Testing Software Defined Networks with OpenDaylight and Mininet

P. David Arjona Villicaña

Facultad de Ingeniería
Universidad Autónoma de San Luis Potosí

Version 1.2

August 2025

Preface to version 1.0

This document started as a guide for my students on how to build and test their own Software-Defined Networks (SDN). However, as I continued working in this topic, I realized there is a broader need to show how to start working and experimenting with this type of networks.

At the moment of writing this text, Mininet is the most used network emulator, and OpenDaylight is one of the most stable, supported and used SDN controllers freely available. The combination of these tools allows to experiment how these type of networks work, and learn how to configure them for specific scenarios. Hence, the main objective of this guide is to help engineers learn how to use these tools and promote the use of SDN in research and professional environments.

Although the activities in this document have been tested, the technology moves fast and it is probable that, as this document ages, some of the commands, programs and functionalities described here become obsolete. I will try to update this document as much as possible, but I cannot make promises that it will stay current forever. Such is the penalty of an evolving world!

P. David Arjona Villicaña
San Luis Potosi, Mexico
August 2024

Contents

1 Introduction **1**

1.1 Anatomy of the SDN network 1

1.2 Tools needed to implement an SDN lab 3

1.2.1 Mininet 3

1.2.2 OpenDaylight 4

1.2.3 VirtualBox 4

1.2.4 Docker 4

1.3 Practical considerations 5

2 The First Experiment **7**

2.1 About VirtualBox 7

2.2 Mininet 8

2.2.1 Mininet as a VM 9

2.2.2 Mininet as a Docker container 10

2.3 Installing OpenDaylight and Java 10

2.3.1 Configuring JAVA_HOME 11

2.4 OpenDayLight 12

2.4.1 Downloading ODL 12

2.4.2 First run 13

2.4.3 First test 13

2.4.4 Finishing the test 16

2.5 curlx application 17

3 Introduction to Docker **19**

3.1 Obtaining an image and creating a container 19

3.2 Managing, stoping and restarting a container 20

3.3 Other useful commands 20

4 Exploring ODL **23**
4.1 ODL NETCONF/RESTCONF 23

5 ODL Cluster Configuration and Use **27**
5.1 ODL cluster architecture 27
5.2 Cluster configuration with Docker 28
5.3 ODL cluster testing 30
5.4 Conclusions 32

6 ODL Manual Configuration **33**
6.1 The spine-leaf topology 33
6.2 Configuring the Address Resolution Protocol 34
6.3 Configuring the IP paths 35
6.4 Basic flow test 36
6.5 Advanced flow analysis 36
6.6 Conclusions 37

A JSON Example files **39**
A.1 Flood example 39
A.2 Transfer example 40
A.3 ARP example 41
A.4 Transfer example for regular IP packets 42
A.5 Flow based on source IP address 43
A.6 Flow based on destination IP address 44

B Mininet Example Files **45**
B.1 ODL cluster example 45
B.2 Spine-leaf example 46

Bibliography **50**

Chapter 1

Introduction

Network administrators are responsible for ensuring that their network's resources are used in an efficient manner and are always available for their organization's needs. There are many tasks that need to be correctly completed in order to guarantee this level of service, and users do not hesitate to complain when the efficiency of the network is not as expected. Therefore, Information Technology departments spend a considerable amount of time and resources making sure their network is in top condition.

Lately, a new technology has promised to ease and centralize network administration and configuration, called *Software Defined Networks* or SDN. This technology allows to separate the network elements' control plane from the data plane, which in turn allows faster management operations and increases the configuration flexibility for the network. This introductory Chapter explains what is SDN and how it works. It also provides a brief description of the different tools that will be used in the rest of this guide.

1.1 Anatomy of the SDN network

The main objective of an SDN network is to separate the operations that move packets around the network, from the administration and configuration tasks that the network needs to work efficiently. In turn, this requires a new network element, a controller, whose main task is to handle proper network configuration at the same time as it controls how packets travel around the network.

As shown in Figure 1.1, an SDN network architecture is divided in three levels. The *data plane* is at the bottom. Switches, routers and other network elements belong in this layer, which is responsible to move packets around

the network, following the configuration defined by the network administrator. As the name says, this is where the users' data is processed and moved.

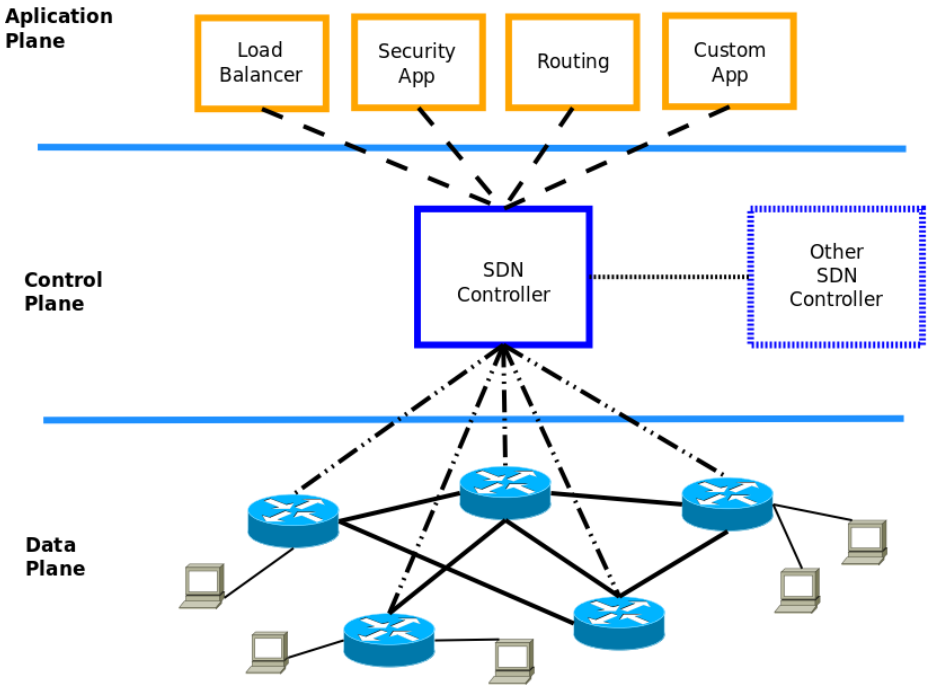

Figure 1.1: SDN network architecture

The *control plane* sits at the middle of the SDN structure. Here, SDN controllers are responsible for managing the network elements at the data layer to make sure the network behaves as intended. When a switch does not know how to handle a packet, it queries its controller for further instructions on how to process it. Controllers are not only responsible for monitoring and controlling the data plane, they also need to interact with the instructions received from the application layer.

The control plane delivers instructions to the data plane using a *southbound interface*, which is mostly the protocol that will be used to communicate the controller to the network switches. Currently, the most used southbound interface protocols are OpenFlow and NETCONF.

Originally, A single controller would be responsible to manage a complete network. However, this is not a good solution in terms of security, redundancy, and processing bottle necks. Therefore, the idea of creating

a cluster of controllers becomes necessary. This book takes this into consideration and will provide instructions about how to create and configure clusters using the OpenDaylight controller.

The *application plane* is the top layer of the SDN structure. This is where network administrators should be able to use and implement different applications that will be used to configure, monitor and manage the SDN network in a more flexible and efficient manner than it is possible with current network configuration tools. Currently, there is no predominant technology for this layer.

The easiest way to experience and test an SDN is to implement a laboratory using virtual systems and network emulators. This book will guide the reader to create a virtual laboratory that allows it to test the functionality offered by a network controller, OpenDaylight, and the Mininet network emulator.

1.2 Tools needed to implement an SDN lab

The following chapters in this book will use a variety of tools to implement a virtual SDN laboratory. This section offers a preview of the most important technologies that will be introduced in more detail later on.

1.2.1 Mininet

Mininet is a network emulator which was originally developed to test and experiment SDN networks. It is able to generate a network with individual hosts, switches, and a default controller, all of which may get connected between themselves using different topologies. Mininet also allows to connect its emulated network to an external controller, which then communicates with the switches using the OpenFlow protocol.

This text will use Mininet to implement the data plane layer of an SDN network. This means that the default controller will not be used and that connection to external controllers is required for all future activities.

Mininet has its own official page [1] which includes comprehensive documentation and examples on how to install and use this tool. There are also many pages in the web that provide different examples on how to employ Mininet to implement different types of networks and tests. Therefore, this document will not try to further explain the inner workings of this network emulator.

1.2.2 OpenDaylight

OpenDaylight (ODL) is an SDN controller supported by the Linux Foundation. It is an open source project and is programmed in Java. ODL has a development and support community, which allows communication with people interested in contributing to this project or in testing this controller in their own networks. One of the most desirable features of this controller is that it supports the implementation of clusters. A cluster is a group of at least three controllers that may be used to manage a network. Clusters provide redundancy and better availability for network administrators and they will be covered in Chapter 5.

For more information on ODL, consult its main [2] and documentation [3] pages.

1.2.3 VirtualBox

VirtualBox [4] is a computer virtualization tool developed by Oracle. It is freeware, which means it may be used without the need to buy a license. This tool allows to run almost any operating system in any computer as a virtual machine (VM). The VM works as if it were an independent computer and it has its own network interface and IP address. However, there are different ways to use and configure this interface.

For this text, the main advantage of a VM is that it allows to install and run software that was not designed to run on your native operating system or that has the potential to reconfigure important settings. Another important feature of VirtualBox, is that it allows to run more than one computer at the same time, in the same hardware.

1.2.4 Docker

Docker [5] is a container tool developed by Docker Inc. Just like VirtualBox, this tool is a freeware that allows to run Linux applications, just as if it were a VM. However, a Docker container is not a VM, as it uses the same kernel as a real Linux machine. This means that a container has some limitations when compared to a VM. On the other hand, they are very light on the host computer and thus, very fast.

In this text, Docker will be used similarly to VirtualBox, with the added advantage that multiple containers may run in the same computer without seriously impacting the overall performance of all these systems. An introduction on how to use Docker and its most common commands are provided at Chapter 3.

1.3 Practical considerations

In order to ease the implementation of the experiments described in here, there are some practical considerations about the hardware and systems the readers should use.

The first consideration is about the amount of memory needed. For those using a VM, it is necessary to reserve at least 4 GB of RAM memory for each virtualization. Some experiments, specially the ones in a cluster configuration, may need up to 8 GB of memory to run correctly. Also, having a multi-core processor is always recommended.

In terms of the system, Docker has had problems when installing in non-supported Linux versions. Therefore, it is best to use the platforms recommended at the Docker web site [5], which are Ubuntu, Debian, RHEL and Fedora.

Chapter 2

The First Experiment

This Chapter provides the instructions needed to install and test an SDN network using Mininet and OpenDaylight (ODL). Mininet will emulate a network with hosts and switches, and ODL will provide instructions to the switches, so they are able to control this network. This means that both of these tools need to be running in different systems. This chapter describes two different ways to implement such experiment, either by using virtual machines or Docker containers.

Section 2.1 describes how to configure VirtualBox, which is the tool used to implement an independent Linux system in your computer. While Section 2.2 explains the two options available to install Mininet and ODL. Section 2.3 describes the preliminary steps that are required for the system that will run ODL, it is important to follow the recommendations in this section. Then, Section 2.4 shows how to download, install and use ODL for the fist time, which is the main objective in this chapter. Finally, Section 2.5 briefly introduces curlx, which is a simple tool that allows better display of ODL output in JSON format.

2.1 About VirtualBox

This guide will employ virtual machines (VM) using Oracle's VirtualBox software. There are other virtualization programs available, but they will not be covered in here.

If you are using a Linux system, it may not be necessary to install VirtualBox. However, regardless of your native system (Linux, Windows or Mac) it is always recommended to run the experiments in this guide in a VM, in order to prevent conflicts with your native operating system.

The experiments in this guide will require different systems to communicate between themselves, even if they are running in the same computer. Therefore, it is recommended to use VirtualBox's Ethernet network adapter, instead of WiFi. Consider that VirtualBox supports Ethernet network adapters only when they are available in the host system, which means that if your machine does not have one, you will need to install it.

In order to support Ethernet communication between two VM, it is necessary that their adapters are configured as *Bridged Adapter*. This allows each VM to have its own IP address and communicate freely with other computers, real or virtual, in the network. Before performing this configuration, read Nakivo's guide on how to use different VM network adapters [6]. Also, it is recommended to enable the *promiscuous mode*. This funnily named setting instructs the network adapter to pass all received packets to the operating system, and it is needed when using a sniffer to analyze packets flowing through the network. These configurations are shown in Figure 2.1.

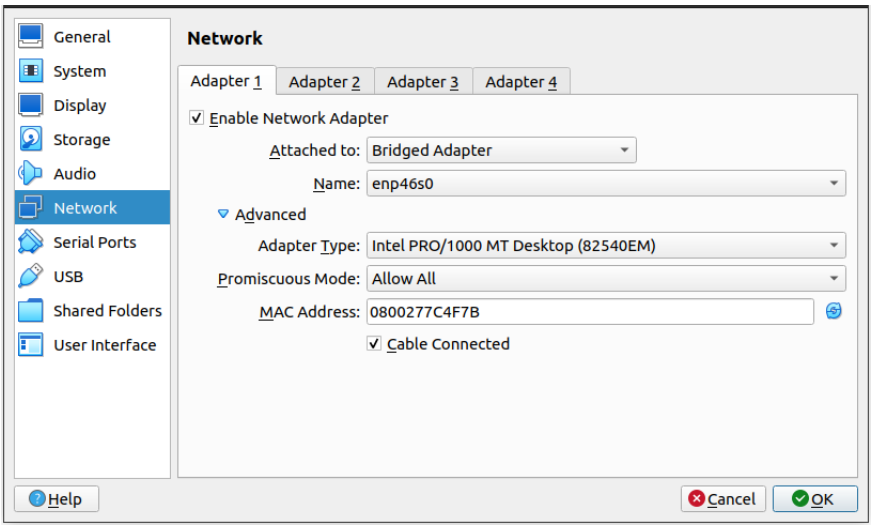

Figure 2.1: Recommended network adapter configuration

2.2 Mininet

Mininet is a very popular and well documented network emulator, and there are many references that show how to use this tool. This book recommends

to read and follow the instructions from the official web page [1]. Specifically, the *Get Started* and *Walkthrough* sections are good references on how to start working with this tool.

The following subsections describe two different ways to implement an SDN network using Mininet: Section 2.2.1 describes how to implement a network using two independent VMs, while Section 2.2.2 does the same but with Docker containers.

2.2.1 Mininet as a VM

The main advantage of installing Mininet and ODL in separate VMs is that it is possible to employ the hardware resources of two different computers to implement experiments. The computers will exchange information using the network. Figure 2.2 shows this setup.

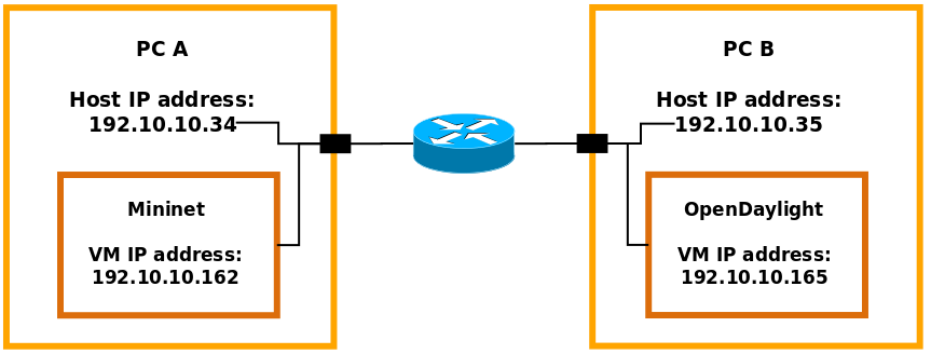

Figure 2.2: Setup using separate VMs in two different host systems

The easiest way to obtain Mininet in a VM is to download one of the images provided at this tool's web site [1]. Another option is to install Mininet from source packages. The *Get Started* section of this same web page has information on how to complete either of these solutions.

When using this configuration, remember that the VM's IP address is usually displayed at start up, or it may be obtained from the *terminal* by using either the *ip addr* or *ifconfig* commands.

Consider that Mininet's VM does not include a graphical user interface (GUI), fortunately you may use more than one *ssh* session to better interact with this VM.

2.2.2 Mininet as a Docker container

The main advantage of installing Mininet and ODL in two Docker containers is that all the experiments can run in a single machine. The computers will exchange information using Docker's in-built network. Figure 2.3 shows this setup.

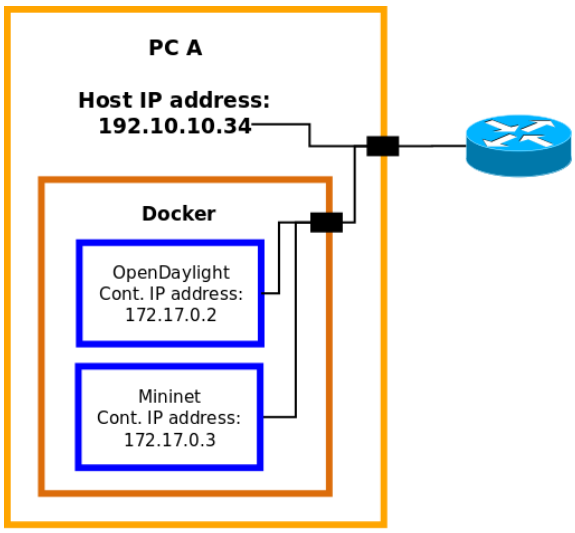

Figure 2.3: Setup using Docker containers in a single host

Before trying to obtain Mininet in a container, read Chapter 3 for a brief introduction on how to use Docker. Then, download and install the Mininet image at the Docker hub page [7]. In order to obtain a container's IP address, use the commands included at Section 3.3

2.3 Installing OpenDaylight and Java

Since there is no official ODL VM or container, the first step is to create the system where ODL needs to run. Fortunately, this is not a difficult task. If the VM option was selected (Section 2.2.1), install a new Linux VM running the latest Ubuntu or Fedora distribution. If the Docker option was chosen (Section 2.2.2), download an official image for either Ubuntu or Fedora from the Docker hub page [7], and then create a container.

ODL runs over Apache Karaf [8], which is an application container that provides support for terminal commands and for running other applications.

Karaf is included in the ODL running file. Therefore, it is not individually installed.

The second step is to install an appropriate Java version in the system that will run ODL. At the moment of writing this document, Scandium release and newer require Java 21, while Calcium release and previous employ Java 17. Make sure to install the correct Java version for the ODL release you are planning to use. If you need to support different Java versions in the same system, the following command may be used to switch between them:

```
$> sudo update-alternatives --config java
[sudo] password for myuser1:
There are 2 choices for the alternative java (providing
/usr/bin/java).

Selection    Path                                Priority   Status
------------------------------------------------------------
  0    /usr/lib/jvm/java-17-oracle/bin/java  1091   auto mode
  1    /usr/lib/jvm/java-11-oracle/bin/java  1091   manual mode
* 2    /usr/lib/jvm/java-17-oracle/bin/java  1091   manual mode

Press <enter> to keep the current choice[*], or type selection
number: 1
update-alternatives: using /usr/lib/jvm/java-11-oracle/bin/java
to provide /usr/bin/java (java) in manual mode
```

Once Java has been installed, the following two commands may be used to verify the Java version that the system is currently using and its location, which is usually */usr/bin*:

```
$> java --version
java 11.0.13 2021-10-19 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.13+10-LTS-370)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.13+10-LTS-
370, mixed mode)
$> which java
/usr/bin/java
```

2.3.1 Configuring JAVA_HOME

ODL needs a JAVA_HOME system variable to store Java's location. There are two ways to define this variable. The first and most immediate way is to simply type the following command at the terminal:

```
$> export JAVA_HOME=/usr
```

Notice that the */bin* part of the path has been omitted. The only problem with this method is that after loging out of this session, the variable gets lost. In order to make it permanent, it is necessary to add the previous command, as a single line, at the end of the *.bashrc* file, which is always in the user's home directory.

To verify that this variable has been declared correctly, simply type:

```
$> echo $JAVA_HOME
```

2.4 OpenDayLight

There is more than one way to install ODL. Since this chapter is oriented to configure an easy-to-use lab environment, ODL will be installed from a *tar.gz* file. Although installing and running is not very difficult, configuration and testing may require more work than expected.

It is important to consider that if the VM option was selected (Section 2.2.1), ODL will most probably run from a user's account and home directory; whereas for the Docker option (Section 2.2.2), ODL will run from the root account, which has */root/* as its default home directory. The differences between these two options are described in the following subsection.

2.4.1 Downloading ODL

At the ODL's documentation page [3], select the *Downloads* section. Copy the link for the latest Tar release. At the moment of writing this text, it was 20.0.0 (Calcium version).

If you are running ODL in a VM, the easiest way to download this file is opening a terminal at the system where it needs to be installed, and use the *wget* command and the copied link to download the *tar.gz* file:

```
$> wget [Paste link from the ODL docs page]
```

If you are running ODL in a container, download the *tar.gz* file to the host system using which ever method you prefer. Then, copy the file to the Docker container using an instruction similar to the following:

```
$> sudo docker cp karaf-0.20.0.tar.gz  odl-1:/root/
```

Once the *tar.gz* file has been placed in the system and directory where it needs to be installed, uncompress the file using the following command:

```
$> tar xzf [Doumloaded file]
```

2.4.2 First run

To start running ODL, just move to the directory where it has been uncompressed and run the following command:

```
$> cd [ODL directory]
$> ./bin/karaf
```

It usually takes more than 30 seconds for karaf to start and return a command prompt, which is the command line interface (CLI) that may be used to interact with ODL. The following command may be used to verify which applications are already installed in ODL:

```
odl> feature:list -i
```

If this is the first time executing ODL, it is necessary to install the *OpenFlowPlugin* features recommended for users at the OpenFlowPlugin documentation. Although it should be possible to use the *feature:install* command for each feature that needs to be installed, in practice this has produced configuration errors, therefore it is best to run the following command as a single-line instruction:

```
odl> feature:install odl-openflowplugin-flow-services-rest
     odl-openflowplugin-app-table-miss-enforcer
     odl-openflowplugin-nxm-extensions
```

The *logout* command may be used to exit ODL.

2.4.3 First test

To perform this test, start ODL and run the following command to verify that there are no connections active:

```
odl> ofp:show-session-stats
```

Then, use the following Mininet command to create the network shown at Figure 2.4. This network has a simple tree topology with 3 switches and 4 hosts.

```
$> sudo  mn  --controller=remote,ip=[IP addres for ODL],port=6653
   --topo  tree,2  --switch  ovsk,protocols=OpenFlow13
```

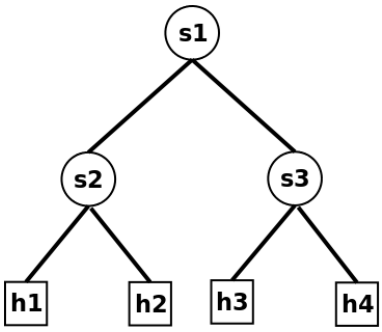

Figure 2.4: Network defined by the *mn* (Mininet) command

Run again the *ofp:show-session-stats* command an verify that ODL is now connected to the three openflow switches. At this point ODL, is connected to the network defined by Mininet, however the switches do not have instructions on how to process the packets that need to traverse the network. This can be verified when the *pingall* command fails at the Mininet terminal:

```
mininet> pingall
```

In order to provide instructions for the switches so they know how to route packets through the network, it is necessary to define *flows* at the ODL controller, which then will be passed to the switches as instructions. ODL defines flows using the JSON format, which can be uploaded as individual files. To upload this files to ODL, this guide employs the *curl* command, however there are many other API testing applications that allow easy editing of outgoing commands and better visualization of incoming JSON messages. One of such tools is *curlx*, which will be further described at Section 2.5.

For the current experiment, it is necessary to define flows that instruct switches $s2$ and $s3$ to flood incoming packets in every port to all their other ports. For switch $s1$ it is necessary to define a flow that transfers packets received at port 1 to port 2, and another flow that transfers packets in the opposite direction, from port 2 to port 1. This configuration is shown at Figure 2.5.

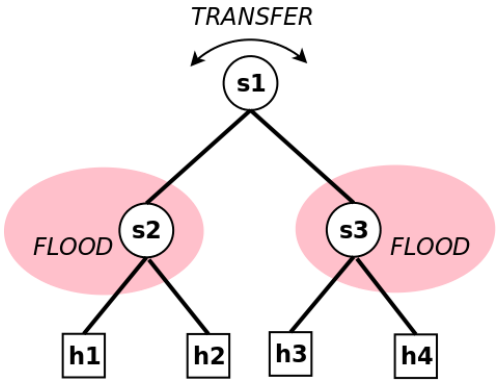

Figure 2.5: Flows that need to be configured

Appendix A.1 shows a JSON file example with instructions for packets received at port 1 of switch $s2$ (openflow:2:1). The instructions say that such packets need to be retransmitted (FLOOD) to the other ports in this switch. This file should be uploaded to ODL using the following instruction:

```
curl -u admin:admin -X PUT -d "@[File with JSON instructions]"
-H "Content-Type: application/json" http://[IP address for ODL]
:8181/rests/data/opendaylight-inventory:nodes/node=openflow:2
/flow-node-inventory:table=0/flow=201
```

Notice that the *id* at the file in Appendix A.1 and the *flow* parameter at the previous instruction coincide. The same is true for the *table_id* and *table* parameters. Also, both use the same switch identifier, *openflow:2*. Consider that as this flow only includes port 1 of switch $s2$, other similar configuration files need to be provided for ports 2 and 3, and for all the 3 ports of switch $s3$.

Two notes of caution about the *curl* command: The *at* symbol (`@`) needs to precede the JSON filename, for example: `"@workdir/myfile.json"` is a valid instruction. Also, the *curl* command does not return any message when successful. To receive a confirmation it is neccesary to add the *-v* option when executing this command.

Appendix A.2 shows a JSON file example with instructions for packets received at port 2 of switch $s1$ (openflow:1:2). The instructions say that such packets need to be sent to port 1 of this same switch. The command for uploading this file is:

```
curl -u admin:admin -X PUT -d "@[File with JSON instructions]"
-H "Content-Type: application/json" http://[IP address for ODL]
:8181/rests/data/opendaylight-inventory:nodes/node=openflow:1
/flow-node-inventory:table=0/flow=102
```

Another similar file is needed for transferring packets from port 1 to port 2 at switch $s1$.

After all the flows have been uploaded, use the following command to verify that ODL correctly records them for each switch. The following example is meant to work for switch $s1$ (openflow:1), at table 0:

```
curl -u admin:admin -X GET http://[IP address for ODL]:8181
/rests/data/opendaylight-inventory:nodes/node=openflow:1
/flow-node-inventory:table=0
```

If everything looks correct, verify that all hosts can ping to each other using the *pingall* Mininet command.

2.4.4 Finishing the test

To finish the test, first use Mininet's *exit* command to stop the network. Then it is important to run the following command, which clears all Mininet variables and allows this tool to run correctly the next time you start it:

```
$> sudo mn -c
```

Now stop ODL by typing *logout* at this tool's command prompt.

ODL remembers the flows that were last uploaded. Therefore, the next time ODL starts, it will use the flows defined in the previous subsection. For the experiments in the following Chapters you may need to upload new flows which will conflict with the current ones, hence it is recommended to use the following command to delete flows that are not longer needed.

```
curl -u admin:admin -X DELETE http://[IP address for ODL]:8181
/rests/data/opendaylight-inventory:nodes/node=openflow:1
/flow-node-inventory:table=0/flow=101
```

2.5 curlx application

The curlx application is an extension to the curl command. One of its most useful features is that it automatically formats JSON output in a readable format. Therefore, this guide recommends to use curlx when the GET option is used to retrieve information from the ODL controller.

For more information on how to download, install and use curlx, refer to its official web page [9].

Chapter 3

Introduction to Docker

This Chapter lists and describes some of the most useful Docker commands that will be used in this guide to implement SDN experiments. For more detailed information on how to install and use Docker, refer to its official web page [5].

3.1 Obtaining an image and creating a container

The first step to use Docker is to obtain an *image*, which is the file that will be used to create a container. The Docker hub website [7] contains a large number of images that can easily get downloaded into your system. To donwload an image, use the following command:

```
$> sudo docker pull [image name]
```

To display all the images currently stored in your system use the command:

```
$> sudo docker images
```

In order to delete an image from the system, use the following command:

```
$> sudo docker image rm [image ID]
```

The next step is to create and start a container using the *run* command. This command has many different options. A very basic form of this command is:

```
$> sudo docker run -t -d --name [container name] [image name]
```

The *-d* option allows the container to run as a background process, and the *-t* option requests for a TTY terminal. For example, the command *sudo docker run -t -d --name sys-1 ubuntu:22.04* creates and runs a container called sys-1 using the ubuntu:22.04 image file. There are many other options that allow the user to define network configurations, display modes, system resources the container is allowed to use and other settings.

3.2 Managing, stoping and restarting a container

Once you have a container up and running, it is possible to verify its status using the following command:

```
$> sudo docker container ls
```

If you want to see all containers, even the ones that are not currently running, use the *-a* option:

```
$> sudo docker container ls -a
```

The following command allows to see the ammount of memory and cpu your running containers are using:

```
$> sudo docker stats
```

If you want to turn off one or more containers:

```
$> sudo docker container stop [list of containers to stop]
```

Finally, to restart a stopped container use:

```
$> sudo docker restart [container name]
```

3.3 Other useful commands

The following command displays the properties of a container:

```
$> sudo docker inspect [container name]
```

If you need to find the IP address of a running container use the following command:

```
$> sudo docker inspect [container name] | grep -i ipaddress
```

To upload a file from your local system to a container use:

```
$> sudo docker cp [local_file container_name]:[container_path]
```

For example, the command *sudo docker cp conf.txt sys-1:/sbin/* uploads a copy of the *conf.txt* file to the *sys-1* container, at the *sbin* directory.

The following command starts a regular terminal for a container that is currently running:

```
$> sudo docker exec -it [container name] bash
```

Chapter 4

Exploring ODL

This Chapter explores some of the functionality and commands that may become useful when experimenting with OpenDaylight (ODL).

4.1 ODL NETCONF/RESTCONF

ODL supports the NETCONF and RESTCONF protocols to display, modify, and delete its configuration and settings. More information about the NETCONF and RESTCONF protocols may be found at RFCs 6241 [10] and 8040 [11] respectively. Particularly, the RESTCONF protocol allows Web applications to access the configuration data originally developed for NETCONF using the HTTP protocol.

An easy way to start interacting with RESTCONF is using ODL's odl-restconf-openapi feautre, which may be installed using the following command:

```
odl> feature:install odl-restconf-openapi
```

Then, it is possible to access ODL's RESTCONF API page using a Web browser and the following address:

```
http://[ODL IP address]:8181/openapi/explorer/index.html
```

When the page requests a username and password, use the default values: *admin/admin*. As shown in Figure 4.1, the RESTCONF API page shows the ODL modules that are currently installed.

By clicking on a module it is possible to consult the different RESTCONF commands available. Figure 4.2 shows some of the commands previously

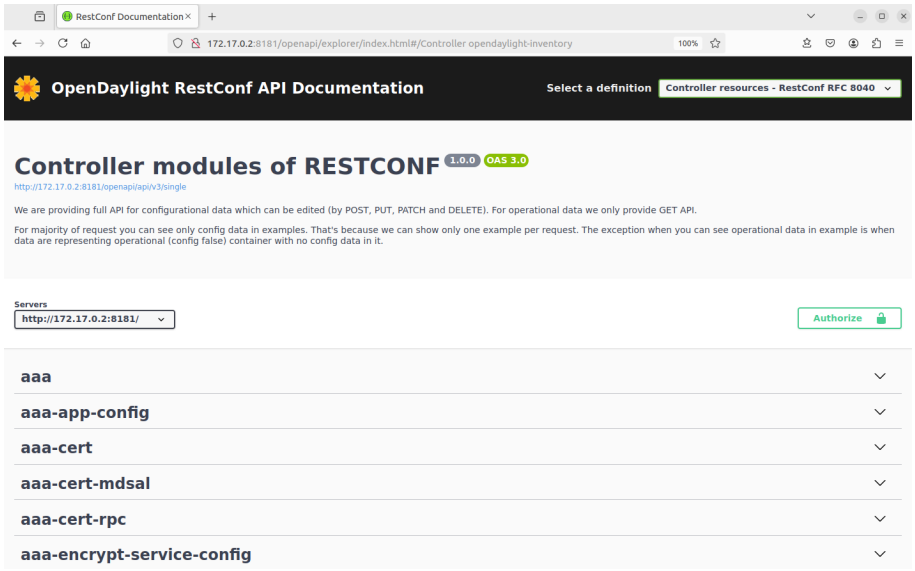

Figure 4.1: OpenDaylight RESTCONF API page

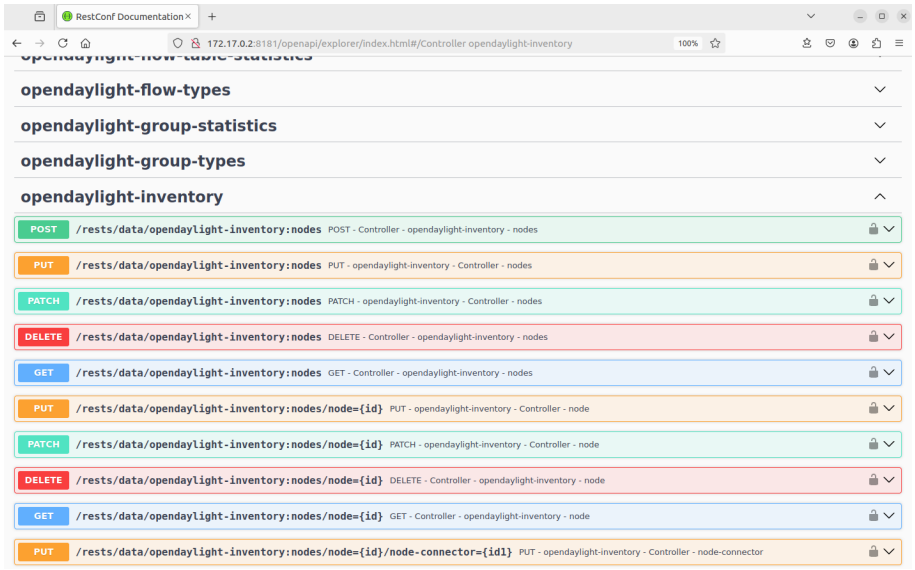

Figure 4.2: opendaylight-inventory module's commands

used at Section 2.4.3. There are different types of commands: the GET command allows to consult data from the controller, PUT and POST allow to upload data, and DELETE allows for the removal of data.

There are two different ways to execute the commands listed in the RESTCONF API page: The first one is simply using the same RESTCONF API page. To do this, select a command and click the *Try it out* button. Then fill the information requested by that particular command and click on the *Execute* button. The page should display the result of executing the command.

The second method is to simply use an external tool, like *curl* or *cx*, as it has been previously done at Section 2.4.3. Remember that it is necessary to include the username and password when using this method. The following is an example for the GET command for */rests/data/opendaylight-inventory:nodes/node={id}*.

```
curl -u admin:admin -X 'GET' \
  'http://172.17.0.2:8181/rests/data/opendaylight-inventory:
  nodes/node=openflow:2?content=config' \
  -H 'accept: application/xml'
```

Chapter 5

ODL Cluster Configuration and Use

OpenDaylight (ODL) allows for multiple controllers to manage the operation of a network at the same time, in order to increase its redundancy and resiliency. This is called a cluster configuration. This Chapter describes how to perform such configuration using the ODL controller.

5.1 ODL cluster architecture

Information about ODL cluster configuration may be found at the OpenDaylight documentation web page [3]. These pages recommend to define clusters with at least three computers, since the algorithm sometimes needs a majority to take decisions. For the specific case of ODL clusters, each of these computers is called a *member*. Also, the information is contained in a shard, which is a defined memory set that can be shared between the cluster's members.

The basic ODL cluster example shown at Figure 5.1 uses three Docker containers to simulate the three required member controllers. ODL cluster configuration is an involved process and care is needed at the moment of editing the configuration files.

The experiment described in this Chapter needs to run three Docker instances and a network, which will be implemented in Mininet. Each Docker instance will have a different IP address, which is usually in the 172.17.0.0/24 CIDR subnetwork and is reachable from the host computer. These addresses are important because these processes need to communicate with each other and the network.

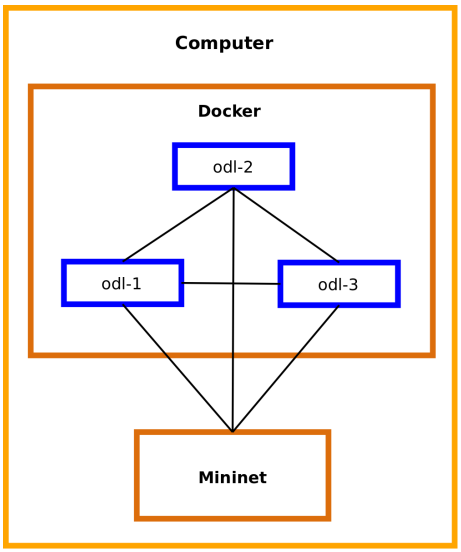

Figure 5.1: Basic OpenDaylight cluster example

In practice, this experiment usually fails when the host system has less than 4 GB of RAM memory, and at least 8 GB of memory are recommended for decent performance.

5.2 Cluster configuration with Docker

The following procedure describes how to configure an ODL cluster using three Docker containers, one for each cluster member:

1. The following instruction downloads the most recent Ubuntu image, which may be found at the Docker Hub page [7].

```
$> sudo docker pull ubuntu:22.04
```

2. Start running the Ubuntu image as three different Docker instances: odl-1, odl-2 and odl-3.

```
$> sudo docker run -t -d --name odl-1 ubuntu:22.04
$> sudo docker run -t -d --name odl-2 ubuntu:22.04
$> sudo docker run -t -d --name odl-3 ubuntu:22.04
```

3. Download ODL's latest Tar release (2.4) and copy this file into each Docker instance.

```
$> sudo docker cp -q [Downloaded file] odl-1:/root/
$> sudo docker cp -q [Downloaded file] odl-2:/root/
$> sudo docker cp -q [Downloaded file] odl-3:/root/
```

4. Verify the IP address for each ODL instance.

```
$> sudo docker inspect odl-1 | grep -i ipaddress
$> sudo docker inspect odl-2 | grep -i ipaddress
$> sudo docker inspect odl-3 | grep -i ipaddress
```

5. Log into each Docker instance by running the following commands in different terminals.

```
$> sudo docker exec -it odl-1 bash
$> sudo docker exec -it odl-2 bash
$> sudo docker exec -it odl-3 bash
```

   The following steps apply for each Docker terminal started above.

6. Run the *apt update* and *apt upgrade* commands at each Docker instance.

7. Follow the instructions at Section 2.3 to install Java at each Docker instance. Do not forget to define the JAVA_HOME system variable.

8. Uncompress the Tar file, run ODL for the first time and install the features listed bellow. Notice that, with the exception of the last one (odl-mdsal-distributed-datastore), these features are the same as in Section 2.4.2.

```
odl> feature:install odl-openflowplugin-flow-services-rest
     odl-openflowplugin-app-table-miss-enforcer
     odl-openflowplugin-nxm-extensions
     odl-mdsal-distributed-datastore
```

9. Logout of ODL and start it again to apply changes.

10. Download clustering information from each Docker instance to verify ODL is working correctly by using the following two commands:

```
curl -u admin:admin -X GET http://[Docker instance IP address]
:8181/jolokia/read/org.opendaylight.controller:type=
DistributedConfigDatastore,Category=ShardManager,
name=shard-manager-config
```

```
curl -u admin:admin -X GET http://[Docker instance IP address]
:8181/jolokia/read/org.opendaylight.controller:type=
DistributedOperationalDatastore,Category=ShardManager,
name=shard-manager-operational
```

11. Logout of ODL and follow the instructions at section *Setting Up a Multiple Node Cluster* at ODL's Setting Up Clustering page [12]. Pay special attention when modifying files *pekko.conf* and *module-shards.conf*, since they both need to have the correct IP address for each of the cluster's members. Currently, ODL is migrating from Akka to Pekko, therefore, it is possible that your ODL version still uses file *akka.conf* instead of *pekko.conf*. Both files are similar and need to be modified similarly.

12. Start ODL again to apply changes.

13. Verify that files *pekko.conf* and *module-shards.conf* have been correctly modified by downloading clustering information again (step 10).

This concludes the cluster configuration for ODL. The next step is to test that the cluster is working. This is described at the following section.

5.3 ODL cluster testing

The following procedure describes how to test the basic ODL cluster configured at Section 5.2. Therefore, the cluster needs to be running before performing the following steps. This experiment uses Mininet to build a small network with three switches and four hosts (Figure 5.2), and then connects this switches to the three cluster's controllers.

1. Start mininet by running the Python program at Appendix B.1.

```
$> sudo python3 [mininet file with py extension]
```

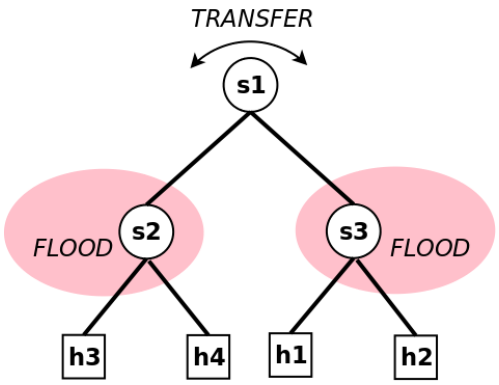

Figure 5.2: Network and flows to be tested

2. Run the *pingall* command and verify that, since no flows have been uploaded, the pings do not work.

3. Review Section 2.4.3 and use *curl* to upload the same flows used there into one of the controllers in the cluster.

4. Run the *pingall* command again and verify that the pings now complete successfully.

5. Use the *curlx* application (Section 2.5) to verify that the uploaded flows have been transfered to the other controllers in the cluster.

6. The following commands may be used to verify the cluster configuration at each Docker instance:

```
curl -u admin:admin -X GET http://[Docker instance IP address]
:8181/jolokia/read/org.opendaylight.controller:Category=
Shards,name=member-[instance number]-shard-default-
operational,type=DistributedOperationalDatastore
```

```
curl -u admin:admin -X GET http://[Docker instance IP address]
:8181/jolokia/read/org.opendaylight.controller:Category=
Shards,name=name=member-[instance number]-shard-topology-
operational,type=DistributedOperationalDatastore
```

```
curl -u admin:admin -X GET http://[Docker instance IP address]
:8181/jolokia/read/org.opendaylight.controller:Category=
```

```
Shards,name=name=member-[instance number]-shard-inventory-
operational,type=DistributedOperationalDatastore
```

5.4 Conclusions

This Chapter has described how to use Docker containers and Mininet to test ODL controller clusters. This is a more complex configuration than the one explored in Chapter 2, with the added benefit that clusters increase the resilience and availability of an SDN network.

Is is very important to remember that **controllers in a cluster configuration will not work properly if they are run as a single controller**. In order to revert to a single configuration, the best solution is to create a new ODL controller that is not configured for clustering.

Chapter 6

ODL Manual Configuration

This Chapter implements a simple experiment that shows how to configure ODL to route packets through a spine-leaf network. This topology is used to distribute traffic in data-centers and provides route diversity, which is used to define data flows. The setup in the following sections has a more customized approach, which means that it is necessary to configure the protocols needed to facilitate node communication.

The first protocol that needs to be configured is the Address Resolution Protocol (ARP), which allows to link the hosts' IP addresses with their corresponding MAC addresses. The configuration needed for this protocol to work is described at Section 6.2. Then it is necessary to configure the different flows that will be used to transfer IP packets. The steps needed to establish this configuration is provided at Section 6.3.

6.1 The spine-leaf topology

A spine-leaf topology includes two different type of switches: the *spine* switches, which connect to each of the leaf switches; and the *leaf* switches, which also connect to the network hosts. Figure 6.1 is an example of the simplest spine-leaf topology possible, which only has two spine switches, in pink, and two leaf switches, in green. In this example, each leaf switch has two hosts that will be used to define different packet flows between them.

The switches in this Chapter will be configured to support the following four packet flows:

1. Packets originated at $h1$ will travel through switch $s1$.
2. Packets originated at $h2$ will travel through switch $s2$.

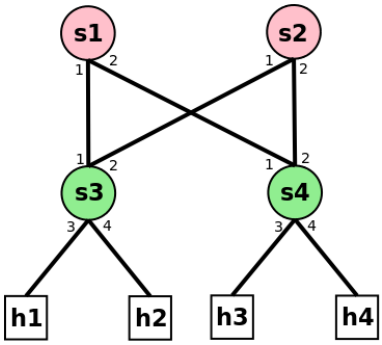

Figure 6.1: Spine-leaf topology (Note: smaller numbers represent switch's port numbers)

3. Packets originated at $h3$ will travel through switch $s1$.

4. Packets originated at $h4$ will travel through switch $s2$.

This means that switches $s3$ and $s4$ need to distribute traffic according to the host's source IP address, while switches $s1$ and $s2$ only need to transfer traffic from one interface to the other, similarly to what was configured in Section 2.4.3.

In this example, all hosts belong to the same local area network (LAN). Therefore, the ARP protocol is needed to allow Ethernet to link each host's IP address to its corresponding MAC address. This is further described in the following section.

6.2 Configuring the Address Resolution Protocol

To configure the Address Resolution Protocol (ARP) it is necessary to create a flow that asks the switch to behave as a regular switch when an ARP packet is received. The normal behavior of this protocol is to broadcast to all the hosts in the network asking for the owner of the desired IP address. The broadcast MAC address is `ff:ff:ff:ff:ff:ff`. When the owner of the requested IP address receives the message, it answers with its own MAC address. This means that the easiest way to support this protocol is to let the switch propagate the ARP packet to all its ports. ODL labels this as the NORMAL behavior.

In order to identify all ARP packets, Ethernet sets its *Type* field to the hexadecimal value 0806, which is equivalent to the 2054 decimal value. An

example of a flow instruction for handling this type of packets may be found at Appendix A.3. Notice that besides Type, **no other matching criteria is used**. This means that all ARP packets will be processed using the normal behavior.

Since the ARP protocol NORMAL behavior is to broadcast packets to all its ports, loops and infinite cycles will be generated if all switches in the spine-leaf topology send this type of packets. To avoid this problem, all the leaf switches, but only one of the spine switches need to implement these ARP flows. This guarantees that ARP packets are broadcasted following a tree topology.

6.3 Configuring the IP paths

For this example, spine switches paths are simple: they just need to transfer IP packets received at port 1 to port 2, and IP packets received at port 2 to port 1. This is similar to the configuration used at Section 2.4.3. However, these flows need to consider that they should only apply to regular IP packets, which use Ethernet Type 2048. This is so they are processed independently from the ARP packets and flows defined in the previous section. An example of this transfer flow is included at Appendix A.4, and switches $s1$ and $s2$ need to configure two transfer flows each.

Leaf switches have a more complex configuration. For example, the flows that $s3$ needs to support are:

1. Packets originated at IP address 10.0.0.1 ($h1$) will be sent to port 1.
2. Packets originated at IP address 10.0.0.2 ($h2$) will be sent to port 2.
3. Packets finishing at IP address 10.0.0.1 ($h1$) will be sent to port 3.
4. Packets finishing at IP address 10.0.0.2 ($h2$) will be sent to port 4.

Appendix A.5 is an example of the configuration file needed for the first flow, while Appendix A.6 is an example for the third flow. Notice that the third flow needs to have higher priority (9) than the first one (8). This is because if $h1$ sends a packet to $h2$, $s3$ needs to send this packet to port 3, instead of port 1. For any other case, packets from $h1$ should be sent to port 1.

The flows at Appendix A.5 and A.6 also filter for Ethernet Type 2048, which is equivalent to the 0800 hexadecimal value and corresponds to regular

IP packets. This means that flows defined for the ARP protocol at Section 6.2, which use Type 2054, are processed independently from these IP flows.

Similar flows to the four described above, need to be edited and uploaded to $s4$, but for hosts $h3$ (ip address 10.0.0.3) and $h4$ (10.0.0.4). Once the flows for the four switches have been configured, it is necessary to test thay they are working as intended. This is described in the following section.

6.4 Basic flow test

The first step to testing the flows implemented in the previous sections, is to use a Python program that asks Mininet to build the network shown in Figure 6.1. An example of such program is provided at Appendix B.2. To start this network, run the Python program using the following command:

```
$> sudo python3 [mininet file with py extension]
```

Once the network has been created, run the *pingall* command to verify that all packets can reach its destination.

6.5 Advanced flow analysis

Although the previous test demonstrates that the network is functional, it does not demonstrate that the flows follow the rules defined at Section 6.1. To verify this, it is necessary to use the *ovs-ofctl* command and the following procedure:

1. Open a terminal from the same system that is running Mininet and type the following commands to display the flows currently defined in all the network's switches:

   ```
   $> sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s1
   $> sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s2
   $> sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s3
   $> sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s4
   ```

2. Use a table similar to the one displayed at Table 6.1 to record the number of packets initially counted at each flow in each switch.

3. From the Mininet terminal, run a ping between two hosts, and let it ping for at least 10 times; then kill the ping using *Ctrl+c*. An example of how to run a *ping* between $h1$ and $h2$ is:

```
mininet> h1 ping h2
```

4. Display again the flows in all the network's switches and verify that the packet count for the flows that should have been used by the *ping* command have increased by the same number of pings that have happened at step 3. Record this new number at your table.

5. Repeat steps 3 and 4 until you have demonstrated that all the flows work as defined at Section 6.1.

Table 6.1: Example of a packet count record table

Switch	Flow	Test 1	Test 2	Test 3	...
$s1$	101	22	22	32	...
	102	15	25	25	...
	ARP	23	24	24	...
$s2$	201	9	19	19	...
	202	11	11	21	...
	ARP	19	19	19	...
$s3$	301	22	22	32	...
	302	18	28	28	...
	311	13	14	14	...
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

6.6 Conclusions

This Chapter has demonstrated how to define SDN flows based on the packet's IP addresses, and how to employ this to control the traffic in a network. The spine-leaf topology used in this example is very simple, but the reader should be able to extend it to larger and more complex topologies.

Appendix A

JSON Example files

A.1 Flood example

The following instructions are used to flood all packets received at port 1 of switch openflow:2, to all ports of this same switch. This flow's identifier is 201 and is stored at table 0:

```
{
   "flow": [
      {
         "table_id": 0,
         "id": "201",
         "priority": 4,
         "cookie": "4",
         "match": {
             "in-port": "openflow:2:1"
         },
         "instructions": {
            "instruction": [
               {
                  "order": 0,
                  "apply-actions": {
                     "action": [
                        {
                           "order": 0,
                           "output-action": {
                              "output-node-connector": "FLOOD"
                           }
                        }
                     ]
                  }
               }
```

```
            ]
         }
      }
   ]
}
```

A.2 Transfer example

The following instructions are used to transfer a packet that has arrived at port 2 of switch openflow:1, to port 1 of this same switch. This flow's identifier is 102 and is stored at table 0:

```
{
   "flow": [
      {
         "table_id": 0,
         "id": "102",
         "priority": 4,
         "cookie": "4",
         "match": {
            "in-port": "openflow:1:2"
         },
         "instructions": {
            "instruction": [
               {
                  "order": 0,
                  "apply-actions": {
                     "action": [
                        {
                           "order": 0,
                           "output-action": {
                               "output-node-connector": "1"
                           }
                        }
                     ]
                  }
               }
            ]
         }
      }
   ]
}
```

A.3 ARP example

The following instructions match all packets received with an Ethernet type equal to 2054 (decimal), or 0806 hexadecimal, to be treated as NORMAL packets in the switch. The 2054 value is exclusive for ARP packets, and the normal behavior is to broadcast the packet in all ports. This flow's identifier is simply ARP and is stored at table 0:

```
{
    "flow": [
        {
            "table_id": 0,
            "id": "ARP",
            "priority": 4,
            "cookie": "4",
            "match": {
                "ethernet-match": {
                    "ethernet-type": {
                        "type": 2054
                    }
                }
            },
            "instructions": {
                "instruction": [
                  {
                    "order": 0,
                    "apply-actions": {
                      "action": [
                        {
                          "order": 0,
                          "output-action": {
                             "output-node-connector": "NORMAL"
                          }
                        }
                      ]
                    }
                  }
                ]
            }
        }
    ]
}
```

A.4 Transfer example for regular IP packets

The following instructions are used to transfer a packet that has arrived at port 2 of switch openflow:1, to port 1 of this same switch. Notice that this instruction only applies to regular IP packets, which have an Ethernet type equal to 2048 (decimal), or 0800 hexadecimal:

```
{
   "flow": [
      {
         "table_id": 0,
         "id": "102",
         "priority": 4,
         "cookie": "4",
         "match": {
            "in-port": "openflow:1:2"
            "ethernet-match": {
                "ethernet-type": {
                    "type": 2048
                }
            }
         },
         "instructions": {
            "instruction": [
               {
                  "order": 0,
                  "apply-actions": {
                     "action": [
                        {
                           "order": 0,
                           "output-action": {
                               "output-node-connector": "1"
                           }
                        }
                     ]
                  }
               }
            ]
         }
      }
   ]
}
```

A.5 Flow based on source IP address

The following instructions match all packets received with a source IP address equal to 10.0.0.1, and then send these to port 1. Since these are regular IP packets, their Ethernet type must be equal to 2048 (decimal), or 0800 hexadecimal. Notice that the priority for this instruction is 8, which is smaller than the priority for the destination IP address flow (A.6):

```
{
    "flow": [
        {
            "table_id": 0,
            "id": "301",
            "priority": 8,
            "cookie": "8",
            "match": {
                "ipv4-source": "10.0.0.1/32",
                "ethernet-match": {
                    "ethernet-type": {
                        "type": 2048
                    }
                }
            },
            "instructions": {
                "instruction": [
                    {
                       "order": 0,
                       "apply-actions": {
                          "action": [
                             {
                                "order": 0,
                                "output-action": {
                                   "output-node-connector": "1"
                                }
                             }
                          ]
                       }
                    }
                ]
            }
        }
    ]
}
```

A.6 Flow based on destination IP address

The following instructions match all packets received with a destination IP address equal to 10.0.0.1, and then send these to port 3. Since these are regular IP packets, their Ethernet type must be equal to 2048 (decimal), or 0800 hexadecimal. Notice that the priority for this instruction is 9, which is larger than the priority for the source IP address flow (A.5):

```
{
    "flow": [
        {
            "table_id": 0,
            "id": "411",
            "priority": 9,
            "cookie": "9",
            "match": {
                "ipv4-destination": "10.0.0.1/32",
                "ethernet-match": {
                    "ethernet-type": {
                        "type": 2048
                    }
                }
            },
            "instructions": {
                "instruction": [
                    {
                       "order": 0,
                       "apply-actions": {
                          "action": [
                             {
                                "order": 0,
                                "output-action": {
                                   "output-node-connector": "3"
                                }
                             }
                          ]
                       }
                    }
                ]
            }
        }
    ]
}
```

Appendix B

Mininet Example Files

B.1 ODL cluster example

The following python code is used to test ODL's cluster at Section 5.3:

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Controller, OVSSwitch,
RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel, info

def MyNet():
    net = Mininet(controller=RemoteController,
          switch=OVSSwitch)
    c1 = net.addController('c1', controller=RemoteController,
         ip="172.17.0.2", port=6653)
    c2 = net.addController('c2', controller=RemoteController,
         ip="172.17.0.3", port=6653)
    c3 = net.addController('c3', controller=RemoteController,
         ip="172.17.0.4", port=6653)

    h1 = net.addHost( 'h1' ) #, ip='192.168.1.10' )"
    h2 = net.addHost( 'h2' ) #, ip='192.168.1.20' )"
    h3 = net.addHost( 'h3' ) #, ip='192.168.2.40' )"
    h4 = net.addHost( 'h4' ) #, ip='192.168.2.50' )"

    s1 = net.addSwitch( 's1' , protocols="OpenFlow13")
    s2 = net.addSwitch( 's2' , protocols="OpenFlow13")
    s3 = net.addSwitch( 's3' , protocols="OpenFlow13")
    s3.linkTo( h1 )
```

```
    s3.linkTo( h2 )
    s2.linkTo( h3 )
    s2.linkTo( h4 )
    s1.linkTo( s2 )
    s1.linkTo( s3 )
    net.build()
    c1.start()
    c2.start()
    c3.start()
    s1.start( [c1, c2, c3] )
    s2.start( [c1, c2, c3] )
    s3.start( [c3, c2, c1] )
    net.start()
    net.staticArp()
    CLI( net )
    net.stop()

if __name__ == '__main__':
     setLogLevel( 'info' )
     MyNet()
```

B.2 Spine-leaf example

The following python code is used to test the spine-leaf network topology at Section 6.4:

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Controller, OVSSwitch,
RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel, info

def spineLeafNet(N=6):
    # Need to verify that N is an even number and larger than 3
    if N<4:
        info('Error: N should be greater than 3 and even.')
        return False
    else:
        if N%2 == 1:
            info('Error: N should be even and greater than 3.')
            return False

    net = Mininet(controller=RemoteController,
                  switch=OVSSwitch)
```

```
    c1 = net.addController('c1', controller=RemoteController,
                           ip="172.17.0.2", port=6653)

    # The value of i needs to be increased by 1 because the
    # switch and host names should start on that number. A
    # value of 0 creates inconsistencies with OpenDaylight.
    switchArray = []
    for i in range(N):
        switchArray.append(net.addSwitch(f's{i+1}',
                           protocols="OpenFlow13"))

    hostArray = []
    for i in range(N):
        hostArray.append(net.addHost(f'h{i+1}'))

    # Connect spine switches (0,N/2) to leaf switches (N/2,N)
    for i in range(N//2):
        for j in range (N//2,N):
            net.addLink(switchArray[i], switchArray[j])

    # Connect leaf switches (N/2+1,N) to 2N hosts
    j = 0
    for i in range(N//2,N):
        #info("Running loop " + str(i) + "," + str(j) + "\n")
        net.addLink(switchArray[i], hostArray[j])
        j += 1
        #info("Again t loop " + str(i) + "," + str(j) + "\n")
        net.addLink(switchArray[i], hostArray[j])
        j += 1

    net.build()

    c1.start()

    for switch in switchArray:
        switch.start([c1])

    net.start()

    CLI( net )
    net.stop()

if __name__ == '__main__':
     setLogLevel( 'info' )
     spineLeafNet()
```

Bibliography

[1] "Mininet." [On-line]. Available: http://mininet.org/, Nov. 2022.

[2] "OpenDaylight." [On-line]. Available: http://www.opendaylight.org/, May 2024.

[3] "Official OpenDaylight documentation page." [On-line]. Available: https://docs.opendaylight.org, Nov. 2022.

[4] "VirtualBox." [On-line]. Available: https://www.virtualbox.org/, May 2024.

[5] "Docker." [On-line]. Available: https://www.docker.com/, May 2024.

[6] M. Bose, "VirtualBox network settings: Complete guide." [On-line]. Available: https://www.nakivo.com/blog/virtualbox-network-setting-guide/, July 2019.

[7] "Docker hub page." [On-line]. Available: https://hub.docker.com/, Jan. 2023.

[8] "Apache Karaf page." [On-line]. Available: https://karaf.apache.org/, Nov. 2022.

[9] "curlx." [On-line]. Available: https://www.curlx.dev/, May 2024.

[10] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "RFC 6241: Network Configuration Protocol (NETCONF)." [On-line]. Available: https://datatracker.ietf.org/doc/rfc6241, June 2011.

[11] A. Bierman, M. Bjorklund, and K. Watsen, "RFC 8040: RESTCONF Protocol." [On-line]. Available: https://datatracker.ietf.org/doc/rfc8040/, Jan. 2017.

[12] "OpenDaylight Documentation: Setting Up Clustering." [On-line]. Available: https://docs.opendaylight.org/en/latest/getting-started-guide/clustering.html, Feb. 2023.