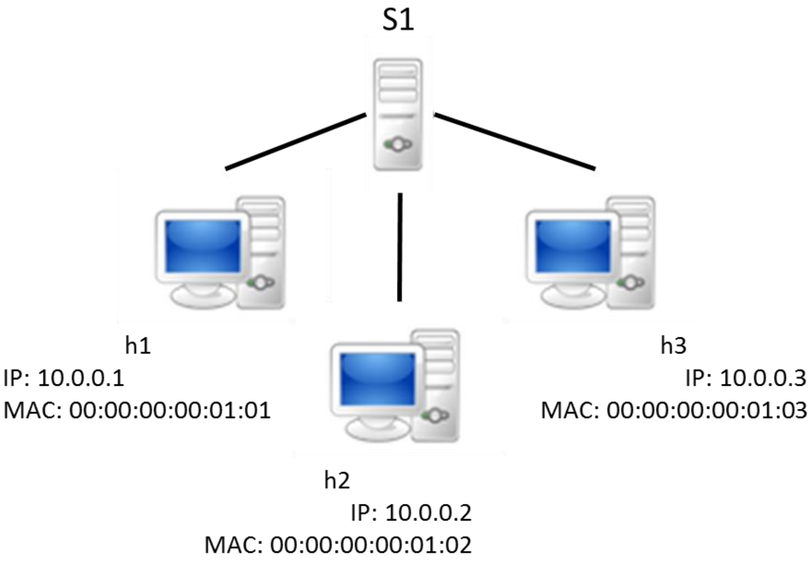
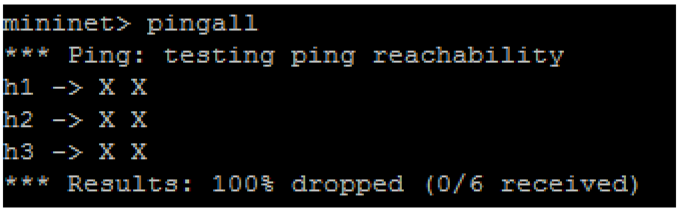


Figure 9. Topology created in SimpleTopo for this test

4.1.1. First scenario: SimpleTopo

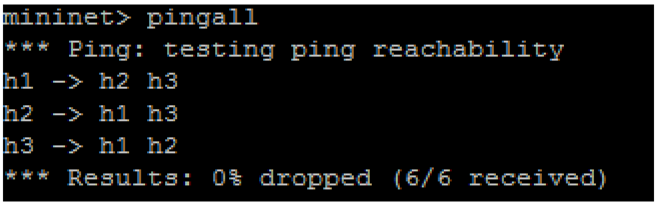


```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X
h2 -> X X
h3 -> X X
*** Results: 100% dropped (0/6 received)
```

Figure 10. In scenario 1, the hosts cannot reach each other

No OF packets captured in eth0, no visibility between hosts and no flow table on the switch. This makes sense as no one is pushing OpenFlow commands into the switch.

4.1.2. Second scenario: SimpleTopo + Floodlight Controller



```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

Figure 11. In scenario 2, all the hosts have visibility due to the flow table

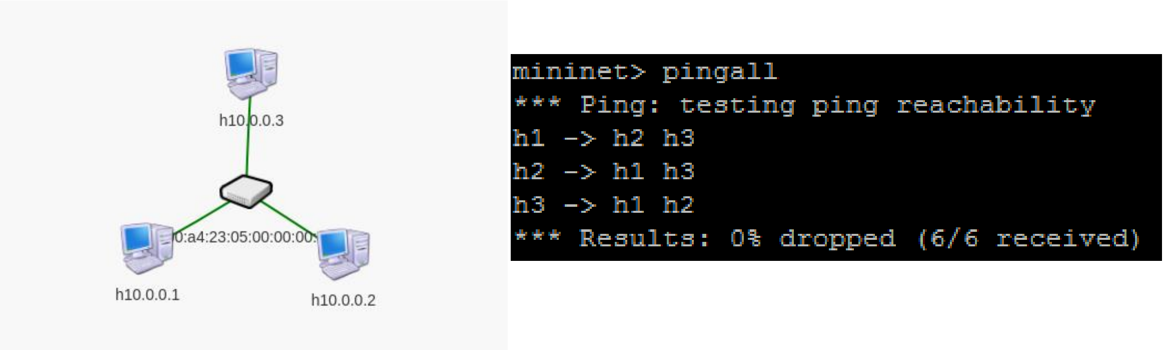
In this second scenario, we find that the remote controller works as expected and push a flow table into the switch and, in doing so, allowing the hosts to see each other.

```
luis@minitower68:~$ sudo ovs-ofctl dump-flows s1

NXST_FLOW reply (xid=0x4):

 cookie=0x2000001b000000, duration=1.231s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=1,dl_src=00:00:00:00:01:01,dl_dst=00:00:00:00:01:02,nw_src=10.0.0.1,
nw_dst=10.0.0.2 actions=output:2

 cookie=0x2000001c000000, duration=1.230s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=2,dl_src=00:00:00:00:01:02,dl_dst=00:00:00:00:01:01,nw_src=10.0.0.2,
nw_dst=10.0.0.1 actions=output:1

 cookie=0x2000001d000000, duration=1.226s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=1,dl_src=00:00:00:00:01:01,dl_dst=00:00:00:00:01:03,nw_src=10.0.0.1,
nw_dst=10.0.0.3 actions=output:3

 cookie=0x2000001e000000, duration=1.224s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=3,dl_src=00:00:00:00:01:03,dl_dst=00:00:00:00:01:01,nw_src=10.0.0.3,
nw_dst=10.0.0.1 actions=output:1

 cookie=0x2000001f000000, duration=1.219s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=2,dl_src=00:00:00:00:01:02,dl_dst=00:00:00:00:01:03,nw_src=10.0.0.2,
nw_dst=10.0.0.3 actions=output:3

 cookie=0x20000020000000, duration=1.217s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=3,dl_src=00:00:00:00:01:03,dl_dst=00:00:00:00:01:02,nw_src=10.0.0.3,
nw_dst=10.0.0.2 actions=output:2
```

Table 1. Flow table on switch S1.

4.1.3. Third scenario: SimpleTopo + OpenVirteX + Floodlight

In this final scenario, we test the whole set up with the minimum topology. Seems that everything works as expected as can be appreciated in the following figure.



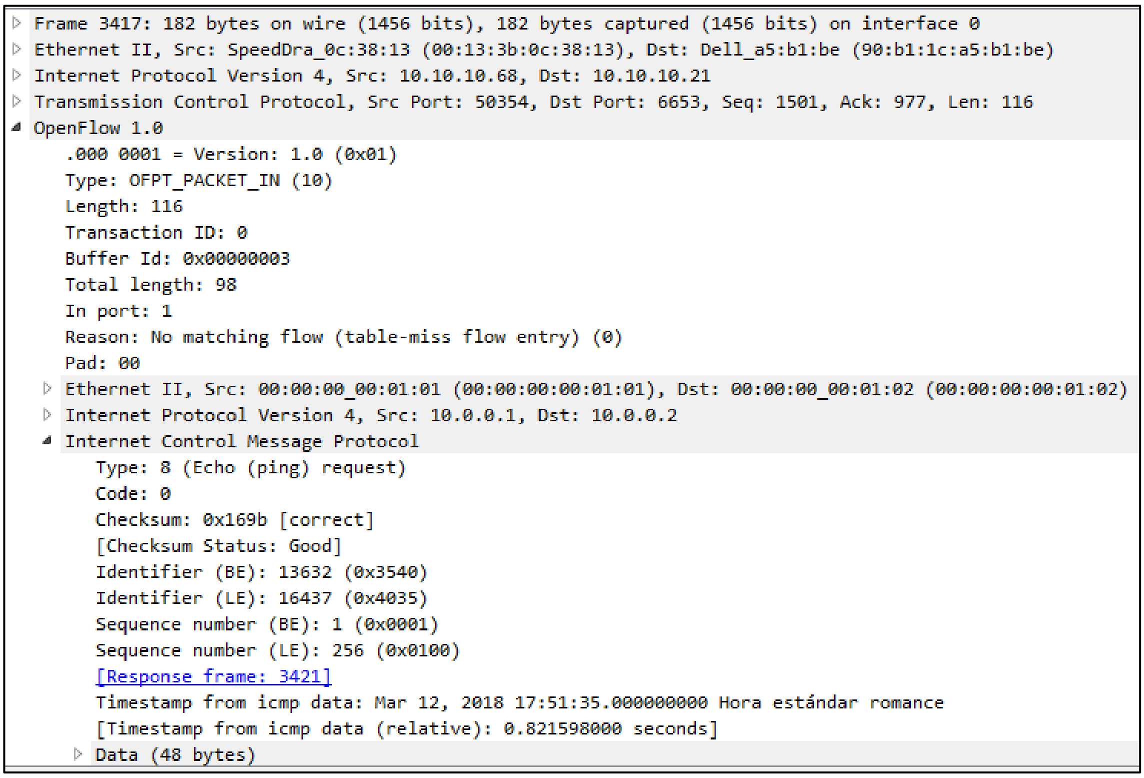
```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

Figure 12. Emulated network as seen from the controller and the hosts being able to reach each other

It is relevant to say that the network seen by the Floodlight controller is not the one created by the Mininet anymore, but the emulated one created by OpenVirteX on top of the former.

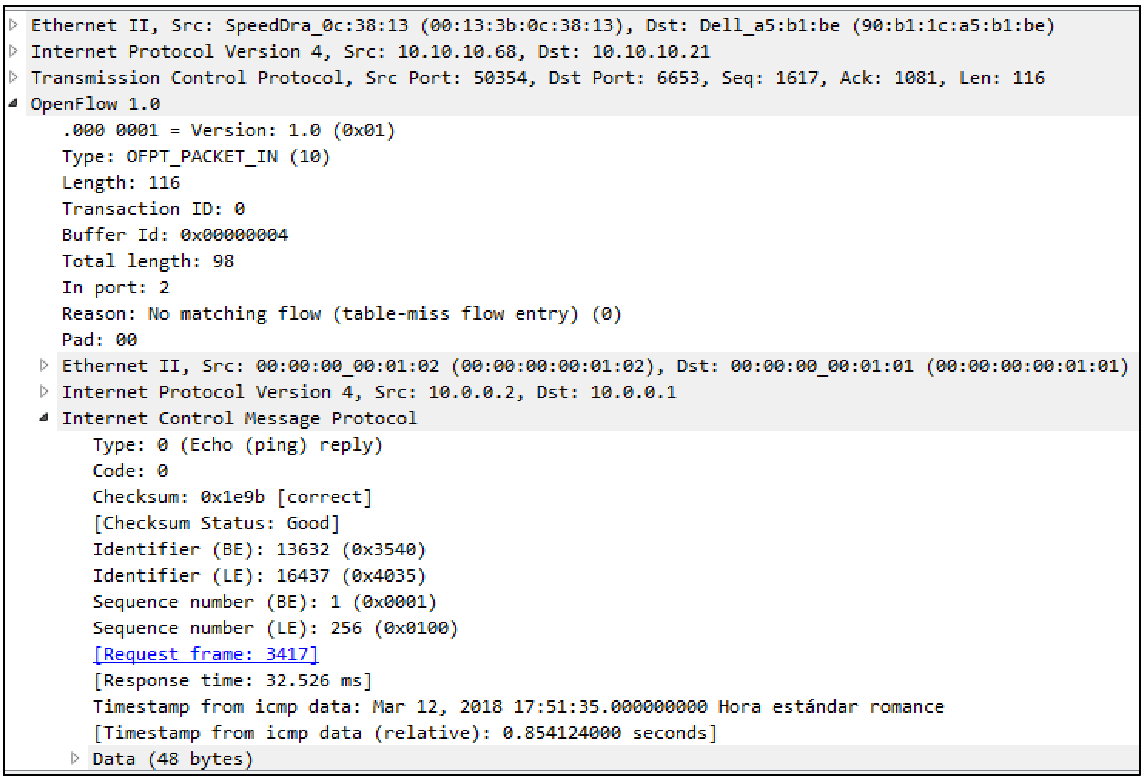
```
luis@minitower68:~/OpenVirteX/scripts$ sh simpleNet.sh
Virtual network has been created (network_id {u'mask': 16, u'networkAddress': 167772160, u'controllerUrls': [u'tcp:10.10.10.21:6653'], u'tenantId': 1}).
Virtual switch has been created (tenant_id 1, switch_id 00:a4:23:05:00:00:00:01)
Virtual port has been created (tenant_id 1, switch_id 00:a4:23:05:00:00:00:01, port_id 1)
Virtual port has been created (tenant_id 1, switch_id 00:a4:23:05:00:00:00:01, port_id 2)
Virtual port has been created (tenant_id 1, switch_id 00:a4:23:05:00:00:00:01, port_id 3)
Host (host_id 1) has been connected to virtual port
Host (host id 2) has been connected to virtual port
```

Table 2. Log of the OVX script that creates the emulated network



```
Frame 3417: 182 bytes on wire (1456 bits), 182 bytes captured (1456 bits) on interface 0
Ethernet II, Src: SpeedDra_0c:38:13 (00:13:3b:0c:38:13), Dst: Dell_a5:b1:be (90:b1:1c:a5:b1:be)
Internet Protocol Version 4, Src: 10.10.10.68, Dst: 10.10.10.21
Transmission Control Protocol, Src Port: 50354, Dst Port: 6653, Seq: 1501, Ack: 977, Len: 116
OpenFlow 1.0
    .000 0001 = Version: 1.0 (0x01)
    Type: OFPT_PACKET_IN (10)
    Length: 116
    Transaction ID: 0
    Buffer Id: 0x00000003
    Total length: 98
    In port: 1
    Reason: No matching flow (table-miss flow entry) (0)
    Pad: 00
    Ethernet II, Src: 00:00:00_00:01:01 (00:00:00:00:01:01), Dst: 00:00:00_00:01:02 (00:00:00:00:01:02)
    Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.2
    Internet Control Message Protocol
        Type: 8 (Echo (ping) request)
        Code: 0
        Checksum: 0x169b [correct]
        [Checksum Status: Good]
        Identifier (BE): 13632 (0x3540)
        Identifier (LE): 16437 (0x4035)
        Sequence number (BE): 1 (0x0001)
        Sequence number (LE): 256 (0x0100)
        [Response frame: 3421]
        Timestamp from icmp data: Mar 12, 2018 17:51:35.000000000 Hora estándar romance
        [Timestamp from icmp data (relative): 0.821598000 seconds]
      Data (48 bytes)
```

Figure 13. Capture of the ICMP echo request

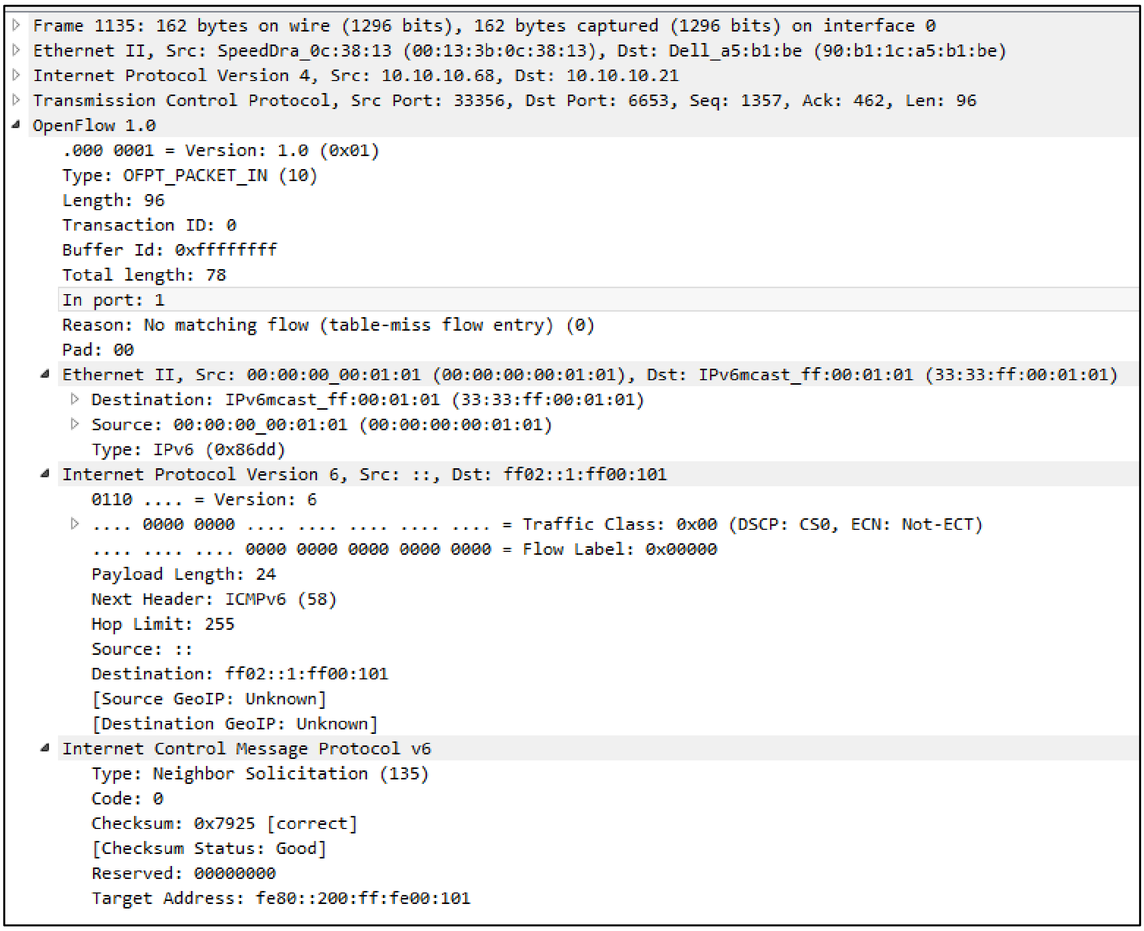


```
Ethernet II, Src: SpeedDra_0c:38:13 (00:13:3b:0c:38:13), Dst: Dell_a5:b1:be (90:b1:1c:a5:b1:be)
Internet Protocol Version 4, Src: 10.10.10.68, Dst: 10.10.10.21
Transmission Control Protocol, Src Port: 50354, Dst Port: 6653, Seq: 1617, Ack: 1081, Len: 116
OpenFlow 1.0
    .000 0001 = Version: 1.0 (0x01)
    Type: OFPT_PACKET_IN (10)
    Length: 116
    Transaction ID: 0
    Buffer Id: 0x00000004
    Total length: 98
    In port: 2
    Reason: No matching flow (table-miss flow entry) (0)
    Pad: 00
    Ethernet II, Src: 00:00:00_00:01:02 (00:00:00:00:01:02), Dst: 00:00:00_00:01:01 (00:00:00:00:01:01)
    Internet Protocol Version 4, Src: 10.0.0.2, Dst: 10.0.0.1
    Internet Control Message Protocol
        Type: 0 (Echo (ping) reply)
        Code: 0
        Checksum: 0x1e9b [correct]
        [Checksum Status: Good]
        Identifier (BE): 13632 (0x3540)
        Identifier (LE): 16437 (0x4035)
        Sequence number (BE): 1 (0x0001)
        Sequence number (LE): 256 (0x0100)
        [Request frame: 3417]
        [Response time: 32.526 ms]
        Timestamp from icmp data: Mar 12, 2018 17:51:35.000000000 Hora estándar romance
        [Timestamp from icmp data (relative): 0.854124000 seconds]
      Data (48 bytes)
```

Figure 14. Capture of the reply to the ICMP echo reply

```
luis@minitower68:~ $ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x100000004, duration=1.201s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=1,dl_src=00:00:00:00:01:01,dl_dst=00:00:00:00:01:02,nw_src=10
.0.0.1,nw_dst=10.0.0.2
actions=mod_nw_dst:1.0.0.2,mod_nw_src:1.0.0.1,mod_nw_src:10.0.0.1,mod_nw_dst:
10.0.0.2,output:2
 cookie=0x100000008, duration=1.195s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=2,dl_src=00:00:00:00:01:02,dl_dst=00:00:00:00:01:01,nw_src=10
.0.0.2,nw_dst=10.0.0.1
actions=mod_nw_dst:1.0.0.1,mod_nw_src:1.0.0.2,mod_nw_src:10.0.0.2,mod_nw_dst:
10.0.0.1,output:1
 cookie=0x100000009, duration=1.185s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=1,dl_src=00:00:00:00:01:01,dl_dst=00:00:00:00:01:03,nw_src=10
.0.0.1,nw_dst=10.0.0.3
actions=mod_nw_dst:1.0.0.3,mod_nw_src:1.0.0.1,mod_nw_src:10.0.0.1,mod_nw_dst:
10.0.0.3,output:3
 cookie=0x10000000a, duration=1.180s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=3,dl_src=00:00:00:00:01:03,dl_dst=00:00:00:00:01:01,nw_src=10
.0.0.3,nw_dst=10.0.0.1
actions=mod_nw_dst:1.0.0.1,mod_nw_src:1.0.0.3,mod_nw_src:10.0.0.3,mod_nw_dst:
10.0.0.1,output:1
 cookie=0x100000006, duration=1.159s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=2,dl_src=00:00:00:00:01:02,dl_dst=00:00:00:00:01:03,nw_src=10
.0.0.2,nw_dst=10.0.0.3
actions=mod_nw_dst:1.0.0.3,mod_nw_src:1.0.0.2,mod_nw_src:10.0.0.2,mod_nw_dst:
10.0.0.3,output:3
 cookie=0x100000002, duration=1.156s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1,
priority=1,ip,in_port=3,dl_src=00:00:00:00:01:03,dl_dst=00:00:00:00:01:02,nw_src=10
.0.0.3,nw_dst=10.0.0.2
actions=mod_nw_dst:1.0.0.2,mod_nw_src:1.0.0.3,mod_nw_src:10.0.0.3,mod_nw_dst:
10.0.0.2,output:2
```

Table 3. Flow table in S1 switch in scenario 3 test



```
▷ Frame 1135: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits) on interface 0
▷ Ethernet II, Src: SpeedDra_0c:38:13 (00:13:3b:0c:38:13), Dst: Dell_a5:b1:be (90:b1:1c:a5:b1:be)
▷ Internet Protocol Version 4, Src: 10.10.10.68, Dst: 10.10.10.21
▷ Transmission Control Protocol, Src Port: 33356, Dst Port: 6653, Seq: 1357, Ack: 462, Len: 96
◢ OpenFlow 1.0
    .000 0001 = Version: 1.0 (0x01)
    Type: OFPT_PACKET_IN (10)
    Length: 96
    Transaction ID: 0
    Buffer Id: 0xffffffff
    Total length: 78
    In port: 1
    Reason: No matching flow (table-miss flow entry) (0)
    Pad: 00
  ◢ Ethernet II, Src: 00:00:00_00:01:01 (00:00:00:00:01:01), Dst: IPv6mcast_ff:00:01:01 (33:33:ff:00:01:01)
    ▷ Destination: IPv6mcast_ff:00:01:01 (33:33:ff:00:01:01)
    ▷ Source: 00:00:00_00:01:01 (00:00:00:00:01:01)
      Type: IPv6 (0x86dd)
  ◢ Internet Protocol Version 6, Src: ::, Dst: ff02::1:ff00:101
      0110 .... = Version: 6
    ▷ .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
      .... .... .... 0000 0000 0000 0000 0000 = Flow Label: 0x00000
      Payload Length: 24
      Next Header: ICMPv6 (58)
      Hop Limit: 255
      Source: ::
      Destination: ff02::1:ff00:101
      [Source GeoIP: Unknown]
      [Destination GeoIP: Unknown]
  ◢ Internet Control Message Protocol v6
      Type: Neighbor Solicitation (135)
      Code: 0
      Checksum: 0x7925 [correct]
      [Checksum Status: Good]
      Reserved: 00000000
      Target Address: fe80::200:ff:fe00:101
```

Figure 15. Capture of the ICMP v6 packet

4.2. Second Test: 2 Switches, 4 Hosts

Once the basics are working as expected, we set a more complex topology. In this second case of study, the composition will be 2 switches with 2 hosts each one as shown in the following figure.

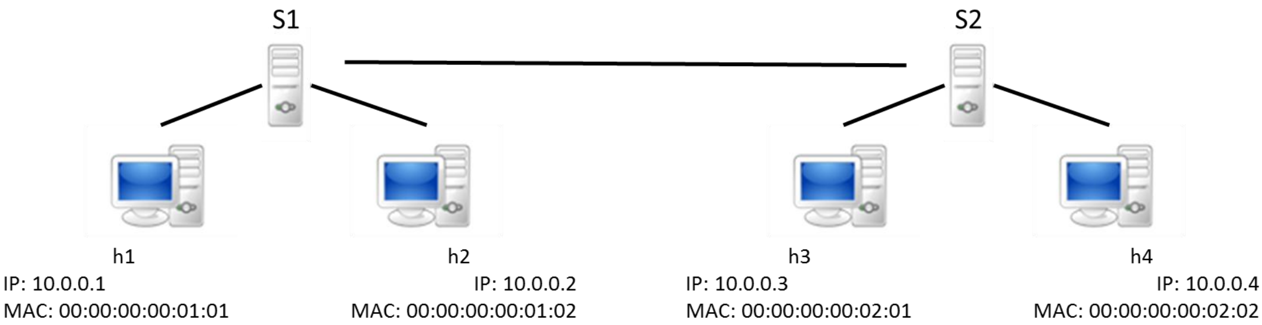
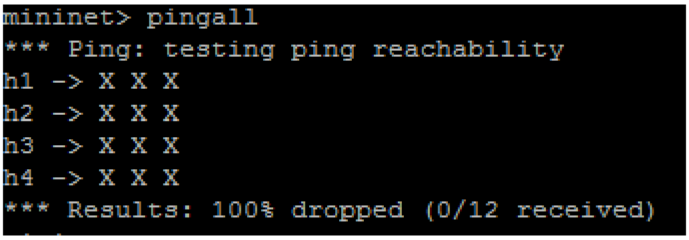


Figure 16. Mininet Network Topology for the second test

As in the previous test, we will work in three different progressive scenarios to be sure that each part performs as expected. The scenarios and the procedure will be the same ones.

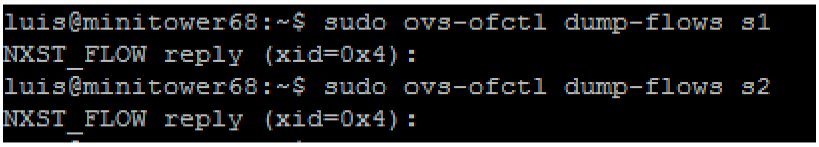
4.2.1. First scenario: 2sw-4hs



```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X
h2 -> X X X
h3 -> X X X
h4 -> X X X
*** Results: 100% dropped (0/12 received)
```

Figure 17. Hosts are unable to reach any other when the network does not have a controller

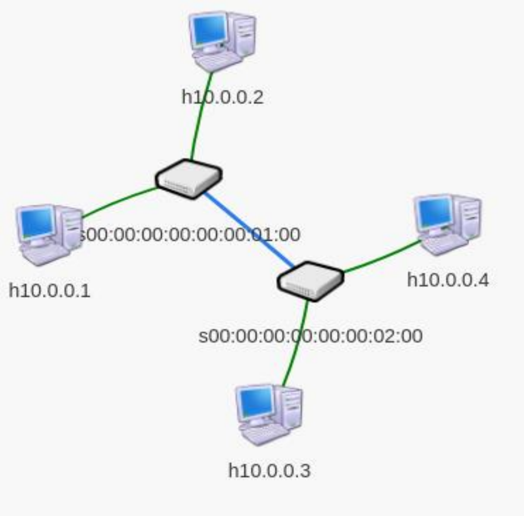
As expected, the network is not able to function in the absence of a controller, we do not see any OF packets on the interface eth0, there is no flow tables and the hosts are not able to communicate.

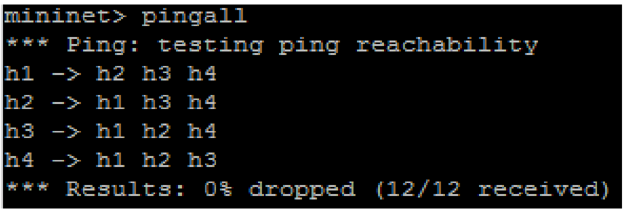


```
luis@minitower68:~$ sudo ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
luis@minitower68:~$ sudo ovs-ofctl dump-flows s2
NXST_FLOW reply (xid=0x4):
```

Figure 18. No flow criteria are established in the switches

4.2.2. Second scenario: 2sw-4hs + Floodlight Controller





```
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 19. Network as seen from the controller and reachability of the hosts

In this scenario we can observe that the network perfoms as expected, discovering all the elements in the network and pushing the corresponding flow tables into the switches.

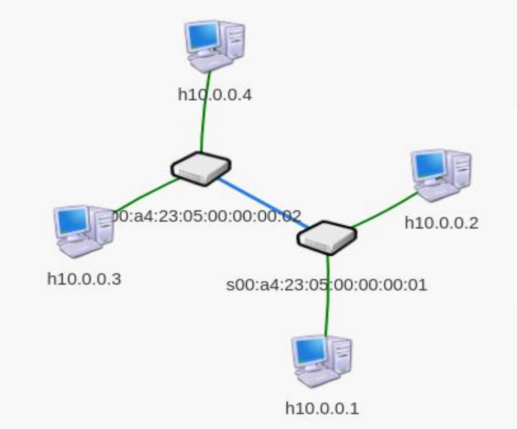
```
NXST_FLOW reply (xid=0x4):

 cookie=0x20000038000000, duration=1.844s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1, priority=1, ip, in_port=1, dl_src=00:00:00:00:01:01,
dl_dst=00:00:00:00:01:02,nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=output:2

 cookie=0x20000039000000, duration=1.843s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1, priority=1, ip,in_port=2, dl_src=00:00:00:00:01:02,
dl_dst=00:00:00:00:01:01, nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=output:1

 cookie=0x2000003a000000, duration=1.838s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1, priority=1, ip,in_port=1,dl_src=00:00:00:00:01:01,
dl_dst=00:00:00:00:02:01, nw_src=10.0.0.1,nw_dst=10.0.0.3 actions=output:3

 cookie=0x2000003b000000, duration=1.837s, table=0, n_packets=2, n_bytes=196,
idle_timeout=5, idle_age=1, priority=1, ip, in_port=3, dl_src=00:00:00:00:02:01,
dl_dst=00:00:00:00:01:01,nw_src=10.0.0.3,nw_dst=10.0.0.1 actions=output:1

 cookie=0x2000003c000000, duration=1.832s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1, priority=1,ip,in_port=1,dl_src=00:00:00:00:01:01,
dl_dst=00:00:00:00:02:02,nw_src=10.0.0.1,nw_dst=10.0.0.4 actions=output:3

 cookie=0x2000003d000000, duration=1.830s, table=0, n_packets=2, n_bytes=196,
idle_timeout=5, idle_age=1, priority=1,ip,in_port=3,dl_src=00:00:00:00:02:02,
dl_dst=00:00:00:00:01:01,nw_src=10.0.0.4,nw_dst=10.0.0.1 actions=output:1

 cookie=0x2000003e000000, duration=1.825s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1, priority=1,ip,in_port=2,dl_src=00:00:00:00:01:02,
dl_dst=00:00:00:00:02:01,nw_src=10.0.0.2,nw_dst=10.0.0.3 actions=output:3

 cookie=0x2000003f000000, duration=1.824s, table=0, n_packets=2, n_bytes=196,
idle_timeout=5, idle_age=1, priority=1, ip, in_port=3, dl_src=00:00:00:00:02:01,
dl_dst=00:00:00:00:01:02,nw_src=10.0.0.3,nw_dst=10.0.0.2 actions=output:2

 cookie=0x20000040000000, duration=1.821s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=1, priority=1, ip, in_port=2, dl_src=00:00:00:00:01:02,
dl_dst=00:00:00:00:02:02, nw_src=10.0.0.2,nw_dst=10.0.0.4 actions=output:3

 cookie=0x20000041000000, duration=1.819s, table=0, n_packets=2, n_bytes=196,
idle_timeout=5, idle_age=1, priority=1, ip, in_port=3, dl_src=00:00:00:00:02:02,
dl_dst=00:00:00:00:01:02, nw_src=10.0.0.4,nw_dst=10.0.0.2 actions=output:2
```

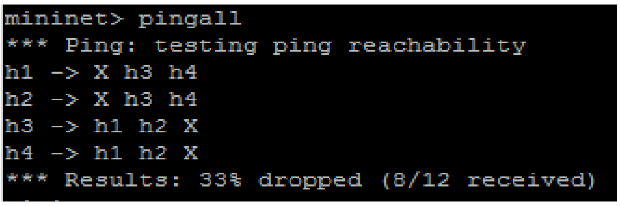
Table 4. Flow tables in S1 in scenario 2

```
NXST_FLOW reply (xid=0x4):
 cookie=0x2000003a000000, duration=3.540s, table=0, n_packets=2, n_bytes=196,
idle_timeout=5, idle_age=3,
priority=1,ip,in_port=3,dl_src=00:00:00:00:01:01,dl_dst=00:00:00:00:02:01,nw_src=10.
0.0.1,nw_dst=10.0.0.3 actions=output:1
 cookie=0x2000003b000000, duration=3.539s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=3,
priority=1,ip,in_port=1,dl_src=00:00:00:00:02:01,dl_dst=00:00:00:00:01:01,nw_src=10.
0.0.3,nw_dst=10.0.0.1 actions=output:3
 cookie=0x2000003c000000, duration=3.534s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=3,
priority=1,ip,in_port=3,dl_src=00:00:00:00:01:01,dl_dst=00:00:00:00:02:02,nw_src=10.
0.0.1,nw_dst=10.0.0.4 actions=output:2
 cookie=0x2000003d000000, duration=3.532s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=3,
priority=1,ip,in_port=2,dl_src=00:00:00:00:02:02,dl_dst=00:00:00:00:01:01,nw_src=10.
0.0.4,nw_dst=10.0.0.1 actions=output:3
 cookie=0x2000003e000000, duration=3.527s, table=0, n_packets=2, n_bytes=196,
idle_timeout=5, idle_age=3,
priority=1,ip,in_port=3,dl_src=00:00:00:00:01:02,dl_dst=00:00:00:00:02:01,nw_src=10.
0.0.2,nw_dst=10.0.0.3 actions=output:1
 cookie=0x2000003f000000, duration=3.526s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=3,
priority=1,ip,in_port=1,dl_src=00:00:00:00:02:01,dl_dst=00:00:00:00:01:02,nw_src=10.
0.0.3,nw_dst=10.0.0.2 actions=output:3
 cookie=0x20000040000000, duration=3.523s, table=0, n_packets=2, n_bytes=196,
idle_timeout=5, idle_age=3,
priority=1,ip,in_port=3,dl_src=00:00:00:00:01:02,dl_dst=00:00:00:00:02:02,nw_src=10.
0.0.2,nw_dst=10.0.0.4 actions=output:2
 cookie=0x20000041000000, duration=3.521s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=3,
priority=1,ip,in_port=2,dl_src=00:00:00:00:02:02,dl_dst=00:00:00:00:01:02,nw_src=10.
0.0.4,nw_dst=10.0.0.2 actions=output:3
 cookie=0x20000042000000, duration=3.515s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=3,
priority=1,ip,in_port=1,dl_src=00:00:00:00:02:01,dl_dst=00:00:00:00:02:02,nw_src=10.
0.0.3,nw_dst=10.0.0.4 actions=output:2
 cookie=0x20000043000000, duration=3.514s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=3,
priority=1,ip,in_port=2,dl_src=00:00:00:00:02:02,dl_dst=00:00:00:00:02:01,nw_src=10.
0.0.4,nw_dst=10.0.0.3 actions=output:1
```

Table 5. Flow tables in S2 in scenario 2

4.2.3. Third scenario: 2sw-4hs + OpenVirteX + Floodlight





```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X h3 h4
h2 -> X h3 h4
h3 -> h1 h2 X
h4 -> h1 h2 X
*** Results: 33% dropped (8/12 received)
```

Figure 20. Network as seen from the controller and host reachability

With all the elements interacting seem to be some visibility problems between the hosts connected to the same switch, but we can see that the emulated network created with the OVX is deployed correctly.

From the previous scenario we know that the OVS switches as well as the Mininet topology perform correctly so the problem must be in the OpenVirteX layer. To discover what went wrong we proceed to compare the flow tables of both scenarios (tables 4,5 and 6,7 respectively) and we can observe that Floodlight is defining one rule for every possible combination between two different hosts except for those which share a switch. In order to find out what is causing this effect we ran the 3rd series of tests.

```
cookie=0x10000000b, duration=7.402s, table=0, n_packets=1, n_bytes=42, idle_timeout=5,
idle_age=2, priority=1, arp, in_port=3, dl_src=a4:23:05:01:00:00, dl_dst=a4:23:05:10:00:0e,
actions=mod_dl_src:00:00:00:00:02:02, mod_dl_dst:00:00:00:00:01:02, output:2

 cookie=0x100000007, duration=2.353s, table=0, n_packets=0, n_bytes=0, idle_timeout=5,
idle_age=2, priority=1, arp, in_port=2, dl_src=00:00:00:00:01:02, dl_dst=00:00:00:00:02:02
actions=mod_dl_src:a4:23:05:01:00:00, mod_dl_dst:a4:23:05:10:00:04, output:3

 cookie=0x100000008, duration=2.341s, table=0, n_packets=0, n_bytes=0, idle_timeout=5,
idle_age=2, priority=1, arp, in_port=3, dl_src=a4:23:05:01:00:00, dl_dst=a4:23:05:10:00:0b
actions=mod_dl_src:00:00:00:00:02:01, mod_dl_dst:00:00:00:00:01:01, output:1

 cookie=0x100000004, duration=2.340s, table=0, n_packets=0, n_bytes=0, idle_timeout=5,
idle_age=2, priority=1,arp,in_port=3,dl_src=a4:23:05:01:00:00,dl_dst=a4:23:05:10:00:0a
actions=mod_dl_src:00:00:00:00:02:01,mod_dl_dst:00:00:00:00:01:02,output:2

 cookie=0x100000009, duration=2.322s, table=0, n_packets=0, n_bytes=0, idle_timeout=5,
idle_age=2, priority=1,arp,in_port=1,dl_src=00:00:00:00:01:01,dl_dst=00:00:00:00:02:01
actions=mod_dl_src:a4:23:05:01:00:00,mod_dl_dst:a4:23:05:10:00:07,output:3

 cookie=0x100000003, duration=2.316s, table=0, n_packets=0, n_bytes=0, idle_timeout=5,
idle_age=2, priority=1,arp,in_port=2,dl_src=00:00:00:00:01:02,dl_dst=00:00:00:00:02:01
actions=mod_dl_src:a4:23:05:01:00:00,mod_dl_dst:a4:23:05:10:00:03,output:3

 cookie=0x10000000c, duration=7.383s, table=0, n_packets=1, n_bytes=98,
idle_timeout=5, idle_age=4, priority=1, ip, in_port=2, dl_src=00:00:00:00:01:02,
dl_dst=00:00:00:00:02:02, nw_src=10.0.0.20, nw_dst=10.0.0.40
actions=mod_nw_dst:1.0.0.5, mod_nw_src:1.0.0.3, mod_dl_src:a4:23:05:01:00:00,
mod_dl_dst:a4:23:05:10:00:04, output:3

 cookie=0x10000000d, duration=7.378s,table=0, n_packets=1, n_bytes=98, idle_timeout=5,
idle_age=4, priority=1, ip, in_port=3, dl_src=a4:23:05:01:00:00, dl_dst=a4:23:05:10:00:0e,
nw_src=1.0.0.5,nw_dst=1.0.0.3 actions=mod_nw_src:10.0.0.40, mod_nw_dst:10.0.0.20,
mod_dl_src:00:00:00:00:02:02,mod_dl_dst:00:00:00:00:01:02, output:2

 cookie=0x100000002, duration=4.357s, table=0, n_packets=0, n_bytes=0, idle_timeout=5,
idle_age=4, priority=1, ip, in_port=3, dl_src=a4:23:05:01:00:00, dl_dst=a4:23:05:10:00:0f,
nw_src=1.0.0.5, nw_dst=1.0.0.2 actions=mod_nw_src:10.0.0.40, mod_nw_dst:10.0.0.10,
mod_dl_src:00:00:00:00:02:02, mod_dl_dst:00:00:00:00:01:01, output:1

 cookie=0x100000006, duration=4.329s, table=0, n_packets=0, n_bytes=0, idle_timeout=5,
idle_age=4, priority=1, ip, in_port=1, dl_src=00:00:00:00:01:01, dl_dst=00:00:00:00:02:02,
nw_src=10.0.0.10, nw_dst=10.0.0.40 actions=mod_nw_dst:1.0.0.5, mod_nw_src:1.0.0.2,
mod_dl_src:a4:23:05:01:00:00, mod_dl_dst:a4:23:05:10:00:08, output:3
```

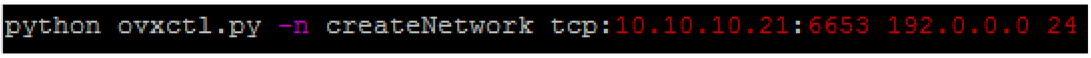
Table 6. Flows in S1 in scenario 3

```
cookie=0x100000004, duration=6.380s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
idle_age=3, priority=1, ip, in_port=3, dl_src=a4:23:05:01:00:00, dl_dst=a4:23:05:10:00:0e,
nw_src=1.0.0.4, nw_dst=1.0.0.3 actions=mod_nw_src:10.0.0.1, mod_nw_dst:10.0.0.3,
mod_dl_src:00:00:00:00:01:01, mod_dl_dst:00:00:00:00:02:01, output:1

 cookie=0x100000009, duration=6.366s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
idle_age=3, priority=1, ip, in_port=1, dl_src=00:00:00:00:02:01, dl_dst=00:00:00:00:01:01,
nw_src=10.0.0.3, nw_dst=10.0.0.1 actions=mod_nw_dst:1.0.0.4, mod_nw_src:1.0.0.3,
mod_dl_src:a4:23:05:01:00:00, mod_dl_dst:a4:23:05:10:00:04, output:3

 cookie=0x10000000b, duration=3.341s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
idle_age=3, priority=1, ip, in_port=3, dl_src=a4:23:05:01:00:00, dl_dst=a4:23:05:10:00:0a,
nw_src=1.0.0.5, nw_dst=1.0.0.3 actions=mod_nw_src:10.0.0.2, mod_nw_dst:10.0.0.3,
mod_dl_src:00:00:00:00:01:02, mod_dl_dst:00:00:00:00:02:01, output:1

 cookie=0x10000000e, duration=3.327s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
idle_age=3, priority=1, ip,in_port=1, dl_src=00:00:00:00:02:01, dl_dst=00:00:00:00:01:02,
nw_src=10.0.0.3, nw_dst=10.0.0.2 actions=mod_nw_dst:1.0.0.5, mod_nw_src:1.0.0.3,
mod_dl_src:a4:23:05:01:00:00, mod_dl_dst:a4:23:05:10:00:03, output:3

 cookie=0x10000000a, duration=3.321s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
idle_age=0, priority=1, ip, in_port=3, dl_src=a4:23:05:01:00:00, dl_dst=a4:23:05:10:00:0b,
nw_src=1.0.0.5, nw_dst=1.0.0.2 actions=mod_nw_src:10.0.0.2, mod_nw_dst:10.0.0.4,
mod_dl_src:00:00:00:00:01:02, mod_dl_dst:00:00:00:00:02:02, output:2

 cookie=0x100000006, duration=3.301s, table=0, n_packets=1, n_bytes=98, idle_timeout=5,
idle_age=0, priority=1, ip, in_port=2, dl_src=00:00:00:00:02:02, dl_dst=00:00:00:00:01:02,
nw_src=10.0.0.4,nw_dst=10.0.0.2 actions=mod_nw_dst:1.0.0.5, mod_nw_src:1.0.0.2,
mod_dl_src:a4:23:05:01:00:00, mod_dl_dst:a4:23:05:10:00:07, output:3

 cookie=0x10000000f, duration=0.287s, table=0, n_packets=0, n_bytes=0, idle_timeout=5,
idle_age=0, priority=1, ip, in_port=2, dl_src=00:00:00:00:02:02, dl_dst=00:00:00:00:01:01,
nw_src=10.0.0.4, nw_dst=10.0.0.1 actions=mod_nw_dst:1.0.0.4, mod_nw_src:1.0.0.2,
mod_dl_src:a4:23:05:01:00:00, mod_dl_dst:a4:23:05:10:00:08, output:3

 cookie=0x100000011, duration=0.283s, table=0, n_packets=0, n_bytes=0, idle_timeout=5,
idle_age=0, priority=1, ip, in_port=3, dl_src=a4:23:05:01:00:00, dl_dst=a4:23:05:10:00:0f,
nw_src=1.0.0.4, nw_dst=1.0.0.2 actions=mod_nw_src:10.0.0.1, mod_nw_dst:10.0.0.4,
mod_dl_src:00:00:00:00:01:01, mod_dl_dst:00:00:00:00:02:02, output:2
```

Table 7. Flows in S2 in scenario 3

4.1. Third Test: SimpleTopo and 2sw-4hs with small variations

In the effort of discover what is causing the malfunctions seen in the third scenario of second test we realize that, unwittingly, we have been using the same address space in both the Mininet network and the OpenVirteX virtual network (See code annexes in sections 6.1.1 to 6.2.2). Considering this could be covering some mistakes we decide to try changing the network address space of the OVX VN.



```
python ovxctl.py -n createNetwork tcp:10.10.10.21:6653 192.0.0.0 24
```

Figure 21. Address space change

By changing this line in the script of the last test we should see hosts addresses like 192.0.0.1, 192.0.0.2 and so on. It should be this way because the Floodlight controller is not connected to the Mininet network but to the VN created by the OVX, which is defined in the script that we are modifying.

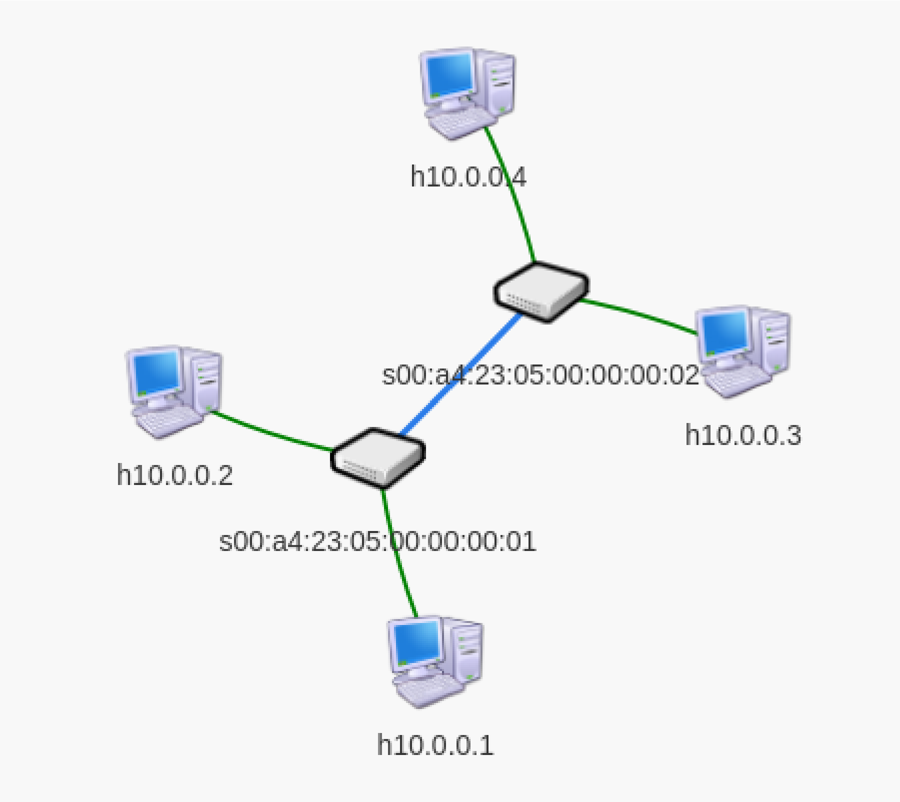
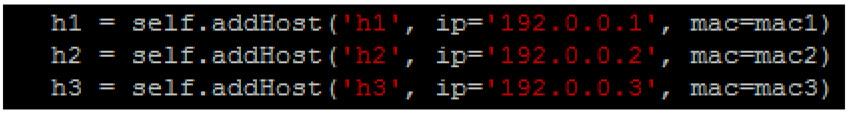


Figure 22. Network as seen from the Floodlight controller

But as shown in the figure above, this is not the case. For some reason the OVX is not performing as expected. After some research in the online help groups of OVX, we found some users with the same issue unsolved, and answers of the developers seem to point out that it is an unresolved issue.

For more security, we try the same changes in the first tested topology, in this case, we decide to change the addresses of the physical network. We obtained similar results.



```
h1 = self.addHost('h1', ip='192.0.0.1', mac=mac1)
h2 = self.addHost('h2', ip='192.0.0.2', mac=mac2)
h3 = self.addHost('h3', ip='192.0.0.3', mac=mac3)
```

Figure 23. Changes in the physical addresses

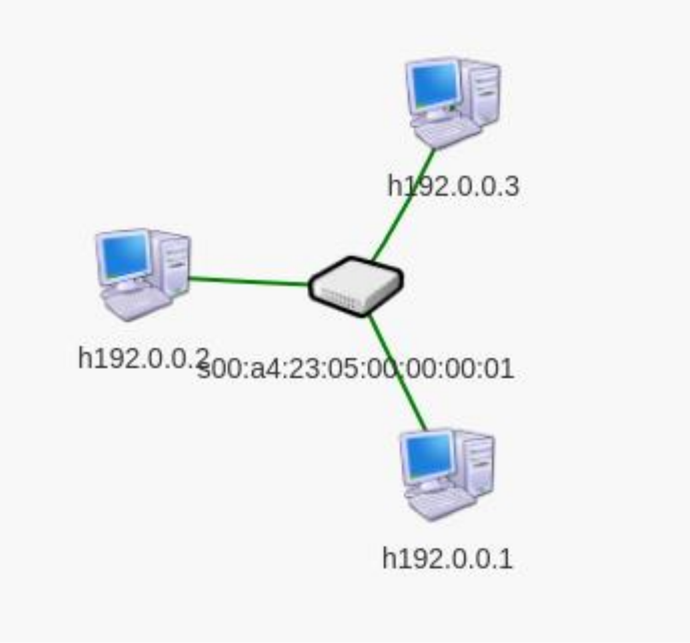


Figure 24. Network as seen from the Floodlight

With this second experiment we can assure that OVX is being transparent for most of the information. At least for the host topology. It seems to, at least, show the virtual switches as defined in the OVX script and correctly set the virtual links. There is no apparent reason why it should act this way with the virtual hosts.

Our hypothesis is that OpenVirteX has some malfunctions when deployed on an environment that differs from the one on its VM, or at least there is something in the configuration of our machines that make it perform worse, therefore we understand that it is a error in the OpenVirteX software.

5. Conclusions and future development:

The main conclusion is that even if the personal objectives of understanding SDN, learn about network virtualization and get fluency in the use of different tools and platforms (such as Mininet, OpenVirteX, Floodlight, Wireshark…) has been largely achieved, the final implementation still needs some improvement to be ready for its originally intended use. Even so, it is a strong base in which have the potential to play a relevant role in the line of investigation initiated by Crestani.

As a future development, the first thing to do is to find a way to overcome the problems we found with OVX and then the next big step is to develop from SDN Controller up (API Northbound interface, SDN Apps and so on) in order to build a reliable and realistic proof of concept of the original sniffer.

Bibliography:

[1] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu and H. Yu, "Advances in Network Simulation," IEEE, 2000.

[2] K. P. A. P. H. W. M. H. A. Richard M. Fujimoto, "Large-Scale Network Simulation: How Big? How Fast?," IEEE, Atlanta, 2003.

[3] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," IEEE.

[4] A. Darabseh, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk and A. Rindos, "SDSecurity: A Software Defined Security Experimental Framework," IEE ICC, Stafford. UK; Raleigh, North Carolina, USA; Research Triangle Park, North Carolina, USA, 2015.

[5] T. R. A. R. Tariq Javid, "A Layer2 Firewall for Software Defined Network," Conference on Information Assurance and Cyber Security (CIACS), University Taxila Cantt-47070, Pakistan, 2014.

[6] J. H. H. K. Y. K. S. Y. S. Lim, "A SDN-Oriented DDoS Blocking Scheme for Botnet-Based Attacks," IEE, Seoul, Korea; Daejon, Korea, 2014.

[7] R. K. R. M. K. B. Adel Zaalouk, "OrchSec: An Orchestrator-Based Architecture For Enhancing Network-Security Using Network Monitoring And SDN Control Functions," IEEE, Aachen, Germany; Darmstadt, Germany, 2014.

[8] R. L. S. d. Oliveira and A. A. Shinoda, "Using Mininet for Emulation and Prototyping Software-Defined Networks," IEEE, Ilha Solteira, Brazil; Jales, Brazil, 2014.

[9] Pooja and M. Sood, "SDN and Mininet: Some Basic Concepts," Int. J. Advanced Networking and Applications, Shimla, 2015.

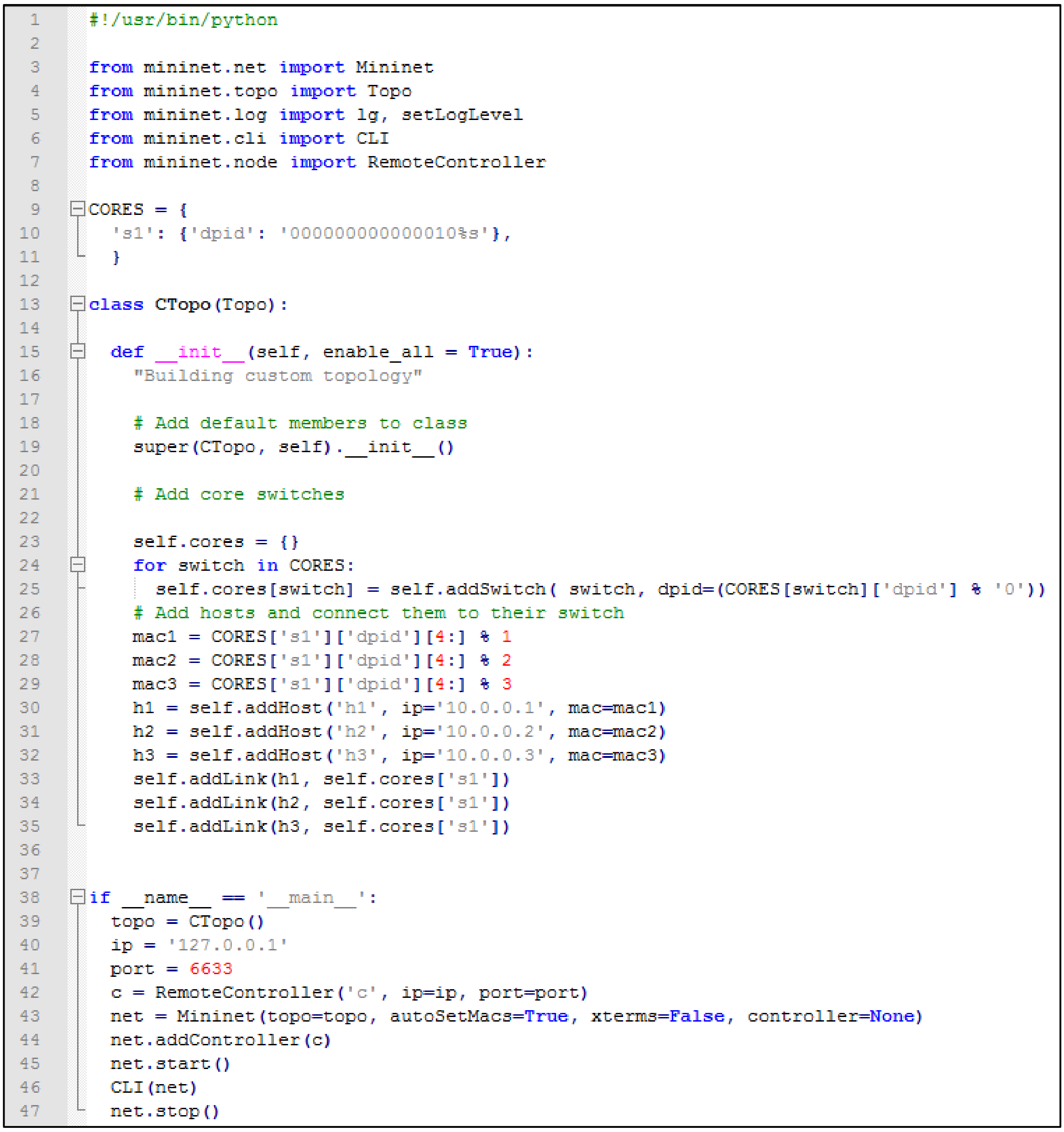
[10] J. S. a. N. S. G. Karamjeet Kaur, "Mininet as Software Defined Networking Testing Platform," International Conference on Communication, Computing & Systems ICCCS, Ferozepur, India, 2014.

[11] P. A. D. L. J. J. R.-M. J. L. a. J. F. Jose Ordonez-Lucena, "Network Slicing for 5G with SDN/NFV: Concepts, Architectures, and Challenges," IEEE Communications Magazine, Granada, 2017.

6. Annexes:

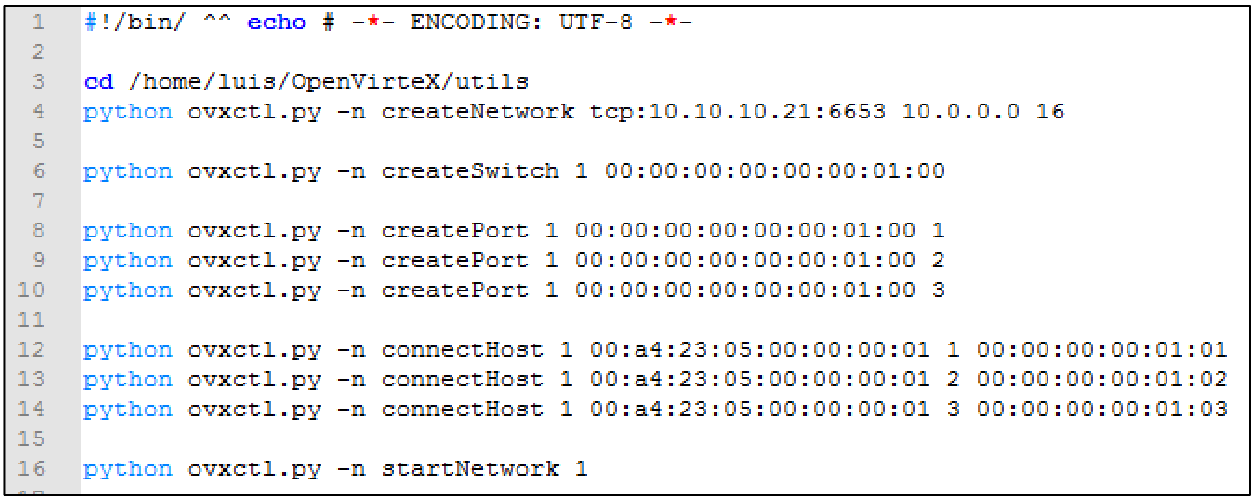
6.1. First Case of Study: simpleTopo

6.1.1. Mininet Code



```python
#!/usr/bin/python

from mininet.net import Mininet
from mininet.topo import Topo
from mininet.log import lg, setLogLevel
from mininet.cli import CLI
from mininet.node import RemoteController

CORES = {
  's1': {'dpid': '0000000000000010%s'},
  }

class CTopo(Topo):

  def __init__(self, enable_all = True):
    "Building custom topology"

    # Add default members to class
    super(CTopo, self).__init__()

    # Add core switches

    self.cores = {}
    for switch in CORES:
      self.cores[switch] = self.addSwitch( switch, dpid=(CORES[switch]['dpid'] % '0'))
    # Add hosts and connect them to their switch
    mac1 = CORES['s1']['dpid'][4:] % 1
    mac2 = CORES['s1']['dpid'][4:] % 2
    mac3 = CORES['s1']['dpid'][4:] % 3
    h1 = self.addHost('h1', ip='10.0.0.1', mac=mac1)
    h2 = self.addHost('h2', ip='10.0.0.2', mac=mac2)
    h3 = self.addHost('h3', ip='10.0.0.3', mac=mac3)
    self.addLink(h1, self.cores['s1'])
    self.addLink(h2, self.cores['s1'])
    self.addLink(h3, self.cores['s1'])


if __name__ == '__main__':
  topo = CTopo()
  ip = '127.0.0.1'
  port = 6633
  c = RemoteController('c', ip=ip, port=port)
  net = Mininet(topo=topo, autoSetMacs=True, xterms=False, controller=None)
  net.addController(c)
  net.start()
  CLI(net)
  net.stop()
```

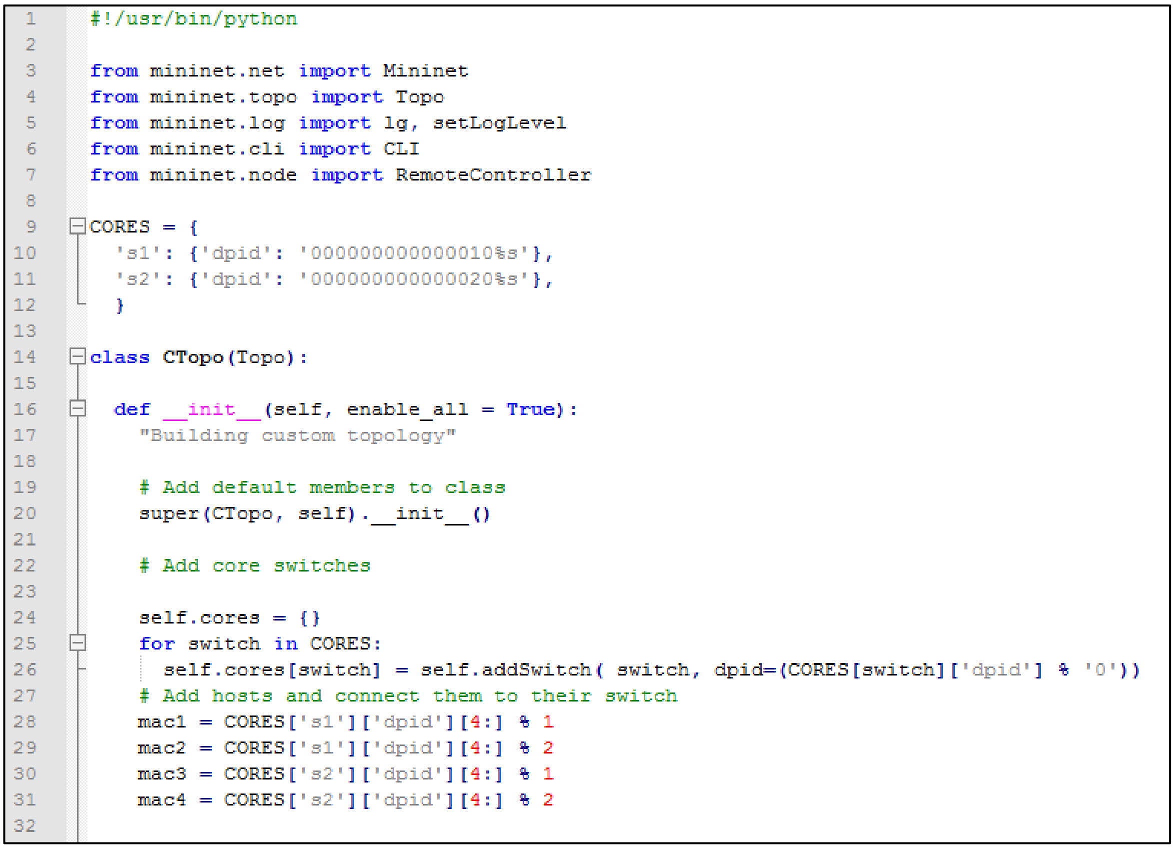
6.1.2. OpenVirteX script code



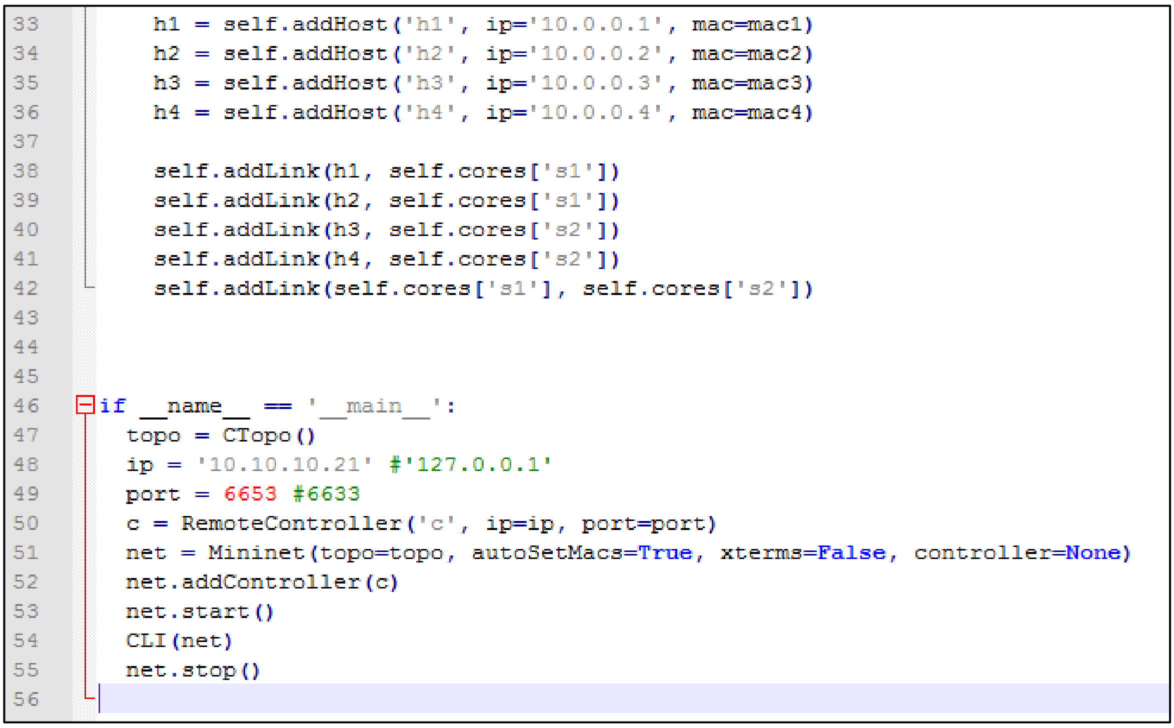
```
#!/bin/ ^^ echo # -*- ENCODING: UTF-8 -*-

cd /home/luis/OpenVirteX/utils
python ovxctl.py -n createNetwork tcp:10.10.10.21:6653 10.0.0.0 16

python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:01:00

python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 1
python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 2
python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 3

python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:01 1 00:00:00:00:00:01:01
python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:01 2 00:00:00:00:00:01:02
python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:01 3 00:00:00:00:00:01:03

python ovxctl.py -n startNetwork 1
```

6.2. Second Case of Study: 2 Switches, 4 Hosts (2sw-4hs)

6.2.1. Mininet Code

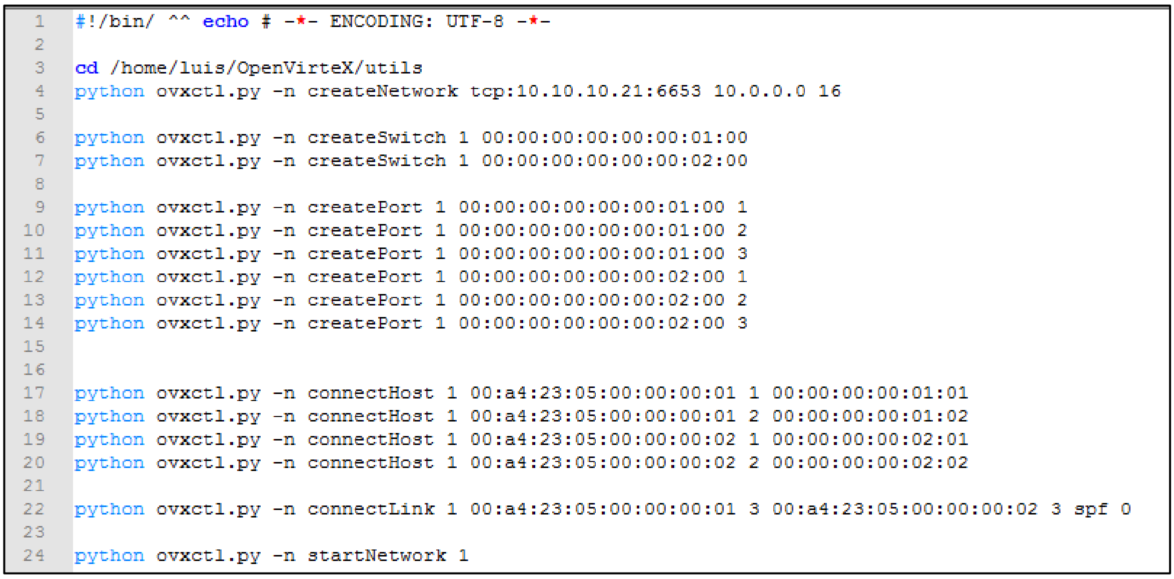


```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.topo import Topo
from mininet.log import lg, setLogLevel
from mininet.cli import CLI
from mininet.node import RemoteController

CORES = {
  's1': {'dpid': '000000000000010%s'},
  's2': {'dpid': '000000000000020%s'},
  }

class CTopo(Topo):

  def __init__(self, enable_all = True):
    "Building custom topology"

    # Add default members to class
    super(CTopo, self).__init__()

    # Add core switches

    self.cores = {}
    for switch in CORES:
      self.cores[switch] = self.addSwitch( switch, dpid=(CORES[switch]['dpid'] % '0'))
    # Add hosts and connect them to their switch
    mac1 = CORES['s1']['dpid'][4:] % 1
    mac2 = CORES['s1']['dpid'][4:] % 2
    mac3 = CORES['s2']['dpid'][4:] % 1
    mac4 = CORES['s2']['dpid'][4:] % 2

```



```
        h1 = self.addHost('h1', ip='10.0.0.1', mac=mac1)
        h2 = self.addHost('h2', ip='10.0.0.2', mac=mac2)
        h3 = self.addHost('h3', ip='10.0.0.3', mac=mac3)
        h4 = self.addHost('h4', ip='10.0.0.4', mac=mac4)

        self.addLink(h1, self.cores['s1'])
        self.addLink(h2, self.cores['s1'])
        self.addLink(h3, self.cores['s2'])
        self.addLink(h4, self.cores['s2'])
        self.addLink(self.cores['s1'], self.cores['s2'])



if __name__ == '__main__':
    topo = CTopo()
    ip = '10.10.10.21' #'127.0.0.1'
    port = 6653 #6633
    c = RemoteController('c', ip=ip, port=port)
    net = Mininet(topo=topo, autoSetMacs=True, xterms=False, controller=None)
    net.addController(c)
    net.start()
    CLI(net)
    net.stop()

```

6.2.2. OpenVirteX script code



```
#!/bin/ ^^ echo # -*- ENCODING: UTF-8 -*-

cd /home/luis/OpenVirteX/utils
python ovxctl.py -n createNetwork tcp:10.10.10.21:6653 10.0.0.0 16

python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:01:00
python ovxctl.py -n createSwitch 1 00:00:00:00:00:00:02:00

python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 1
python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 2
python ovxctl.py -n createPort 1 00:00:00:00:00:00:01:00 3
python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 1
python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 2
python ovxctl.py -n createPort 1 00:00:00:00:00:00:02:00 3


python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:01 1 00:00:00:00:01:01
python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:01 2 00:00:00:00:01:02
python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:02 1 00:00:00:00:02:01
python ovxctl.py -n connectHost 1 00:a4:23:05:00:00:00:02 2 00:00:00:00:02:02

python ovxctl.py -n connectLink 1 00:a4:23:05:00:00:00:01 3 00:a4:23:05:00:00:00:02 3 spf 0

python ovxctl.py -n startNetwork 1
```

Glossary

A list of all acronyms and the meaning they stand for.

DPID: Delivery Point Identifier

hs: in some parts or names is use to abbreviate ‘host’

ICMP: Internet Control Message Protocol

MAC: Medium Access Control

NBI: North Bound Interface

OF: OpenFlow

ONF: Open Networking Foundation

OVX: OpenVirteX

SDN: Software Defined Networking/Network(s)

sw: in some parts or names is use to abbreviate ‘switch’

VN: Virtual Network