UNIVERSIDAD DE VALLADOLID

E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS ESPECÍFICAS DE TELECOMUNICACIÓN

Deployment of a GPON-SDN solution in a server using Docker

Autor:

Don Anil Neupane

Tutor:

Doña Noemí Merayo Álvarez

Don Juan Carlos Aguado Manzano

TÍTULO: **Deployment of a GPON-SDN solution in a server using Docker**
AUTOR: **Don Anil Neupane**
TUTOR: **Doña Noemí Merayo Álvarez**
DEPARTAMENTO: **Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

FECHA: 23 de Junio de 2020
CALIFICACIÓN:

Este Trabajo Final de Grado ha sido desarrollado por el estudiante dentro de una movilidad del programa ERASMUS+. Dado que la carga lectiva del proyecto fin de carrera en su universidad origen es de 30 ECTS, se ha diseñado un programa en la UVa que también implique la realización de 30 ECTS por un trabajo equivalente (en nuestro caso, trabajo fin de grado). Esto conlleva a la realización de un trabajo de 30 ECTS (un cuatrimestre completo) y la presentación del mismo en dos TFGs, en concreto el TFG para el Grado en Ingeniería en Tecnologías de Telecomunicación (6 ECTS) y el TFG del Grado en Ingeniería en Tecnologías Específicas de Telecomunicación (12 ECTS). Para no duplicar esfuerzos se ha incluido todo el trabajo en una única memoria para ambos TFGs.

Abstract

The research carried out in this Project focuses on the transformation of a GPON network to an SDN network using the OpenFlow protocol (SDN-GPON). With this we achieve a dissociation between the control plane in charge of routing the packets and the data plane in the access network. For this, a Linux-based router has been implemented in the central computer and several OVS (Open Virtual Switch) virtual switches have been installed that can use the OpenFlow protocol and communicate with an OpenFlow central controller, in our case OpenDayLight and ONOS located in the backbone. Through this new SDN network scenario we will be able to configure and manage services and subscriber profiles in the access network through OpenFlow. During the project we tried to virtualize most of the applications we needed using the Docker technology, some of the virtualizations were forced upon us because of the unforeseen circumstances (Covid-19, unable to access the labs) but in the end we managed to make it work as much the circumstances allowed us to.

Keywords

SDN (*Software-designed networks)*, GPON (*Gigabit-capable Passive Optical Networks*), OLT (*Optical Line Termination*), ONU (*Optical Network Unit*), OpenFlow, OpenDayLight, Open Network Operating System.

Acknowledgement

This work has not been made in complete solitude. Without the support of many people in my direct and indirect surrounding, this thesis would have never happened. It was not something I was familiar with and I hope I present something that's interesting to read.

First of all I want to thank my supervisor Prof. Noemí who consistently guided me in the right direction whenever I had trouble finding solutions, I'm also grateful to my supervisor Prof. dr. Juan Carlos, for his valuable guidance and for sharing their expertise and even researching new topics which sometimes were also new for him. Both the professors were always ready to answer any questions I had, and helped a lot with the proof reading and I am really grateful for that.

Secondly, I want to thank my friends, who, despite not always understanding what I was talking about, kept listening to my ramblings and reasoning, and helped me to find where they were wrong. Just by talking to them I made a lot of new breakthroughs. I especially want to thank Jelle Debuyzere, Robin Spruytte and Thomas Van Giel aiding me with some of these reasonings.

Finally, I must express my very profound gratitude to my family and all other friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

The research developed in this Final Degree Project has been funded by the Ministerio de Ciencia, Innovación y Universidades in the National Projects ONOFRE-2 (TEC2017-84423-C3-1-P) and Go2Edge (RED2018-102585-T), by the Consejería de Educación de la Junta de Castilla y León in the Project ROBIN (VA085G19), and by the European Regional Development Fund FEDER by means of the Proyect DISRUPTIVE within the framework of the program Interreg V-A España-Portugal (POCTEP) 2014-2020. Opinions are the sole responsibility of the author who issues them.

Index of figures

1 Introduction

1.1 Motivation

In this Project, the implementation of the SDN (Software Designed Network) scenario has been carried out in a GPON access network (Gigabit Passive Optical Network) through the OpenFlow protocol. This access network is located in the Optical Communications Laboratory (2L007) of the Higher Technical School of Telecommunications Engineers of the University of Valladolid.

An SDN scenario implies that the control plane and the data plane are separated, the control plane being separated by software in a controller (in our case, an OpenFlow controller). OpenFlow is an open standard that allows the creation of different experimental protocols. In classic routers and switches, the data path (sending data from one place to another) and the control path (routing decision-making) occurred on the same computer [1]. Using OpenFlow, the control path can be moved to a controller that will communicate with the switch or router where the data path still resides using OpenFlow messages. One of the great advantages of this design is that a single controller could operate on different switches or routers simultaneously, allowing the connections and interactions between them to vary depending on the state of the network.

Thanks to this flexibility and real-time control of our SDN-GPON network, we will be able to offer the end customer the possibility of dynamically managing their contracted services.

1.2 Objectives

1.2.1 *General Objective*

The main objective of this Final Degree Project is to deepen the implementation of OpenFlow in a GPON access network to continue converting the network to an SDN paradigm in which the data planes and the control planes are separated. For this, services and requirements of quality parameters will be configured and managed using OpenFlow, which will require the installation of several virtual switches implemented in different parts of the GPON network. Specifically, they will be placed before the OLT (Optical Line Terminal) and after the ONT (Optical Network Terminal) to be able to control and monitor the entry of any packet into our GPON access network.

When our network is completely SDN and we can control the services and requirements in real time both upstream and downstream, we will implement a web interface so that the client dynamically controls their bandwidth in real time. In this way, a new business model will be introduced for operators in which users may be able to modify their requirements in real time after paying for the use of said network resources. This general objective can be broken down into more specific ones:

1.2.2 *Specific Objectives*

By carrying out this study, the following specific objectives needs to be met:

- Analysis of the GPON network topology prior to carrying out the research for its modification.
- Implementation of a router and DHCP server in an external server.
- Implementation of the SDN layer in the GPON network by installing and configuring virtual switches in an external server in order to emulate the OLT side.
- Evolution in the integration of the OpenFlow standard in a GPON network model.

- Communication between the virtual switches and the OpenDayLight and ONOS controllers for their OpenFlow management.

- Creation of services and requirements management using the OpenFlow standard in the SDN-GPON implementation.

1.3 Stages of the Project

The methodology to follow for the development of the objectives of the Project mainly consists of the phases that will be explained below.

1.3.1 *Analysis stage*

The purpose of this phase is to learn in a basic way how the main components of this Project work:

- *Analysis of the GPON network topology of laboratory 2L007:* study of the different network components and the connection between them to implement a router and a DHCP server within said network.
- *Analysis of the OpenFlow protocol:* study of the basic operation of this protocol, its different versions and types of existing controllers.
- *Analysis of the Open vSwitch:* study of its operation, implementation and configuration, as well as the analysis between its different versions and functionalities.
- *Analysis of the SDN implementation in the GPON network:* study of the inner workings of the OpenDayLight controller, ONOS controller, virtual switches and communication between them for the dynamic management of the bandwidth.
- *Analysis of the Docker Technology:* study of the workings of Docker, virtualisations with docker, inner workings of networks in docker and investigate on running the earlier mentioned controllers, ovs and dhcp server inside a docker container.

1.3.2 *Implementation stage*

This phase aims to implement the different programs or machines necessary to meet specific objectives. Before each implementation, the necessary specific analysis will be carried out.

For this, we will need to optimally manage the Linux operating system, since many of the implementations are based on commands and utilities specific to this operating system, such as *iptables* or *vconfig*.

1.3.3 *Testing Stage*

In this final phase of the project, a test task will be carried out to observe the performance of the real network thanks to two tools, iperf and Wireshark. With the first we are going to be able to measure the network speed and the real up and down traffic on our network. With the second one we will be able to observe the OpenFlow messages and we will be able to obtain graphs on the performance of TCP communications thanks to tcptrace, a utility included within Wireshark.

1.3.4 *Reporting stage*

In this phase, the project reports are carried out.

2 Methodology and Software tools

2.1 Introduction

In this chapter, a descriptive analysis of the main components used in this Project will be carried out, i.e. the GPON access network, the OpenFlow protocol, the OpenDayLight controller, the ONOS controller, the Docker virtualization tool and the Open vSwitch. In addition, the methodology used to carry out this work will also be described.

The GPON access network (from the manufacturer Telnet Redes Inteligentes [2]) is located in the Optical Communications laboratory (2L007) of the Higher Technical School of Telecommunications Engineers at the University of Valladolid. The main elements of this network are the OLT (Optical Line Termination) and the ONUs / ONTs (Optical Network Unit / Terminal) where the OLT provides its services. These services can be configured in two different ways: by using the visual TGMS interface (TELNET GPON Management System) or directly, by accessing the OLT via CLI (Command Line Interface) through its configuration port. However, in this project, we are going to configure and manage the services and subscriber profiles of residential users (connected to ONTs) using the OpenFlow protocol [2].

For this part, OpenFlow is a standard that allows the creation of experimental protocols based on the separation it makes between data path (sending from one place to another) and control path (decision-making), which allows SDN to be introduced in networks. These terms would be like the control plane and the data plane. OpenFlow has several versions and controllers, being version 1.3 and the OpenFlow OpenDayLight [3] and the ONOS controller will be used in this work. Finally, the Open vSwitch virtual switch [4] is an open source virtual switch, with various functions currently under development that allows us to implement an OpenFlow switch transparently to the user and the network in general. This virtual switch allows connecting to OpenFlow controllers to be easily and transparently managed by them through the OpenFlow protocol. This new network scenario is the one we want to implement in our GPON real optical access network.

2.2 GPON testbed in the Optical Communication laboratory

This section of the chapter will describe both the structure and components of the access network and the different management modes available to the GPON network deployed in the laboratory.

An access network is a set of elements that allow end users to connect with service providers, so that they can give said users the services they have contracted.

Our GPON optical access network has a tree topology, where one OLT serves multiple ONUs / ONTs. In the case of our laboratory network, the OLT serves a total of 4 ONTs on port 0, being expandable up to a maximum of 64 ONTs per port. Since the OLT has a total of 4 ports, the OLT can support a total of up to 256 ONTs. Other elements of the network will be the single-mode optical fiber that connects the different optical components together and the splitters (optical dividers), responsible for dividing the signal so that each of the ports of the service OLT to various ONUs. The optical fiber deployed in this testbed can reach 25 km (according to the GPON standard), and the optical splitters are in a 1: 8 ratio (two are available, although one is currently connected). The general appearance of the GPON access network is shown in the image in Figure 1.

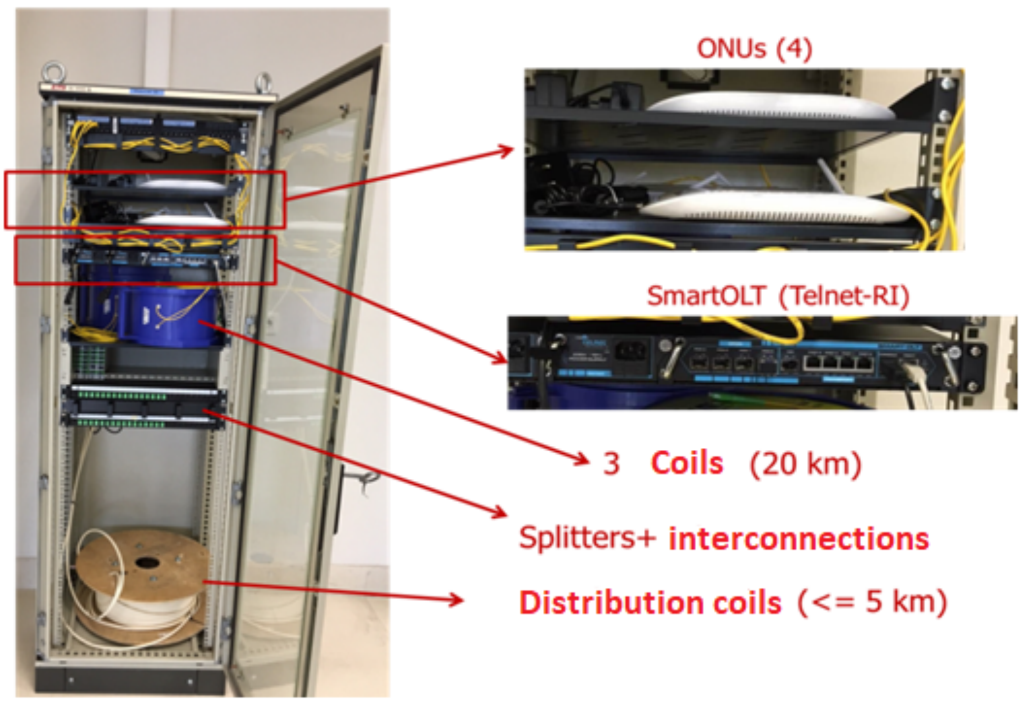


Figure 1 : General diagram of the GPON access network deployed in the laboratory

The components of the access network have been manufactured by the company TELNET Redes Inteligentes [2], so that compatibility between them is guaranteed. Depending on the direction that the data takes, one can speak of two different flows or channels:

- **Downstream channel:** known as downstream, it is the flow that carries data from the central office (OLT) to the end users (ONUs). According to the GPON standard, the maximum downstream rate allowed is 2.5 Gbps and the wavelength is 1490 nm.
- **Upstream channel:** known as upstream, it is the flow that carries data from users to the central office (the ONU is the source). According to the GPON standard, the maximum rate allowed in the upstream is 1.25 Gbps and the wavelength is 1310 nm.

It is important to consider these two channels, since the total sum of the upstream and downstream rates given to each of the services provided to the different ONTs / ONUs cannot exceed the maximum imposed by the GPON standard. Otherwise, the network will trigger an error in its configuration and start-up.

2.3 Docker

Docker is a virtualization tool designed to make it easier to create, deploy, and run applications by using containers [5]. Containers allow a developer to package up an application with all the parts it needs, such as libraries and other dependencies, and deploy it as one package. By doing so, thanks to the container, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that the machine might have that is different from the machine, the code or the application was written on.

In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the host system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.

There are two main internal components of docker: images and containers.

2.3.1.1 Images

Images are read-only templates with instructions to create a docker container. Images can be pulled from the Docker Hub. Docker Hub is very similar to GitHub for the users who are more familiar with it. Most open source communities including onos, node, Debian and many others have prebuilt updated docker-images available for the developers on the Docker Hub.

Instead of using a pre-built images by the organizations a Docker image can also be created using a Dockerfile. Dockerfile is a simple text file with a set of commands or instructions. These instructions are executed successively to perform actions on the base image to create a new docker image. These commands or instructions include installing packages, dependencies, exposing a specific host port to run the webserver application and running a script (For example: instruction to run a NodeJS script, if the user is building an image for NodeJS application).

In the scenario where the image is self-built, the workflow looks like it can be observed in Fig 2.

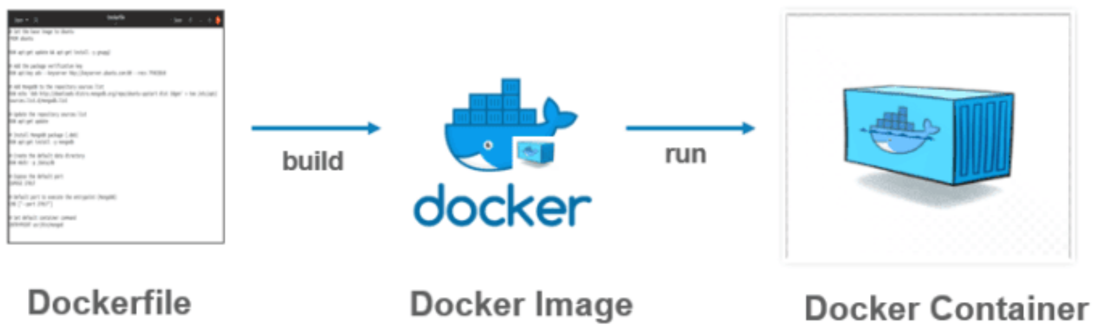


Figure 2 : The workflow when the image is built from Dockerfile

- Create a Dockerfile and mention the instructions to create a docker image.
- Run docker build command which will build a docker image.
- Now the docker image is ready to be used, use docker run command to create container.

In this project we will mainly focus on using pre-built images as building your own images for the complex software tools like SDN controllers and the virtual switches could use up a lot of time and it is easier and faster to use the pre-built images.

2.3.1.2 Containers

A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. An image becomes a container at runtime. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure of the Docker host [5]. Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing costs.

Containers and traditional virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient.

2.4 SDN (Software Defining Networking)

The basis for the most precise definition of what SDN would be the idea of programmability. A technology that separates the control plane management of network devices from the underlying data plane that forwards the network traffic [1].

SDN architectures feature software-defined overlays or controllers that are abstracted from the underlying network hardware, offering intent-or policy-based management of the network. This results in a datacenter network that is better aligned with the needs of application workloads through automated (thereby faster) provisioning, and programmatic network management.

SDN promises to reduce the complexity of statically defined networks; make automating network functions much easier and allow for simpler provisioning and management of networked resources.

At its heart SDN has a centralized or distributed intelligent entity that has an entire view of the network, that can make routing and switching decisions based on that view.

Typically, network routers and switches only know about their neighboring network gear. But with a properly configured SDN environment, that central entity can control everything, from easily changing policies to simplifying configuration and automation across the enterprise. Figure 3 shows how different switches are controlled by a single SDN controller.

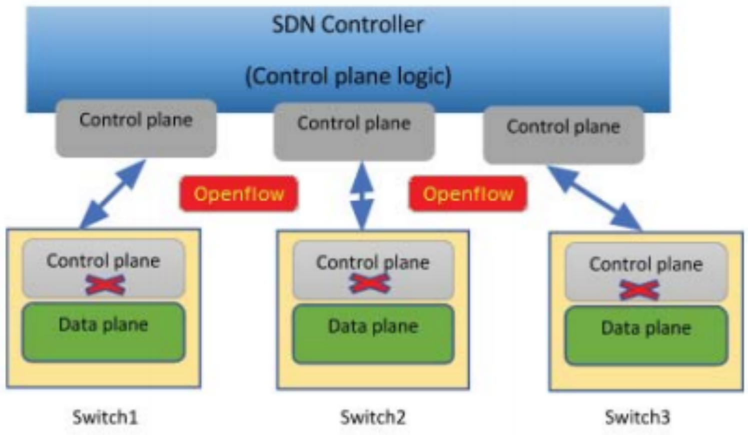


Figure 3 : The control plane remains in the hands of the controller

2.5 OpenFlow standard

OpenFlow is an emerging and open communications protocol that allows the creation of new protocols thanks to the division it makes between transmitting data from one point to another and making routing decisions within a switch or router. With OpenFlow, a part of the data path resides on the same switch, but high-level routing decisions are made by a controller. Both elements communicate through the OpenFlow protocol. This methodology, known as SDN, allows greater effectiveness in the use of network resources than in a conventional network.

In classic routers and switches, the data path (sending data from one place to another) and the control path (routing decision-making) occurs on the same computer [2]. Using OpenFlow, the control path can be moved to a controller that will communicate with the switch or router where the data path still resides using OpenFlow messages. One of the great advantages of this design is that a single Controller could operate on different switches or routers simultaneously, allowing the connections and interactions between them to vary, depending on the state of the network.

The basic operation is as follows: an OpenFlow controller, responsible for routing decisions, connects to an OpenFlow switch or router with several terminals connected on its different ports. This controller constantly communicates with the switch or router through the sending of OpenFlow messages, so that it can configure the connections between the hosts connected to the switch or router based on data that reaches it from there. For example, imagine two different hosts A and B connected to ports 1 and 2 of a switch respectively, and that the switch has no configuration set. If host A wants to send something to host B, in principle it could not due to this lack of configuration. However, thanks to OpenFlow, the switch can detect that it has a packet on port 1 that must go to port 2, send a notification to the controller in the form of an OpenFlow message, and receive in response one or more OpenFlow tables with the necessary configuration for hosts A and B can communicate without any problem.

On the other hand, there has been talk of setting up OpenFlow tables. An OpenFlow table is an entity that contains different flows, which are used by the switch to perform different operations (instructions) if certain conditions are met (match). In the previous example, one of the OpenFlow tables that the controller sends to the switch could be one with a flow that says that if the input port to the switch of the packet is 1, the packet is sent through port 2. In addition, they can nest tables and flows with different priority orders, so that the switch can consider many different parameters when processing the messages that arrive [6].

2.6 SDN controllers

The OpenFlow controller is the "brain" that dictates to the switch or router what to do with the incoming and outgoing data, using different types of OpenFlow messages also known as flow entries. A single controller can manage multiple routers or switches in real time, allowing you to radically change the connections of a network based, for example, on parameters such as traffic passing through certain points [6].

Currently there are several types of controllers that differ in the versions of the OpenFlow standard that they support and in the programming language in which the different applications available in each of them are written. The OpenDayLight controller was used for this work [3], as this controller is the one implemented in previous works, which is a continuation of the research carried out in it. Finally, ONOS [7] controller will also be used to test the same features similar to the ODL.

2.6.1 OpenDayLight

The OpenDayLight controller is a versatile driver programmed in Java and supporting OpenFlow versions 1.0 and 1.3. This controller is not intended exclusively for the use of OpenFlow, since OpenFlow is just one of several protocols and standards that are part of Software Defined Networks (SDN), networks that can vary their functionality by using of different programs [2]. However, in this work we will limit ourselves to using its OpenFlow aspect.

To allow administration, OpenDaylight offers two APIs. An OSGI (Java) API for applications in the same address space and an ordinary REST API. The command line is only for installing packages or features and getting very basic information about the driver, not the network. Unlike ONOS, the OpenDaylight (DLUX) graphical interface is not very friendly. In addition to displaying a simple graph with the network topology, the

OpenDaylight YANG UI module allows you to interact graphically with the REST API, making calls to the API and offering information on the data structures it uses. Through this graphical user interface you can configure the network or obtain detailed information on the infrastructure [8].

The DLUX GUI features are only available when running the older versions of ODL (up to the 8th version). For this project we will be looking to work with the latest versions (11th) so the graphical user interface will be replaced by an application named Postman.

2.6.2 **Open Network Operating System (ONOS)**

ONOS (Open Network Operating System) is an Open Source SDN controller managed by the Linux Foundation [7]. Among its main goals is to offer a scalable, high performance controller with support for high availability and compatible with both traditional devices and OpenFlow devices. It is written in Java and built on Apache Karaf.

It is the most telemarketer-oriented controller but can also be used (and is used) within data processing centers. It has the support of large companies within the telecommunications sector. Mainly, from some of the world's largest telemarketers, such as AT&T, China Unicom, Comcast or Deutsche Telekom [7]. It also has the support of manufacturers like Huawei, who work closely with ONOS to deploy real scenarios.

It is a well a documented controller. All the information is correctly classified and updated. It is worth noting the ease of setting up a simple SDN network with ONOS surpasses that of an ODL controller. Only two applications need to be activated, *org.onosproject.openflow* in order to communicate the OpenFlow switches with the controller, and *org.onosproject.fwd* to enable reactive forwarding. Reactive forwarding consists of creating flows dynamically when the controller does not have any specific flow to handle that traffic (like a normal flow in ODL).

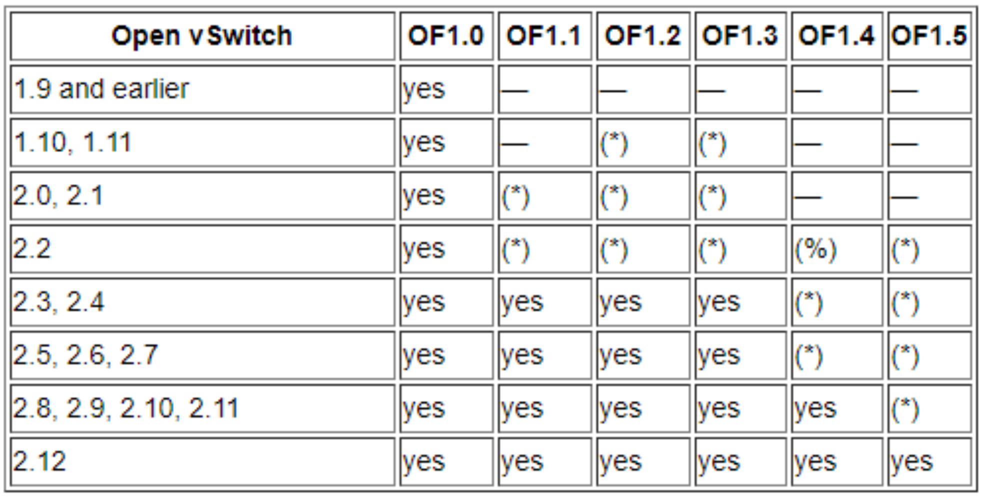
ONOS can be managed through its REST API, command line (CLI) and graphical web interface. The command line is based on the Apache Karaf CLI and allows you to manage the main characteristics of the network. For example, it allows you to create, delete, and modify flows and to manage network devices. Regarding the graphical interface, it is primarily intended to offer a graphical representation of the network topology and its state [8], in this project the ONOS controller will also be configured via the user interface provided by the REST API documents.

2.7 Open Virtual Switch (Open vSwitch – OVS)

Open vSwitch, short for OVS, is open source software designed to be used as a virtual switch in virtualized environments. It is designed to enable massive network automation, while supporting standard management interfaces and protocols (NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, 802.1). Also, it is designed to support a distribution for multiple physical servers similar to the VMWare's vNetwork switch or the Cisco's Nexus 1000V [9].

In relation to our project, Open vSwitch is one of the most popular implementations of OpenFlow and supports this protocol by default. This allows us to implement an OpenFlow layer on different devices transparently to the user and the rest of the network. Conceptually, the functions of a switch can be divided into two planes (the control plane and the data plane). The control plane is the core intelligence of the switch, which is responsible for discovery, routing, path computation, and communication with other switches. In this sense, OpenFlow allows us to download the control plane of all the switches to a central controller that defines the behavior of the network (what is known today as SDN) [10].

In this way, Open vSwitch becomes a perfect option for our project, since it combines default support for the different versions of OpenFlow, as it can be seen in Figure 4 [9]. In addition, it is prepared to perform SDN and is light and transparent to the user and other networks.



Open vSwitch	OF1.0	OF1.1	OF1.2	OF1.3	OF1.4	OF1.5
1.9 and earlier	yes	—	—	—	—	—
1.10, 1.11	yes	—	(*)	(*)	—	—
2.0, 2.1	yes	(*)	(*)	(*)	—	—
2.2	yes	(*)	(*)	(*)	(%)	(*)
2.3, 2.4	yes	yes	yes	yes	(*)	(*)
2.5, 2.6, 2.7	yes	yes	yes	yes	(*)	(*)
2.8, 2.9, 2.10, 2.11	yes	yes	yes	yes	yes	(*)
2.12	yes	yes	yes	yes	yes	yes

Figure 4 : versions of OpenFlow supported by each version of Open vSwitch

The symbols of Figure 4 means:

- — Not supported.
- **yes** Supported and enabled by default.
- (*) Supported but missing features, and must be enabled by user.

2.8 POSTMAN

Postman is an API Development Environment that helps people to build, test, document, monitor and publish documentation for their APIs. Postman users enter data, the data is sent to a special web server address. Typically, information is returned, which Postman presents to the user. Postman's features simplify API building and streamline collaboration.

With Postman all REST queries can be executed. Any type of request can be sent in Postman. It offers a sleek user interface with which to make HTML requests, without the hassle of writing a bunch of code just to test an API's functionality.

Postman can run GET, PUT, PATCH, DELETE, and various other request methods as well, and has utilities to help with developing APIs. Free and paid versions are available for Mac, Windows, Linux, and as a Chrome app [11].

2.9 IPERF

IPERF is an open source tool that can be used to test network performance. Iperf is much more reliable in its test results compared to many other online network speed test providers [12]

An added advantage of using IPERF for testing network performance is the fact that, it is very reliable if you have two servers, both in geographically different locations, and you want to measure network performance between them.

The operating system does not matter, while you are using iperf. The commands for using iperf on windows is the same as in linux and another operating system. Normally in the test environment, iperf client sends data to the server for the test.

2.10 Methodology

In this section of the chapter, a description will be made of the methodology and steps to be followed for the final achievement of the proposed objectives. Recall that the objective of the work is the integration of the OpenFlow standard in our GPON model to implement an end-to-end SDN scenario and thus separate the data plane from the control plane.

2.10.1 *Iperf tests on the Optical Network cards on the server and Docker*

The first step is to use the Iperf technology to test the throughput of the network cards on the servers, this is necessary so we understand the capacity it holds. After the Iperf tests, we also need to study the docker configurations, how it works, and als run the through-put tests inside the docker container to check for differences. Subsequently more research will be done regarding installation of the softwares like OVS, ONOS, ODL, DHCP server inside a docker container.

2.10.2 *Linux Routing programming and a DHCP server*

The second step to achieve the desired scenario is to program a router on a computer. To do this, we will use the Linux kernel utility to forward IP packets between several networks (that is, that our computer acts as a router). We need to perform this task to control the network that reaches the GPON model. Subsequently, a DHCP (Dynamic

Host Configuration Protocol) server will be installed to give the network configuration parameters to the machines that connect to the Router.

2.10.3 *Installation of the OVS*

The third step consisted of installing the OVS, to simulate the implementation of the SDN layer on the OLT side. To do this, we follow the guide provided by its official website [13] and then complete the installation.

2.10.4 *Connection of OVS with controllers (ODL and ONOS)*

The next step is to configure within the different OVS switches in which direction or address the controller is located. After specifying this on all the switches, we must configure the routing tables on the device with OVS and the Router, so that there is communication between the switches and the controller, even if they are on different subnets. At the telematic level, a layer 3 subnet (network layer) is defined as all those devices that are between a router and another router.

In other words, routers divide network space into different networks or subnets. In Figure 5 we can see how our real experimental network is divided into different subnets. In this topology we have several subnets, particularly subnets (192.168.0.0/24 and 10.19.59.0/24) between the router and each of the ONTs (although if we add more VLANs we would have one subnet for each different VLAN); and the subnets of each ONT that is defined between the ONT itself and the end customer (192.168.x.0 / 24). In this way, if we divide the network correctly maintaining connectivity between the devices, we will improve the security and performance of our network. In the GPON network, the OLT is transparent at the network layer, i.e. it is not a router as such, although it does have certain network layer functionalities. Therefore, this device does not divide subnets between its different interfaces.

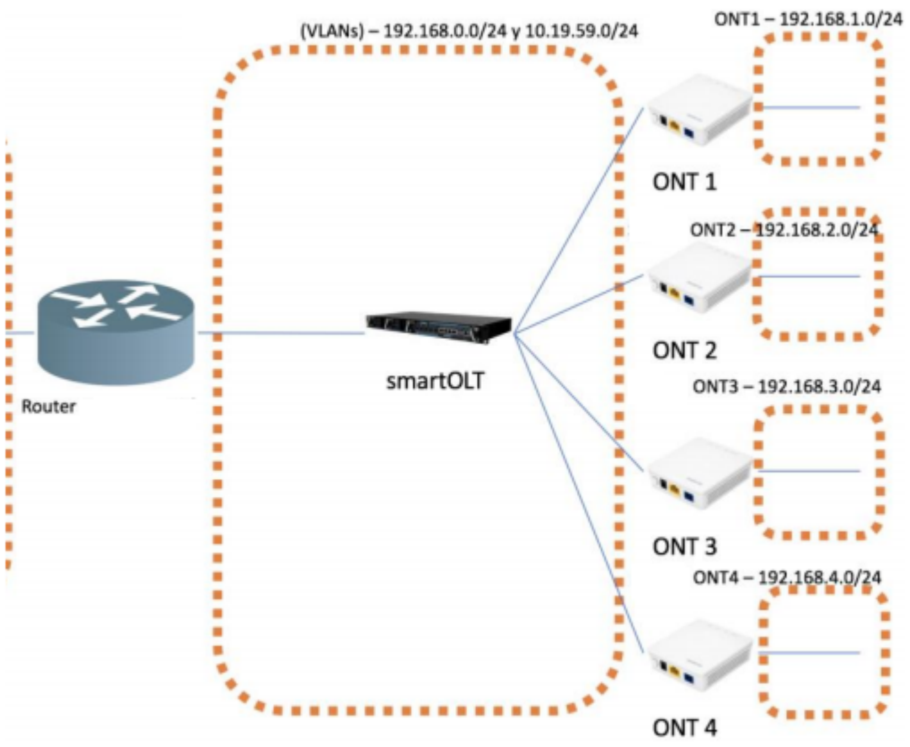


Figure 5 : Structure of the different subnets present in our real network scenario

2.10.5 *Programming and display of flow tables*

The next step is to program the flow tables as well as the flows in the OpenDayLight controller and the ONOS controller instead of doing it manually on each OVS. To do this, on the OpenDayLight, we will use Postman and on the ONOS its own GUI will be used. This both utilities allow configuring the flows with a graphical interface friendlier for the administrator.

OpenFlow 1.3 has several tables in which to introduce our flows. By default, table 0 is used every time we add new flows, although OpenFlow supports up to 255 tables. Although we use only one table in which we enter all the flows, because our flows only have two different instructions, some network administrators prefer to add more tables and dedicate each table to perform a certain action and separate each flow into different tables logically and according to their behavior.

Each flow will have several fields configured (match, action, instruction ...) that we will define later in Chapter 4. Since the objective of this investigation is to configure the services of the GPON network using OpenFlow, we will try to define these services through these OpenFlow flows. Therefore, once the different flows and tables have been configured, it will be proven that the services and traffic of the GPON network are managed efficiently and automatically by means of these configured flows. To do this, the relevant tests will be launched and observed in the real GPON network, the results of which will be shown in Chapter 4.

For example: different flows could be created on different tables for a specific packet. A flow entry on table 0 will forward any packet coming from specific NA (for example:192.168.3.0/24) to table 1, then on the table 1 the packet will be moved on to table 2 if the packet's destination address is 192.168.4.5, finally in the table 3 then flow entry could be configured so that specific packet gets outputted through one of the ports on the switch and eventually even apply a meter to it if it is needed. This method could be used to apply security measures to the packets passing through an OVS bridge just like it is done in traditional routers to discard specific packets.

2.11 Summary

In this chapter, the tools that will be used in this Project have been described, as well as the work methodology to achieve the final objective, that is, implement an SDN layer using the OpenFlow protocol in a real GPON model. Specifically, the steps to carry out said implementation have been explained. To do this, we need to implement a router with DHCP, we need to install a virtual switch, finally, each switch will be connected to the SDN controllers. With this implementation, flows can be programmed in the controllers in order to manage the traffic of the different services assigned in each ONT (user).

3 IPERF tests and Initial DOCKER Configuration

3.1 Introduction

In this chapter we will mainly focus on studying how ifperf works and how we can use it properly to conduct accurate tests that will be required to troubleshoot any problems we might face during this project.

After the iperf tests, we will also study the Docker documentation and test the usual docker commands and explain their working. Docker is the technology that will be used to virtualize this project and it is necessary to understand its working related to networking and its efficiency in general.

3.2 IPERF and IPERF3

As we discussed before in Chapter 2, iperf can be used to perform speed test between remote machines. It works in a client server model. To install the iperf and iperf 3, it can be done with simple command: *apt-install -y iperf iperf3*.

In this way, Figure 6 shows the current connection and IP configuration that exists between the two servers where the Iperf tests will be done. During these tests we will check if the servers are able to give the bandwidth installed on them i.e. 10 Gbps.

To perform these tests, we use the iperf command and the iperf3 command, over TCP and over UDP. The iperf is older version of iperf3 and during these tests we will also decide which one of them will be used during the tests after the SDN scenario has been installed.

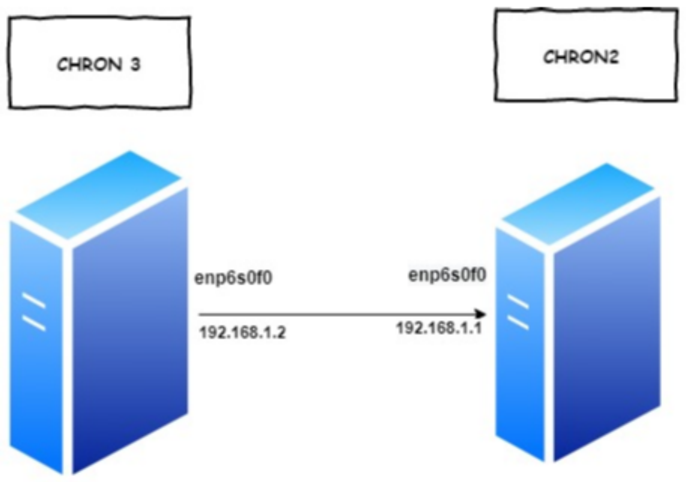


Figure 6 : Current connection between the two servers via optical interface enp6s0f0

Before going ahead with the test, lets understand some networking concepts related to speed test.

<u>Network Throughput:</u>

Transfer rate of data from one place to another with respect to time is called as throughput. Throughput is considered a quality measuring metric for hard disks, network etc. Its measured in Kbps (Kilobits per second), Mbps (Megabits per second), Gbps (Giga bits per second.)

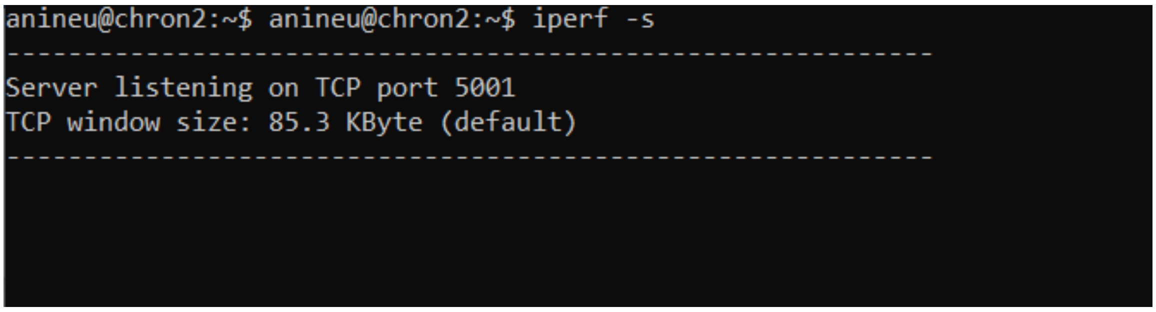
<u>TCP Window:</u>

TCP (Transmission Control Protocol), is a reliable transport layer protocol used for network communications. How TCP works, is beyond the scope of this project. TCP works on a reliable manner, by sending messages and waiting for acknowledgement from the receiver. Whenever two machines are communicating with each other, then each of them will inform the other about the number of bytes it is ready to receive at one time. In other words, the maximum amount of data that a sender can send the other end, without an acknowledgement is called as Window Size. This TCP window size affects network throughput very badly sometimes. If you increase the Window size a little bit to tune TCP, it can bring significant difference to the throughput achieved. This will be easier to understand during the tests. TCP Window Size will be modified during the TCP tests with *-w* parameter if necessary.

3.2.1 ***TCP Tests***

3.2.1.1 Iperf2

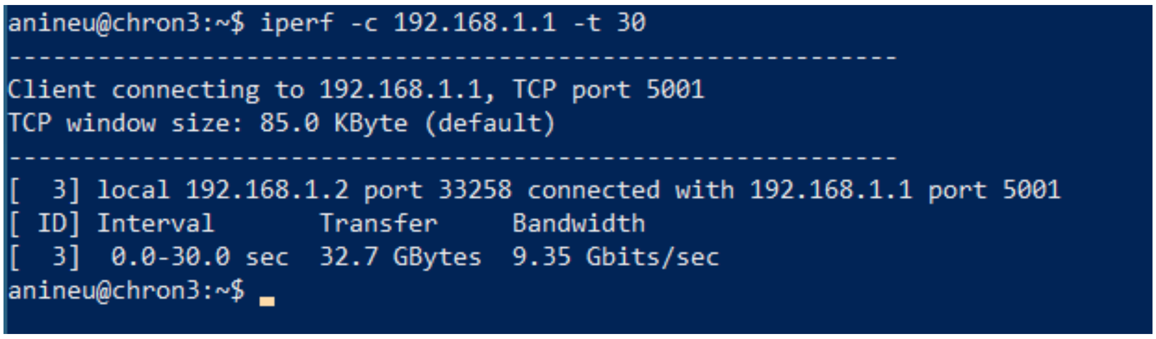
For these tests we make Chron 2 (Figure 7) our iperf server with the command *iperf -s*.



```
anineu@chron2:~$ anineu@chron2:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

Figure 7 : Making Chron 2 a server for iperf

On the Chron 3 the command *iperf -c 192.168.1.1 -t 30* can be typed to make it an iperf client that needs to connect to the server with IP 192.168.1.1 as it can be seen on Figure 8.



```
anineu@chron3:~$ iperf -c 192.168.1.1 -t 30
------------------------------------------------------------
Client connecting to 192.168.1.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 33258 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  32.7 GBytes  9.35 Gbits/sec
anineu@chron3:~$
```

Figure 8 : Making Chron 3 an iperf client

From the output in Figure 8, we can see that we got a speed of 9.35 Gbps. The speed is in the expected Range of 10Gbps. The output shows some more parameters that are explained below.

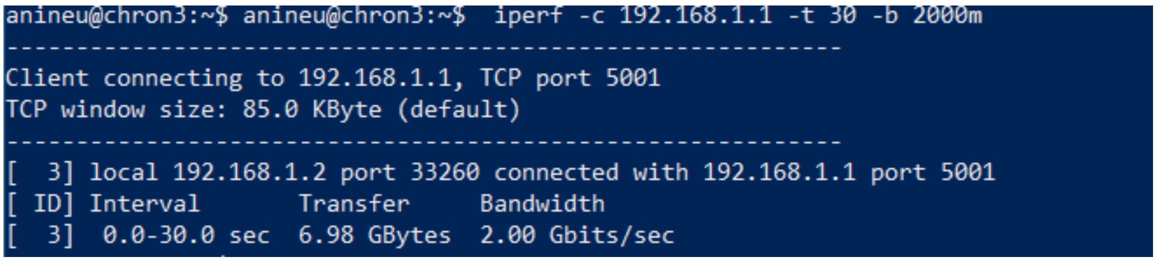
Interval: Interval specifies the time duration for which the data is transferred. This is specified by adding *t* followed by any number the user wants the interval to be with the iperf command on the client (see Figure 8).

Transfer: All data transferred using iperf is through memory and is flushed out after completing the test. So, there is no need to clear the transferred file after the test. This column shows the transferred data size.

Bandwidth: This shows the rate of speed with which the data is transferred.

In the next tests, we will be forcing TCP to have certain bandwidth instead of allowing the system to work at the maximum throughput of 10 Gbps.

The server side remains the same and we can keep it running, we need to change a few details on the command executed on the client side: *iperf -c 192.168.1.1 -t 30 -b 2000m.* We are asking the client to only throughput 2Gbps as it can be seen in Figure 9.

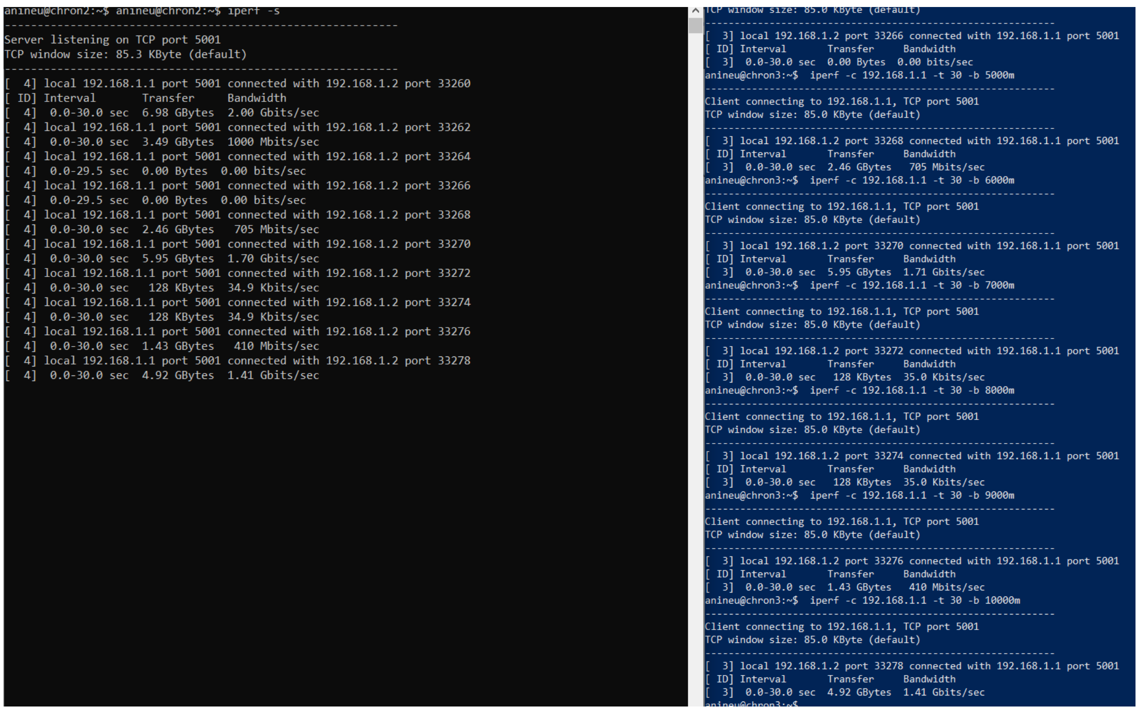


```
anineu@chron3:~$ anineu@chron3:~$  iperf -c 192.168.1.1 -t 30 -b 2000m
------------------------------------------------------------
Client connecting to 192.168.1.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 33260 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  6.98 GBytes  2.00 Gbits/sec
```

Figure 9 : TCP test with iperf for 2Gbps

Figure 9 shows that we received the exact amount of bandwidth we requested. To test if this works for all instances, we will be running these same tests from 1-10Gbps in the next steps.

Requested BW (Gbps)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Received BW (Gbps)	1.0	2.0	0.0	0.0	0.7	1.7	0.0	0.0	0.4	1.4



```
anineu@chron2:~$ anineu@chron2:~$ iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33260
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-30.0 sec  6.98 GBytes  2.00 Gbits/sec
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33262
[  4]  0.0-30.0 sec  3.49 GBytes  1000 Mbits/sec
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33264
[  4]  0.0-29.5 sec  0.00 Bytes  0.00 bits/sec
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33266
[  4]  0.0-29.5 sec  0.00 Bytes  0.00 bits/sec
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33268
[  4]  0.0-30.0 sec  2.46 GBytes   705 Mbits/sec
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33270
[  4]  0.0-30.0 sec  5.95 GBytes  1.70 Gbits/sec
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33272
[  4]  0.0-30.0 sec   128 KBytes  34.9 Kbits/sec
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33274
[  4]  0.0-30.0 sec   128 KBytes  34.9 Kbits/sec
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33276
[  4]  0.0-30.0 sec  1.43 GBytes   410 Mbits/sec
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33278
[  4]  0.0-30.0 sec  4.92 GBytes  1.41 Gbits/sec
```

```
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 33266 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  0.00 Bytes  0.00 bits/sec
anineu@chron3:~$  iperf -c 192.168.1.1 -t 30 -b 5000m
------------------------------------------------------------
Client connecting to 192.168.1.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 33268 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  2.46 GBytes   705 Mbits/sec
anineu@chron3:~$  iperf -c 192.168.1.1 -t 30 -b 6000m
------------------------------------------------------------
Client connecting to 192.168.1.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 33270 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  5.95 GBytes  1.71 Gbits/sec
anineu@chron3:~$  iperf -c 192.168.1.1 -t 30 -b 7000m
------------------------------------------------------------
Client connecting to 192.168.1.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 33272 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec   128 KBytes  35.0 Kbits/sec
anineu@chron3:~$  iperf -c 192.168.1.1 -t 30 -b 8000m
------------------------------------------------------------
Client connecting to 192.168.1.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 33274 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec   128 KBytes  35.0 Kbits/sec
anineu@chron3:~$  iperf -c 192.168.1.1 -t 30 -b 9000m
------------------------------------------------------------
Client connecting to 192.168.1.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 33276 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  1.43 GBytes   410 Mbits/sec
anineu@chron3:~$  iperf -c 192.168.1.1 -t 30 -b 10000m
------------------------------------------------------------
Client connecting to 192.168.1.1, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 33278 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  4.92 GBytes  1.41 Gbits/sec
anineu@chron3:~$
```

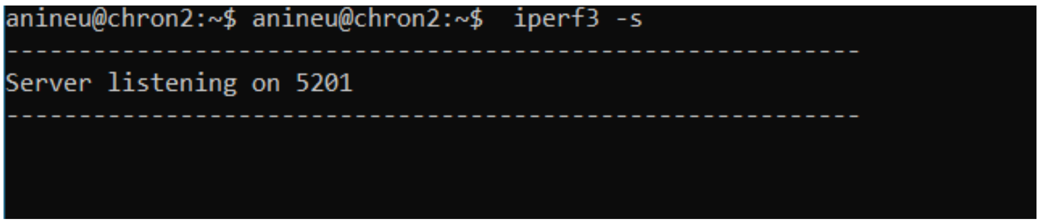
Figure 10 : TCP iperf tests from 1-10Gbps

From the data in the table above and Figure 10 we can see that the results after the 2 Gbps are not good. It is as if iperf only works efficiently when its executed at its default max rate (the maximum bandwidth of the optical cards) and not so well when the user forces what bandwidth it needs to throughput. Even when we tell the client to throughput 10 Gbps by specifying it (which is its maximum Bandwidth), it does not act as intended. In the next steps we shall be running the same tests done above using iperf3.

3.2.1.2 Iperf3

iPerf3 is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, buffers and protocols (TCP, UDP, SCTP with IPv4 and IPv6). For each test it reports the bandwidth, loss, and other parameters. Iperf was originally developed by NLANR/DAST. iPerf3 is principally developed by ESnet/Lawrence Berkeley National Laboratory [14].

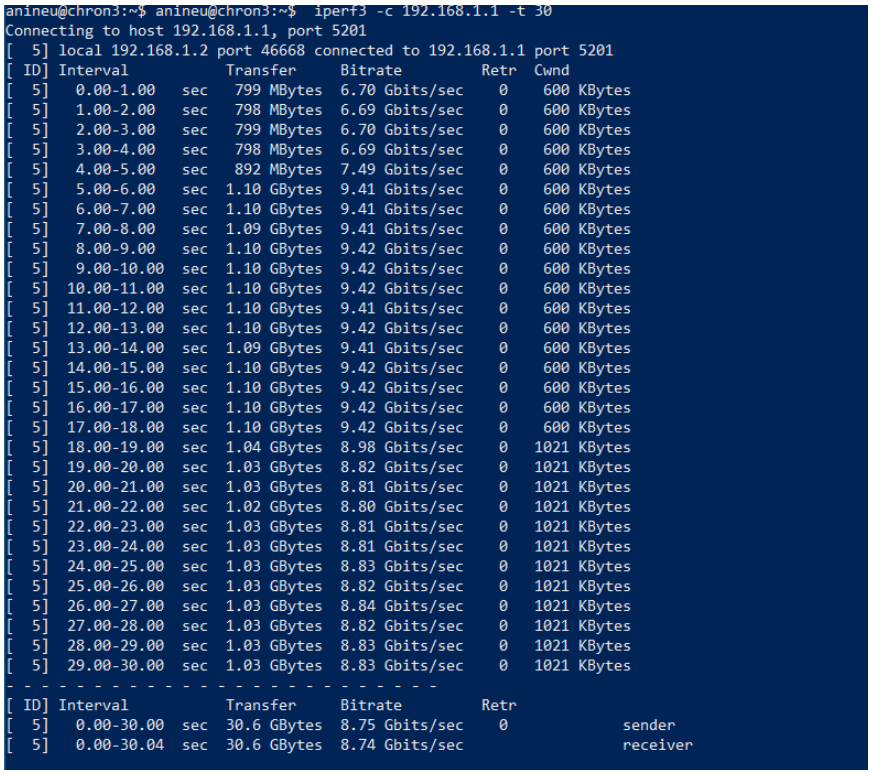
The server side with iperf3 remains almost identical to iperf, so it can be started with command *iperf3 -s* (see Figure 11).



```
anineu@chron2:~$ anineu@chron2:~$  iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 11 : make Chron 2 iperf3 server

The client side is also identical with tiny change in the command: *iperf3 -s 192.168.1.1 -t 30* as it can be observed in Figure 12.



```
anineu@chron3:~$ anineu@chron3:~$  iperf3 -c 192.168.1.1 -t 30
Connecting to host 192.168.1.1, port 5201
[  5] local 192.168.1.2 port 46668 connected to 192.168.1.1 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   799 MBytes  6.70 Gbits/sec    0    600 KBytes
[  5]   1.00-2.00   sec   798 MBytes  6.69 Gbits/sec    0    600 KBytes
[  5]   2.00-3.00   sec   799 MBytes  6.70 Gbits/sec    0    600 KBytes
[  5]   3.00-4.00   sec   798 MBytes  6.69 Gbits/sec    0    600 KBytes
[  5]   4.00-5.00   sec   892 MBytes  7.49 Gbits/sec    0    600 KBytes
[  5]   5.00-6.00   sec  1.10 GBytes  9.41 Gbits/sec    0    600 KBytes
[  5]   6.00-7.00   sec  1.10 GBytes  9.41 Gbits/sec    0    600 KBytes
[  5]   7.00-8.00   sec  1.09 GBytes  9.41 Gbits/sec    0    600 KBytes
[  5]   8.00-9.00   sec  1.10 GBytes  9.42 Gbits/sec    0    600 KBytes
[  5]   9.00-10.00  sec  1.10 GBytes  9.42 Gbits/sec    0    600 KBytes
[  5]  10.00-11.00  sec  1.10 GBytes  9.42 Gbits/sec    0    600 KBytes
[  5]  11.00-12.00  sec  1.10 GBytes  9.41 Gbits/sec    0    600 KBytes
[  5]  12.00-13.00  sec  1.10 GBytes  9.42 Gbits/sec    0    600 KBytes
[  5]  13.00-14.00  sec  1.09 GBytes  9.41 Gbits/sec    0    600 KBytes
[  5]  14.00-15.00  sec  1.10 GBytes  9.42 Gbits/sec    0    600 KBytes
[  5]  15.00-16.00  sec  1.10 GBytes  9.42 Gbits/sec    0    600 KBytes
[  5]  16.00-17.00  sec  1.10 GBytes  9.42 Gbits/sec    0    600 KBytes
[  5]  17.00-18.00  sec  1.10 GBytes  9.42 Gbits/sec    0    600 KBytes
[  5]  18.00-19.00  sec  1.04 GBytes  8.98 Gbits/sec    0   1021 KBytes
[  5]  19.00-20.00  sec  1.03 GBytes  8.82 Gbits/sec    0   1021 KBytes
[  5]  20.00-21.00  sec  1.03 GBytes  8.81 Gbits/sec    0   1021 KBytes
[  5]  21.00-22.00  sec  1.02 GBytes  8.80 Gbits/sec    0   1021 KBytes
[  5]  22.00-23.00  sec  1.03 GBytes  8.81 Gbits/sec    0   1021 KBytes
[  5]  23.00-24.00  sec  1.03 GBytes  8.81 Gbits/sec    0   1021 KBytes
[  5]  24.00-25.00  sec  1.03 GBytes  8.83 Gbits/sec    0   1021 KBytes
[  5]  25.00-26.00  sec  1.03 GBytes  8.82 Gbits/sec    0   1021 KBytes
[  5]  26.00-27.00  sec  1.03 GBytes  8.84 Gbits/sec    0   1021 KBytes
[  5]  27.00-28.00  sec  1.03 GBytes  8.82 Gbits/sec    0   1021 KBytes
[  5]  28.00-29.00  sec  1.03 GBytes  8.83 Gbits/sec    0   1021 KBytes
[  5]  29.00-30.00  sec  1.03 GBytes  8.83 Gbits/sec    0   1021 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-30.00  sec  30.6 GBytes  8.75 Gbits/sec    0             sender
[  5]   0.00-30.04  sec  30.6 GBytes  8.74 Gbits/sec                  receiver
```

Figure 12: iperf3 client side on Chron 3

At the maximum rate of 10 Gbps the results were a bit below regarding the previous reached bit rate, 8.75Gbps (see Figure 12). In the next step we shall try to redo the same tests done with iperf by indicating the bandwidth starting from 1-10 Gbps.

Required BW (Gbps	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Received BW (Gbps)	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	9.4

Conclusion: The problems we had using iperf when the bandwidth was indicated has disappeared, using iperf3 is strongly recommended when forcing a specific bandwidth as the iperf caused some errors that we were not able to fix.

I could not find the exact reasons to why the indicated Bandwidth caused problems with iperf and not with the iperf3 other than that iperf is multithreaded and iperf3 is single thread and it is probably the reason behind it. According to a blog on access agility [15]

"*iPerf2 was orphaned in the late 2000s at version 2.0.5, despite some known bugs and issues. After spending some time trying to fix iperf2's problems, ESnet decided by 2010 that a new, simpler tool was needed, and began development of iperf3.*

The goal was to make the tool as simple as possible, so others could contribute to the code base. For this reason, it was decided to make the tool single threaded, and not worry about backwards compatibility with iperf2.

Then in 2014, Bob (Robert) McMahon from Broadcom restarted development of iperf2. He fixed many of the problems with iperf2 and added several new features like iperf3. iperf2.0.8, released in 2015, made iperf2 a useful tool.

iPerf2's current development is focused on using UDP for latency testing, as well as broad platform support."

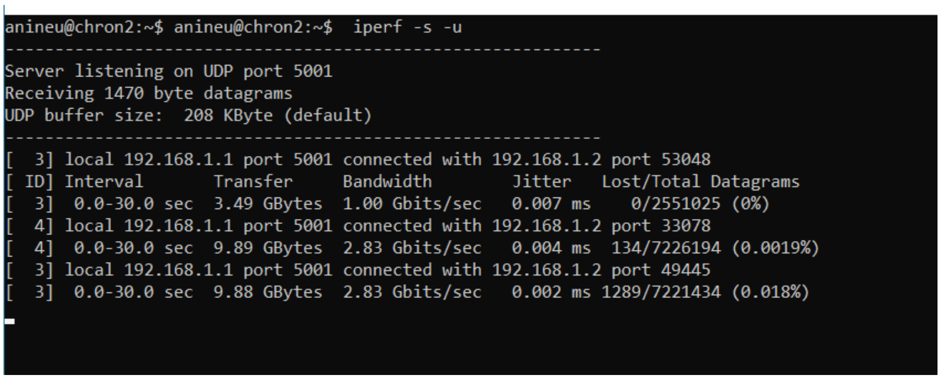
After the above quote it can be concluded that iperf2 is only being developed as a tool for UDP tests and our issue could be related to the TCP bugs that never got fixed.

3.2.2 ***UDP tests***

3.2.2.1 Iperf2

In this section we will be testing the behavior of the infrastructure when we use the UDP protocol. UDP is a connectionless oriented protocol so it's not only necessary to specify the server we are working on, but it is also mandatory to specify the bandwidth.

To start the test on the server side we input the following command: *iperf -s -u.* The *-u* is compulsory to let the client know that it only wants UDP packets. The command and output can be observed in Figure 13.



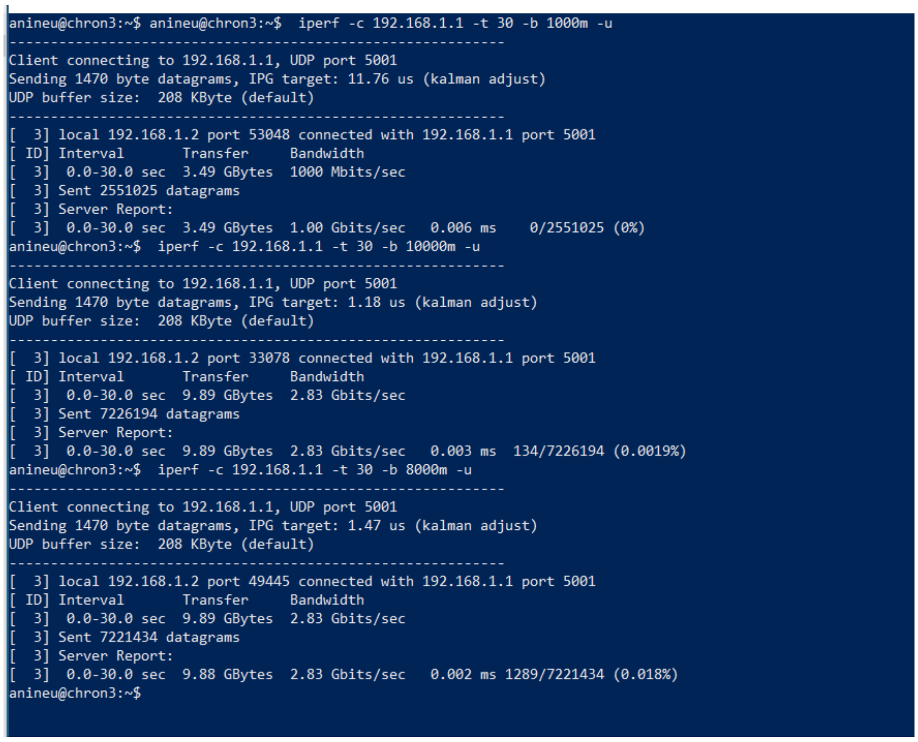
```
anineu@chron2:~$ anineu@chron2:~$  iperf -s -u
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 53048
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-30.0 sec  3.49 GBytes  1.00 Gbits/sec   0.007 ms    0/2551025 (0%)
[  4] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 33078
[  4]  0.0-30.0 sec  9.89 GBytes  2.83 Gbits/sec   0.004 ms  134/7226194 (0.0019%)
[  3] local 192.168.1.1 port 5001 connected with 192.168.1.2 port 49445
[  3]  0.0-30.0 sec  9.88 GBytes  2.83 Gbits/sec   0.002 ms 1289/7221434 (0.018%)
```

Figure 13 : starting iperf server for UDP

On the client side the following command is needed *iperf -c 192.168.1.1 -t 30 -b 1000m -u.* As it can be seen in the command for the UDP transfer we have to specify to the client directly that we want to throughput a specific Bandwidth.

In Figure 13 and 14 we can also see that in UDP mode the iperf returns jitter, packet loss along with the bandwidth.

One important parameter we now have that we did not before is the number of packets that are lost in communication. It is relevant to note that some websites advises to use TCP to measure the bandwidth while UDP is advised to use it to measure reliability, which means that when using TCP in iperf the bandwidth that is returned may not be very real, because it could be including forwarded packages.



```
anineu@chron3:~$ anineu@chron3:~$  iperf -c 192.168.1.1 -t 30 -b 1000m -u
------------------------------------------------------------
Client connecting to 192.168.1.1, UDP port 5001
Sending 1470 byte datagrams, IPG target: 11.76 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 53048 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  3.49 GBytes  1000 Mbits/sec
[  3] Sent 2551025 datagrams
[  3] Server Report:
[  3]  0.0-30.0 sec  3.49 GBytes  1.00 Gbits/sec   0.006 ms    0/2551025 (0%)
anineu@chron3:~$  iperf -c 192.168.1.1 -t 30 -b 10000m -u
------------------------------------------------------------
Client connecting to 192.168.1.1, UDP port 5001
Sending 1470 byte datagrams, IPG target: 1.18 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 33078 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  9.89 GBytes  2.83 Gbits/sec
[  3] Sent 7226194 datagrams
[  3] Server Report:
[  3]  0.0-30.0 sec  9.89 GBytes  2.83 Gbits/sec   0.003 ms  134/7226194 (0.0019%)
anineu@chron3:~$  iperf -c 192.168.1.1 -t 30 -b 8000m -u
------------------------------------------------------------
Client connecting to 192.168.1.1, UDP port 5001
Sending 1470 byte datagrams, IPG target: 1.47 us (kalman adjust)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.2 port 49445 connected with 192.168.1.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  9.89 GBytes  2.83 Gbits/sec
[  3] Sent 7221434 datagrams
[  3] Server Report:
[  3]  0.0-30.0 sec  9.88 GBytes  2.83 Gbits/sec   0.002 ms 1289/7221434 (0.018%)
anineu@chron3:~$
```

Figure 14 : Client-side outputs iperf with UDP

In the next step we test at different Bandwidth starting from 1 Gbps to 10 Gbps. The outputs can be observed in Figure 15.

Req BW	1	2	3	4	5	6	7	8	9	10
Rec BW	1	2	2.87	2.81	2.87	2.89	2.86	2.86	2.89	2.76
Data loss rate %	0.001	0.001	0.019	0.013	0.013	0.013	0.013	0.013	0.013	0.013

Figure 15 : Output UDP 1-10Gbps

Even though the packet loss stays stable around 0.013% the Bandwidth never gets higher than 2.9 Gbps. It is an improvement compared to the TCP where it went to 0 multiple times but the reason behind the 2.9 Gbps cap needs to be researched further.

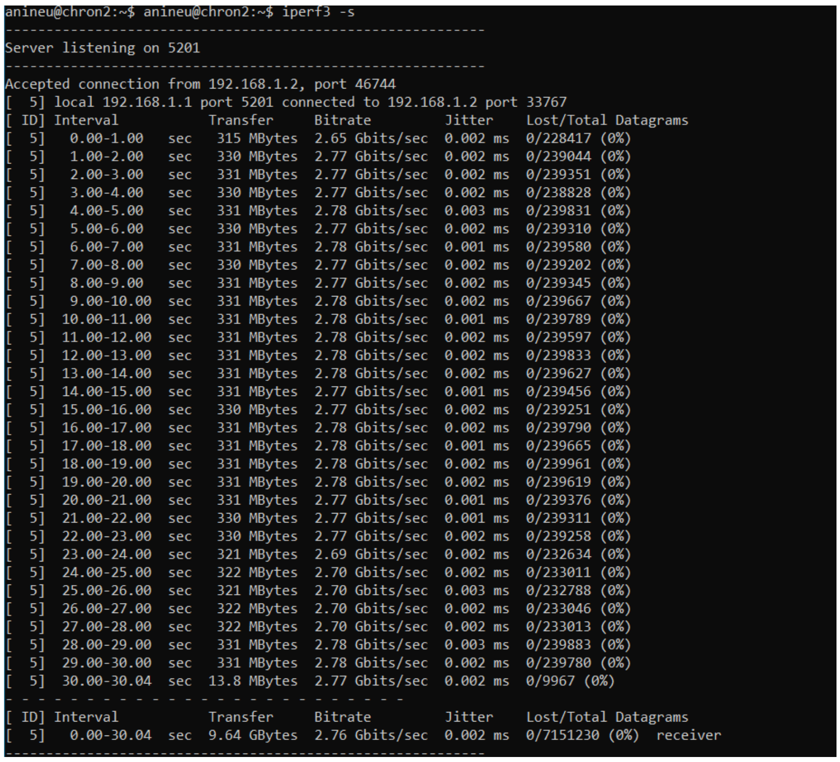
3.2.2.2 Iperf3

An important difference between iperf3 and iperf with UDP is that the *-u* option is not necessary on the server. A basic configuration shall remain the same:

Server: *iperf3 -s*

Client: *iperf3 -c 192.168.1.1 -t 30 -b 0 -u*

As seen in the commands we use the option ‘b 0’ which means the iperf3 will check for the maximum available bandwidth.



```
anineu@chron2:~$ anineu@chron2:~$ iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 192.168.1.2, port 46744
[  5] local 192.168.1.1 port 5201 connected to 192.168.1.2 port 33767
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-1.00   sec   315 MBytes  2.65 Gbits/sec  0.002 ms  0/228417 (0%)
[  5]   1.00-2.00   sec   330 MBytes  2.77 Gbits/sec  0.002 ms  0/239044 (0%)
[  5]   2.00-3.00   sec   331 MBytes  2.77 Gbits/sec  0.002 ms  0/239351 (0%)
[  5]   3.00-4.00   sec   330 MBytes  2.77 Gbits/sec  0.002 ms  0/238828 (0%)
[  5]   4.00-5.00   sec   331 MBytes  2.78 Gbits/sec  0.003 ms  0/239831 (0%)
[  5]   5.00-6.00   sec   330 MBytes  2.77 Gbits/sec  0.002 ms  0/239310 (0%)
[  5]   6.00-7.00   sec   331 MBytes  2.78 Gbits/sec  0.001 ms  0/239580 (0%)
[  5]   7.00-8.00   sec   330 MBytes  2.77 Gbits/sec  0.002 ms  0/239202 (0%)
[  5]   8.00-9.00   sec   331 MBytes  2.77 Gbits/sec  0.002 ms  0/239345 (0%)
[  5]   9.00-10.00  sec   331 MBytes  2.78 Gbits/sec  0.002 ms  0/239667 (0%)
[  5]  10.00-11.00  sec   331 MBytes  2.78 Gbits/sec  0.001 ms  0/239789 (0%)
[  5]  11.00-12.00  sec   331 MBytes  2.78 Gbits/sec  0.002 ms  0/239597 (0%)
[  5]  12.00-13.00  sec   331 MBytes  2.78 Gbits/sec  0.002 ms  0/239833 (0%)
[  5]  13.00-14.00  sec   331 MBytes  2.78 Gbits/sec  0.002 ms  0/239627 (0%)
[  5]  14.00-15.00  sec   331 MBytes  2.77 Gbits/sec  0.001 ms  0/239456 (0%)
[  5]  15.00-16.00  sec   330 MBytes  2.77 Gbits/sec  0.002 ms  0/239251 (0%)
[  5]  16.00-17.00  sec   331 MBytes  2.78 Gbits/sec  0.002 ms  0/239790 (0%)
[  5]  17.00-18.00  sec   331 MBytes  2.78 Gbits/sec  0.001 ms  0/239665 (0%)
[  5]  18.00-19.00  sec   331 MBytes  2.78 Gbits/sec  0.002 ms  0/239961 (0%)
[  5]  19.00-20.00  sec   331 MBytes  2.78 Gbits/sec  0.002 ms  0/239619 (0%)
[  5]  20.00-21.00  sec   331 MBytes  2.77 Gbits/sec  0.001 ms  0/239376 (0%)
[  5]  21.00-22.00  sec   330 MBytes  2.77 Gbits/sec  0.001 ms  0/239311 (0%)
[  5]  22.00-23.00  sec   330 MBytes  2.77 Gbits/sec  0.002 ms  0/239258 (0%)
[  5]  23.00-24.00  sec   321 MBytes  2.69 Gbits/sec  0.002 ms  0/232634 (0%)
[  5]  24.00-25.00  sec   322 MBytes  2.70 Gbits/sec  0.002 ms  0/233011 (0%)
[  5]  25.00-26.00  sec   321 MBytes  2.70 Gbits/sec  0.003 ms  0/232788 (0%)
[  5]  26.00-27.00  sec   322 MBytes  2.70 Gbits/sec  0.002 ms  0/233046 (0%)
[  5]  27.00-28.00  sec   322 MBytes  2.70 Gbits/sec  0.002 ms  0/233013 (0%)
[  5]  28.00-29.00  sec   331 MBytes  2.78 Gbits/sec  0.003 ms  0/239883 (0%)
[  5]  29.00-30.00  sec   331 MBytes  2.78 Gbits/sec  0.002 ms  0/239780 (0%)
[  5]  30.00-30.04  sec  13.8 MBytes  2.77 Gbits/sec  0.002 ms  0/9967 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-30.04  sec  9.64 GBytes  2.76 Gbits/sec  0.002 ms  0/7151230 (0%)  receiver
-----------------------------------------------------------
```

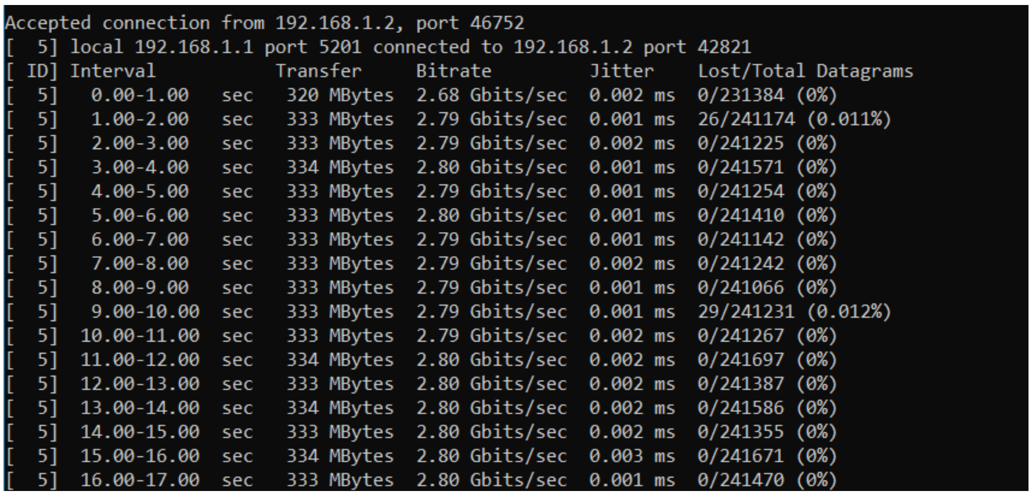
Figure 16 : Iperf3 with UDP : maximum rate

There was no packet-loss with iperf3 as it can be seen in figure 16 but the bitrate is still capped at 2.8 Gbps. In the next steps we will try some measures to improve the performance of the UDP.

3.2.2.3 Improve the UDP performances.

Solution 1: *Try to add a --udp-counters-64bit along with the client-side command*:

It makes iperf3 use 64-bit sequence numbers in the test packets rather than 32-bit sequence numbers. This can be useful in high-packet-rate or long-running tests, which can overflow a 32-bit sequence number space. There is no effect on TCP or SCTP tests. I do not believe there would be any significant performance effect [16].



```
Accepted connection from 192.168.1.2, port 46752
[  5] local 192.168.1.1 port 5201 connected to 192.168.1.2 port 42821
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-1.00   sec   320 MBytes  2.68 Gbits/sec  0.002 ms  0/231384 (0%)
[  5]   1.00-2.00   sec   333 MBytes  2.79 Gbits/sec  0.001 ms  26/241174 (0.011%)
[  5]   2.00-3.00   sec   333 MBytes  2.79 Gbits/sec  0.002 ms  0/241225 (0%)
[  5]   3.00-4.00   sec   334 MBytes  2.80 Gbits/sec  0.001 ms  0/241571 (0%)
[  5]   4.00-5.00   sec   333 MBytes  2.79 Gbits/sec  0.001 ms  0/241254 (0%)
[  5]   5.00-6.00   sec   333 MBytes  2.80 Gbits/sec  0.001 ms  0/241410 (0%)
[  5]   6.00-7.00   sec   333 MBytes  2.79 Gbits/sec  0.001 ms  0/241142 (0%)
[  5]   7.00-8.00   sec   333 MBytes  2.79 Gbits/sec  0.002 ms  0/241242 (0%)
[  5]   8.00-9.00   sec   333 MBytes  2.79 Gbits/sec  0.001 ms  0/241066 (0%)
[  5]   9.00-10.00  sec   333 MBytes  2.79 Gbits/sec  0.001 ms  29/241231 (0.012%)
[  5]  10.00-11.00  sec   333 MBytes  2.79 Gbits/sec  0.002 ms  0/241267 (0%)
[  5]  11.00-12.00  sec   334 MBytes  2.80 Gbits/sec  0.002 ms  0/241697 (0%)
[  5]  12.00-13.00  sec   333 MBytes  2.80 Gbits/sec  0.002 ms  0/241387 (0%)
[  5]  13.00-14.00  sec   334 MBytes  2.80 Gbits/sec  0.002 ms  0/241586 (0%)
[  5]  14.00-15.00  sec   333 MBytes  2.80 Gbits/sec  0.002 ms  0/241355 (0%)
[  5]  15.00-16.00  sec   334 MBytes  2.80 Gbits/sec  0.003 ms  0/241671 (0%)
[  5]  16.00-17.00  sec   333 MBytes  2.80 Gbits/sec  0.001 ms  0/241470 (0%)
```

Figure 17 : UDP test with idp-counters-64

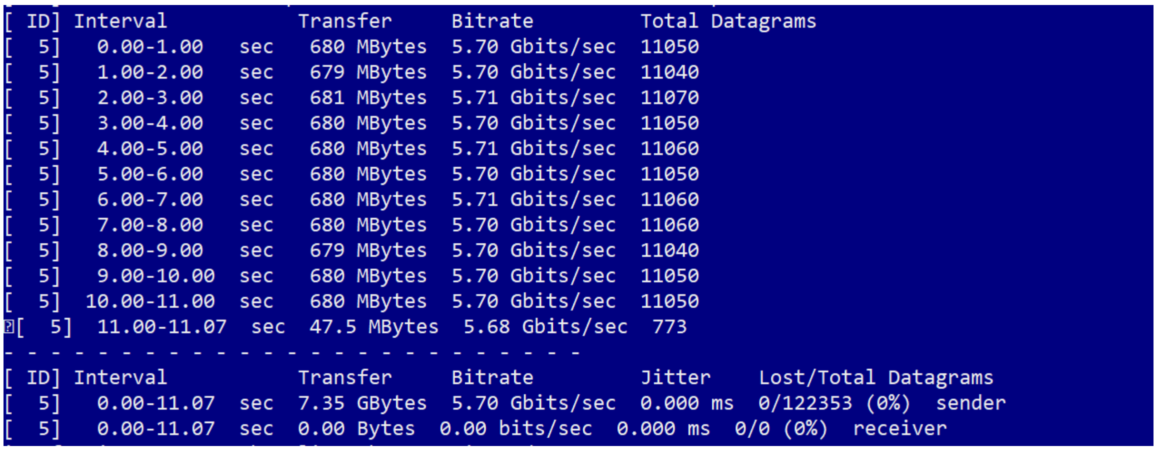
As we can see in Figure 17, this did not make any difference. I suspect that it is running the 64bit feature by default.

Solution 2: *Increase the chunk length -l:*

We run the command *iperf3 -l 63k -c 192.168.1.1 -t 30-b 0- u*

The command remains almost the same. We added two new parameters:

-l 63k: The chunk length refers to the length of buffers to read or write. Iperf works by writing an array of len bytes several times. Default is 128 KB for TCP, 8 KB for UDP and by increasing this from 8K to its maximum value of 63K we can increase our performance [16]. As we can observe in Figure 18 the performance has improved to 5.70Gbps.



```
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  5]   0.00-1.00   sec   680 MBytes  5.70 Gbits/sec  11050
[  5]   1.00-2.00   sec   679 MBytes  5.70 Gbits/sec  11040
[  5]   2.00-3.00   sec   681 MBytes  5.71 Gbits/sec  11070
[  5]   3.00-4.00   sec   680 MBytes  5.70 Gbits/sec  11050
[  5]   4.00-5.00   sec   680 MBytes  5.71 Gbits/sec  11060
[  5]   5.00-6.00   sec   680 MBytes  5.70 Gbits/sec  11050
[  5]   6.00-7.00   sec   680 MBytes  5.71 Gbits/sec  11060
[  5]   7.00-8.00   sec   680 MBytes  5.70 Gbits/sec  11060
[  5]   8.00-9.00   sec   679 MBytes  5.70 Gbits/sec  11040
[  5]   9.00-10.00  sec   680 MBytes  5.70 Gbits/sec  11050
[  5]  10.00-11.00  sec   680 MBytes  5.70 Gbits/sec  11050
[  5]  11.00-11.07  sec  47.5 MBytes  5.68 Gbits/sec  773
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-11.07  sec  7.35 GBytes  5.70 Gbits/sec  0.000 ms  0/122353 (0%)  sender
[  5]   0.00-11.07  sec  0.00 Bytes  0.00 bits/sec  0.000 ms  0/0 (0%)  receiver
```

Figure 18 : UDP test with chunk length increased. Chunk

After these tests we can conclude that it is better to move forwards with iperf3 and increase the chunk length during the UDP tests.

3.3 Docker introduction and initial configurations

Developing apps today requires so much more than writing code. Multiple languages, frameworks, architectures, and discontinuous interfaces between tools for each lifecycle stage creates enormous complexity. Docker simplifies and accelerates the workflow, while giving developers the freedom to innovate with their choice of tools, application stacks, and deployment environments for each project [17].

The docker application has already been installed on the servers (Chron 2 and Chron 3) where we plan to run several tests. We will not document the steps required to install docker in a server, but Docker provides a perfectly described instructions to install it on its official page [17].

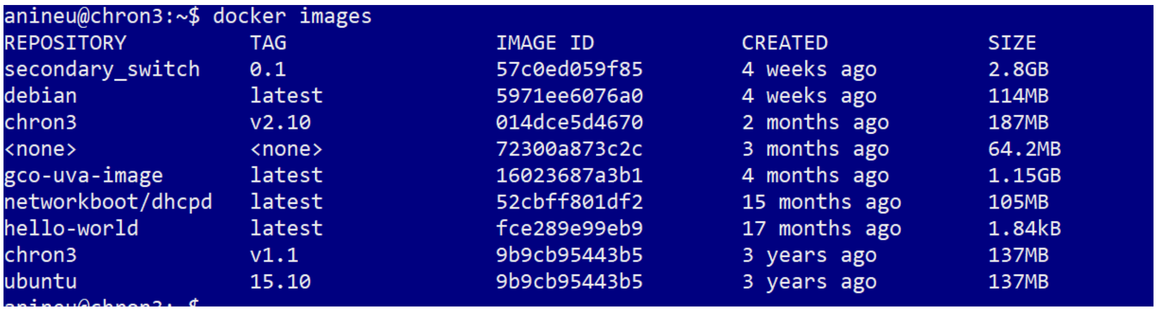
After the introduction of the docker technology in Chapter 2, we understand the concept of images and containers. We will be using the pre-built images from the Docker Hub and the configurations and tests will be more focused around the containers.

3.3.1.1 Getting familiar with Docker

Before starting there are few commands which will be used more than others during this project.

- *docker images* lists the available docker images in the registry of the machine.
- *docker run <image-name>*: runs the image to start a container.
- *docker kill <container-name>*: stops a running container
- *docker commit <container-name> <new-image-name>*: saves the made changes inside a container to a new docker image
- *docker rmi <image-name>:* removes the unwanted images.

If the docker has been installed correctly, the docker registry on the machine where the docker has been installed contains a basic ubuntu image in its images list. We will use this to test our initial settings. This can also be observed in Figure 19.



```
anineu@chron3:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
secondary_switch    0.1                 57c0ed059f85        4 weeks ago         2.8GB
debian              latest              5971ee6076a0        4 weeks ago         114MB
chron3              v2.10               014dce5d4670        2 months ago        187MB
<none>              <none>              72300a873c2c        3 months ago        64.2MB
gco-uva-image       latest              16023687a3b1        4 months ago        1.15GB
networkboot/dhcpd   latest              52cbff801df2        15 months ago       105MB
hello-world         latest              fce289e99eb9        17 months ago       1.84kB
chron3              v1.1                9b9cb95443b5        3 years ago         137MB
ubuntu              15.10               9b9cb95443b5        3 years ago         137MB
```

Figure 19 :output of command docker images

First, we rename the ubuntu image. This is done with the command docker tag <image-name:tag> <new-image-name:tag>, for example, docker tag ubuntu:15.10 chron3:v1.1.

Note: the tag just means the version of the image, when the image gets updated, we will give the new image the same name with different tag i.e. v1.2.

After the rename, the old image can be deleted with command *docker rmi <image-name:tag>*. We do not delete the base ubuntu image, but this command will be used later in the project to delete the older versions of the images.

To start a container from the image we run the following command *docker run -it <image-name:tag>*, in our case this will be : *docker run -it chron3:v1.1*.

The *-it* short for interactive instructs Docker to allocate a pseudo-TTY connected to the container’s stdin; creating an interactive bash shell in the container. It lets us run commands like ls, mkdir inside the container.

Optional: we can also give the container a specific name during the start by adding *--name <container-name>* to the above command.

With this step we have started our very first container. Unlike traditional virtual machines docker container has no resource constraints and can use as much of a given resource as the host’s kernel scheduler allows. Docker can also enforce hard memory limits, which allows the container to use no more than a given amount of user or system memory. For more information regarding memory allocation inside a docker container, we refer to the official docker pages [17].

Once the first container has been started, we need to install a few necessary packages the base image lacks. This is done with the command [18]:

apt-get install -y net-tools nano iproute2 iputils-ping ifupdown iperf iperf3

These packages are only necessary for the initial tests, more packages like dhcp-server vlans will be added in the future when needed.

Our initial plan is to repeat the same iperf3 tests (earlier done on the physical server) inside the containers. We will look if containerizing makes any difference on the throughputs.

3.3.1.2 Iperf tests inside docker containers

First, we start two containers from the same image by executing the same commands twice.

docker run -it chron3:v1.1

Upon executing the commands twice on different CLI's (PowerShell or CMD), two containers are created on the docker in the same server (CHRON 3). Both will receive different IP address from the same default "docker0" network (172.17.0.0/16). The topology can be visualized in Figure 20 and the Ip's can be observed in Figure 21 (Container 1) and Figure 22 (Container 2).

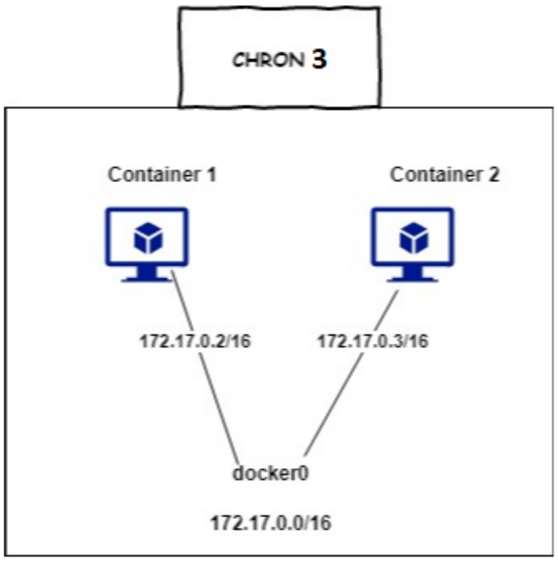
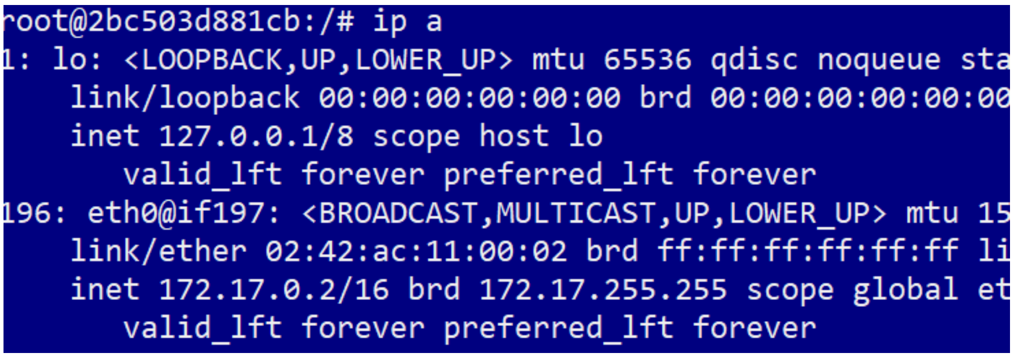
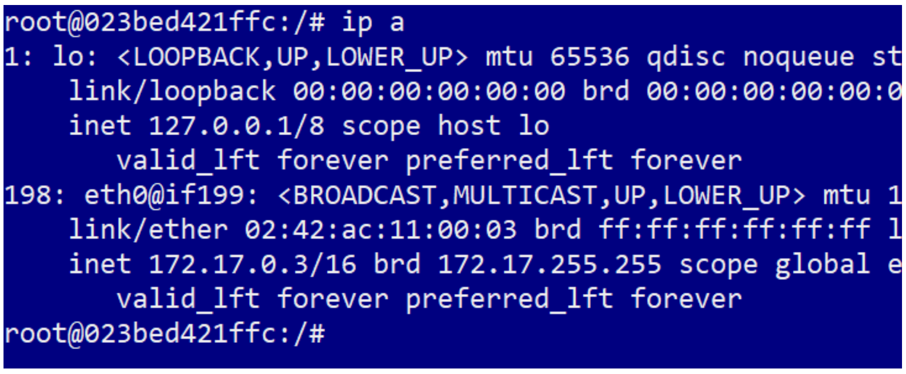


Figure 20 : docker0 network with 2 containers



```
root@2bc503d881cb:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue sta
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
196: eth0@if197: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 15
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff li
    inet 172.17.0.2/16 brd 172.17.255.255 scope global et
       valid_lft forever preferred_lft forever
```

Figure 21 :Container one with docker0 network



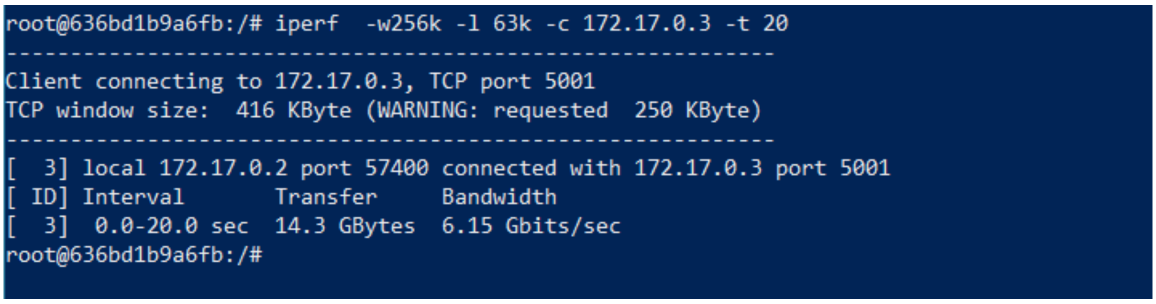
```
root@023bed421ffc:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue st
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:0
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
198: eth0@if199: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff l
    inet 172.17.0.3/16 brd 172.17.255.255 scope global e
       valid_lft forever preferred_lft forever
root@023bed421ffc:/#
```

Figure 22 : Container 2 with docker0 network

Morover, in the next step we test the throughput in the containers using iperf.

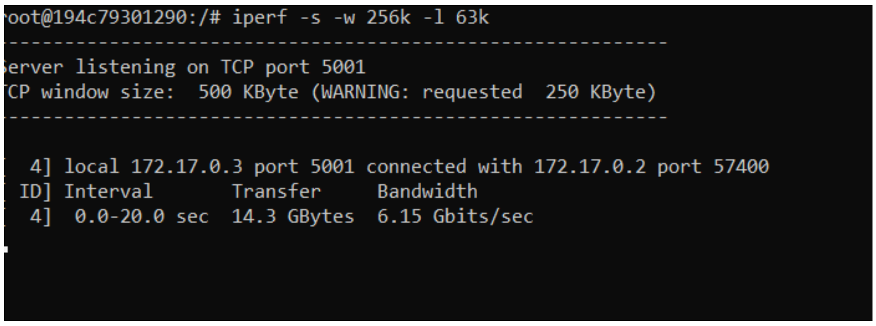
TCP tests with iperf2

The methodology behind the iperf tests remains the same. Figure 23 (Client: Container 1) and Figure 24 (Server: Container 2) show the client side and server side of the tests respectively.



```
root@636bd1b9a6fb:/# iperf  -w256k -l 63k -c 172.17.0.3 -t 20
------------------------------------------------------------
Client connecting to 172.17.0.3, TCP port 5001
TCP window size:  416 KByte (WARNING: requested  250 KByte)
------------------------------------------------------------
[  3] local 172.17.0.2 port 57400 connected with 172.17.0.3 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-20.0 sec  14.3 GBytes  6.15 Gbits/sec
root@636bd1b9a6fb:/#
```

Figure 23 :Client side of the iperf inside a container



```
oot@194c79301290:/# iperf -s -w 256k -l 63k
------------------------------------------------------------
erver listening on TCP port 5001
CP window size:  500 KByte (WARNING: requested  250 KByte)
------------------------------------------------------------

  4] local 172.17.0.3 port 5001 connected with 172.17.0.2 port 57400
 ID] Interval       Transfer     Bandwidth
  4]  0.0-20.0 sec  14.3 GBytes  6.15 Gbits/sec
```

Figure 24 :Server side of the iperf inside a container

At the maximum available rate, we get 6.15 Gbps throughput. This is significantly lower to the test done on the physical servers.

Next we force the TCP to certain bandwidth starting from 1-10Gbps. The results received on the containers when the iperf client indicates exactly how much it wants to send is identical to the results received on the Physical Server. The same issues persist, the bandwidth received after 2 Gbps is not good.

Request Bandwidth	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Rececived Bandwidth	1.0	2.0	0.0	0.0	0.7	1.7	0.0	0.0	0.4	1.4

In the next step we test if the errors persist with Iperf3 as well.

TCP tests with IPERF3

The commands to start a client and the servers remain identical. At the maximum available rate, the BW is capped at 6.58 Gbps and when the bandwidth is specified the results are as follows:

Request Bandwidth	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Rececived Bandwidth	1.0	2.0	3.0	4.0	5.0	6.0	6.2	6.2	6.2	6.3

The results are capped around 6.3 Gbps. The docker bridge seems to be losing throughput of about 3Gbps when compared to the tests done on the physical servers.

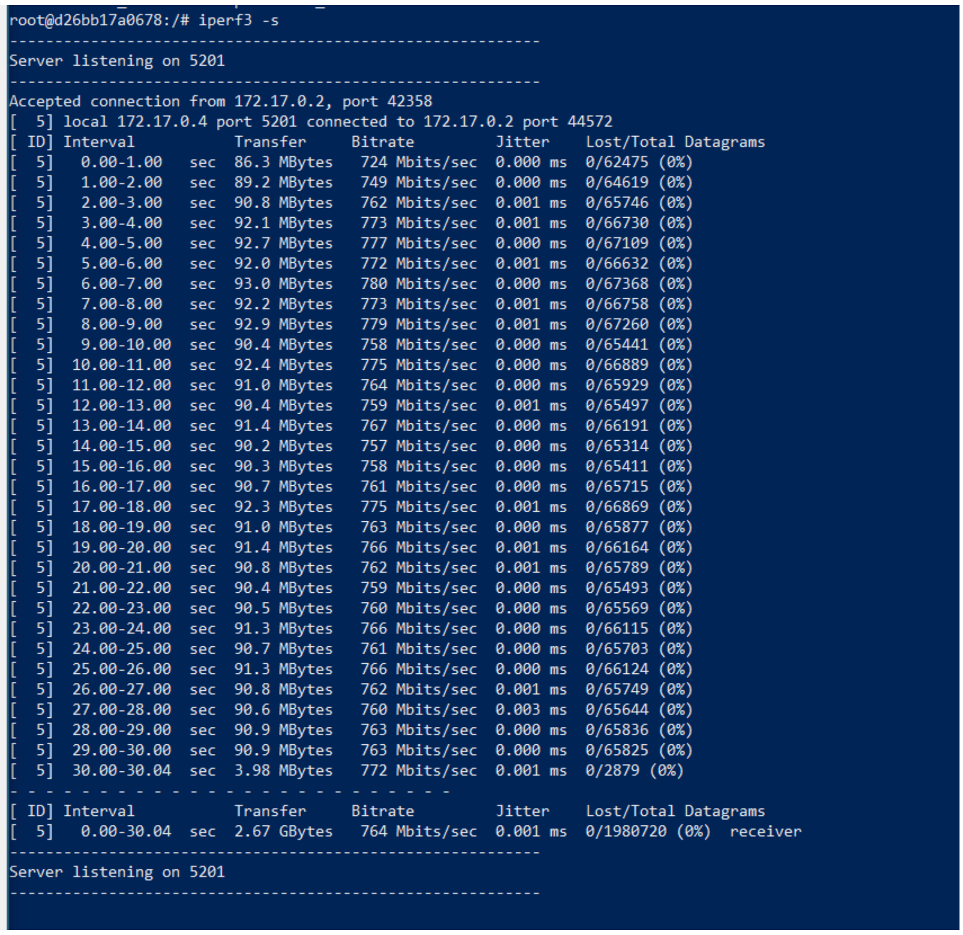
UDP tests with iperf3

A basic configuration shall remain the same:

On the Client: *iperf3 -c 172.17.0.4 -t 30 -b 0*

On the Server : *iperf3 -s*

From the result that we can see in Figure 25, that the UDP through put is much worse than TCP. It is stuck at 760 Mbps.

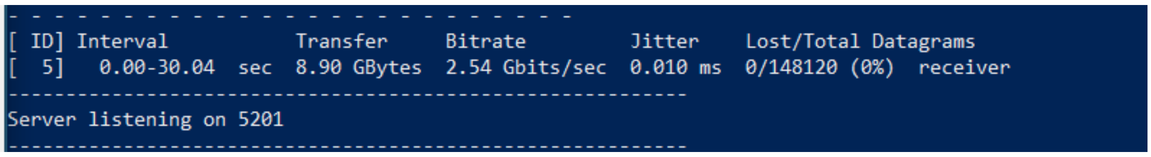


```
root@d26bb17a0678:/# iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 172.17.0.2, port 42358
[  5] local 172.17.0.4 port 5201 connected to 172.17.0.2 port 44572
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-1.00   sec  86.3 MBytes   724 Mbits/sec  0.000 ms  0/62475 (0%)
[  5]   1.00-2.00   sec  89.2 MBytes   749 Mbits/sec  0.000 ms  0/64619 (0%)
[  5]   2.00-3.00   sec  90.8 MBytes   762 Mbits/sec  0.001 ms  0/65746 (0%)
[  5]   3.00-4.00   sec  92.1 MBytes   773 Mbits/sec  0.001 ms  0/66730 (0%)
[  5]   4.00-5.00   sec  92.7 MBytes   777 Mbits/sec  0.000 ms  0/67109 (0%)
[  5]   5.00-6.00   sec  92.0 MBytes   772 Mbits/sec  0.001 ms  0/66632 (0%)
[  5]   6.00-7.00   sec  93.0 MBytes   780 Mbits/sec  0.000 ms  0/67368 (0%)
[  5]   7.00-8.00   sec  92.2 MBytes   773 Mbits/sec  0.001 ms  0/66758 (0%)
[  5]   8.00-9.00   sec  92.9 MBytes   779 Mbits/sec  0.001 ms  0/67260 (0%)
[  5]   9.00-10.00  sec  90.4 MBytes   758 Mbits/sec  0.000 ms  0/65441 (0%)
[  5]  10.00-11.00  sec  92.4 MBytes   775 Mbits/sec  0.000 ms  0/66889 (0%)
[  5]  11.00-12.00  sec  91.0 MBytes   764 Mbits/sec  0.000 ms  0/65929 (0%)
[  5]  12.00-13.00  sec  90.4 MBytes   759 Mbits/sec  0.001 ms  0/65497 (0%)
[  5]  13.00-14.00  sec  91.4 MBytes   767 Mbits/sec  0.000 ms  0/66191 (0%)
[  5]  14.00-15.00  sec  90.2 MBytes   757 Mbits/sec  0.000 ms  0/65314 (0%)
[  5]  15.00-16.00  sec  90.3 MBytes   758 Mbits/sec  0.000 ms  0/65411 (0%)
[  5]  16.00-17.00  sec  90.7 MBytes   761 Mbits/sec  0.000 ms  0/65715 (0%)
[  5]  17.00-18.00  sec  92.3 MBytes   775 Mbits/sec  0.001 ms  0/66869 (0%)
[  5]  18.00-19.00  sec  91.0 MBytes   763 Mbits/sec  0.000 ms  0/65877 (0%)
[  5]  19.00-20.00  sec  91.4 MBytes   766 Mbits/sec  0.001 ms  0/66164 (0%)
[  5]  20.00-21.00  sec  90.8 MBytes   762 Mbits/sec  0.001 ms  0/65789 (0%)
[  5]  21.00-22.00  sec  90.4 MBytes   759 Mbits/sec  0.000 ms  0/65493 (0%)
[  5]  22.00-23.00  sec  90.5 MBytes   760 Mbits/sec  0.000 ms  0/65569 (0%)
[  5]  23.00-24.00  sec  91.3 MBytes   766 Mbits/sec  0.000 ms  0/66115 (0%)
[  5]  24.00-25.00  sec  90.7 MBytes   761 Mbits/sec  0.000 ms  0/65703 (0%)
[  5]  25.00-26.00  sec  91.3 MBytes   766 Mbits/sec  0.000 ms  0/66124 (0%)
[  5]  26.00-27.00  sec  90.8 MBytes   762 Mbits/sec  0.001 ms  0/65749 (0%)
[  5]  27.00-28.00  sec  90.6 MBytes   760 Mbits/sec  0.003 ms  0/65644 (0%)
[  5]  28.00-29.00  sec  90.9 MBytes   763 Mbits/sec  0.000 ms  0/65836 (0%)
[  5]  29.00-30.00  sec  90.9 MBytes   763 Mbits/sec  0.000 ms  0/65825 (0%)
[  5]  30.00-30.04  sec  3.98 MBytes   772 Mbits/sec  0.001 ms  0/2879 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-30.04  sec  2.67 GBytes   764 Mbits/sec  0.001 ms  0/1980720 (0%)  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 25: Iperf3 result with UDP (containers)

In the next step we apply the same parameters we applied we applied during the first tests to improve the performance i.e. increase the clunk length value “-l”:

iperf3 -c 172.17.0.4 -t 30 -b 0 -u -l 63k



```
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  5]   0.00-30.04  sec  8.90 GBytes  2.54 Gbits/sec  0.010 ms  0/148120 (0%)  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

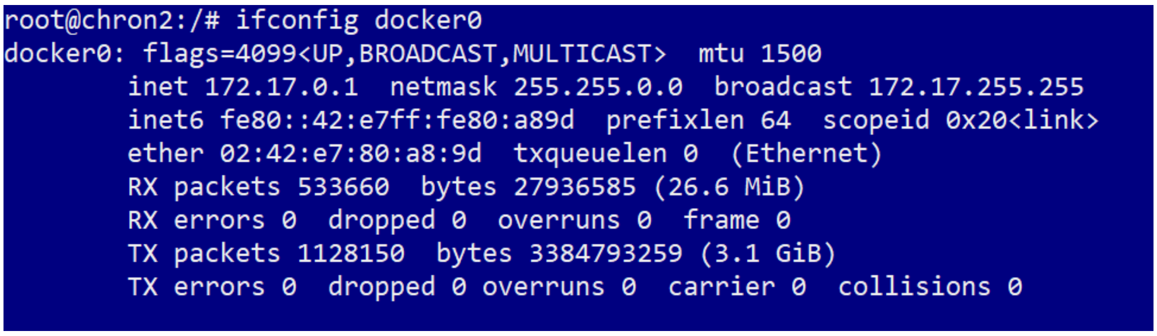
Figure 26 : UDP test with Iperf3 with -l parameter

The results of Figure 26 shows that the throughput improved by 300% and it is now capped at 2.54Gbps. After talking with the mentor, we decided that the bridge networking is not the best option for the project as the loss in throughput is too huge for the final test, and we research further into Docker Networking for an alternative.

We found out that bridge networking doesn’t allow conenctions between the containers on the different hosts. If we want to run iperf tests on two containers on different hosts (Chron2 and Chron 3), we must run the containers on host networking mode so they have connection to each other, but running them on the host mode also means that we are just using the physical interfaces, the iperf tests on the physical interfaces have been done earlier (section 3.2) and the results would be exactly (were) the same, we found no reasons to include them in this report.

3.3.1.3 Docker Networking

When docker is installed, if the network interfaces are checked, we can see that a new interface has been created called “docker0” as it can also be seen in Figure 27.



```
root@chron2:/# ifconfig docker0
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:e7ff:fe80:a89d  prefixlen 64  scopeid 0x20<link>
        ether 02:42:e7:80:a8:9d  txqueuelen 0  (Ethernet)
        RX packets 533660  bytes 27936585 (26.6 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1128150  bytes 3384793259 (3.1 GiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 27 : docker0 default docker bridge

This is because Docker, when installed, creates a Linux bridge to which it by default connects all containers that are created. This bridge behaves like a multi-level traditional switch in which a NAT address translation is performed between the Docker virtual network (default 172.17.0.0/16) and the network your host is connected to. Figure 28 shows how the network model works when the containers are connected to each other in a docker.

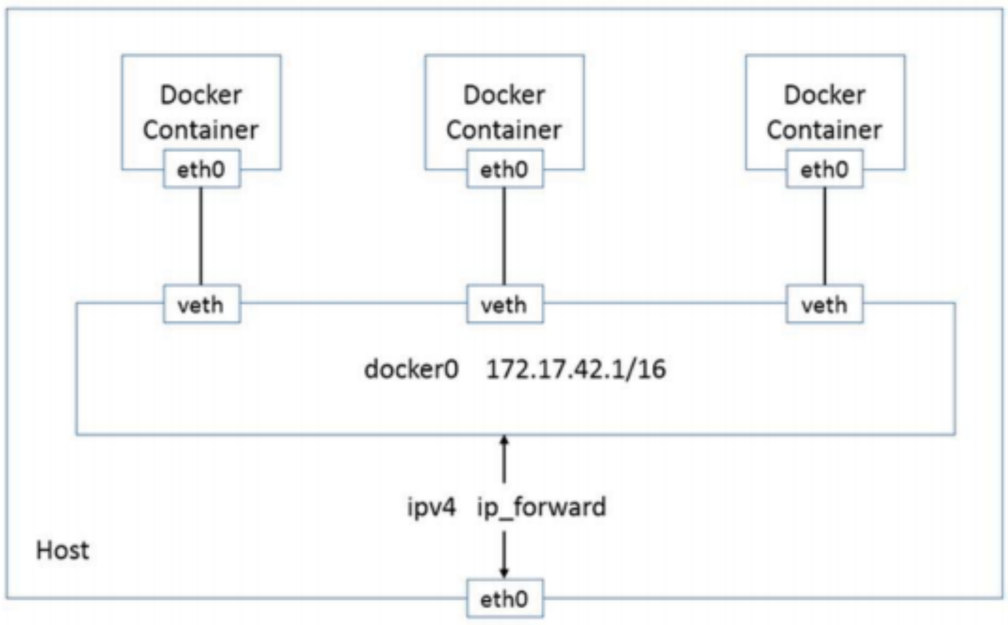


Figure 28 : Default network model of Docker containers

In this way, the containers can connect to the Internet, but are not exposed. This is because the IP to which external traffic is routed is that of your host machine. To understand this concept better, you can imagine a private network with a router that does NAT with Internet. Devices on the internal network can access the Internet thanks to the public IP from the router, but the Internet is not able to see the devices within the network, only the router. In this case something similar happens, the private network is the Docker network and the public IP is the one assigned to the host interface (which, in most cases, is a private NA, so NAT is done twice, once to translate between the Docker network and the private network on the host, and another on the router of that private network to a public address from Internet) [17].

If you want to expose services, you need to tell the container which ports you want to publish/expose so that when packets destined for those ports reach the host, it produces a redirect to the container. We will see later when running the SDN controller inside a container, how this redirection is performed. For example: To run a SDN controller inside a container with a host port published we add: *-p <host-ports-ip>:8181* along the *docker run* commands to get access to the controller on the browser (chrome, Firefox etc.).

When a container is created without specifying any specific network information, it automatically receives an IP from the docker0 network as we saw earlier during the Iperf tests. This also means that the containers created on Chron2 are isolated from Chron3 when the docker0 network is being used.

In the context of this project, the goal is for an SDN controller to oversee managing the network, so it is not very useful for Docker to oversee creating the switches and connecting the virtual machines. The goal is for the OpenFlow switches to be in charge of routing traffic through controller orchestration. Therefore, Open vSwitch (OVS) switches need to be added somewhere in the topology. Therefore, Open vSwitch (OVS) switches should be configured with some other network configuration that is not a docker bridge.

From the above research we can conclude that the docker-bridge network can only be used for the SDN controller, and a webserver if we build/need one, we will create our own docker-bridge network instead of using the docker0 in the later steps as the default bridge has some limitations that a user created bridge wimm not have [19].

We need to look for an alternative if we want to run the OVS and a router inside a container as it does not seem possible with bridge networking. In comparison to a tradition virtual machines where NAT is used, docker uses bridge networking.

A bridge provides a host internal network in which containers on the same host may communicate, but the IP addresses assigned to each container are not accessible from outside the host. Bridge networking leverages iptables for NAT and port-mapping, which provide single-host networking. Bridge networking is the default Docker network type (i.e., docker0), where one end of a virtual network interface pair is connected between the bridge and the container whereas in a virtual machine hardware level virtualisations takes place which also means the network components are virtualized so the virtual machines also receive the IP address from the same network as the host, this is not possible inside a docker unless we use the host networking.

At this point only other solution available seems to be host networking. If we use the host network mode for a container, that container's network stack is not isolated from the Docker host (the container shares the host's networking namespace), and the container does not get its own IP-address allocated. For instance, if you run a container which binds to port 80 and you use host networking, the container's application is available on port 80 on the host's IP address. This also means we do not need to run the iperf tests again as the container is just using the host's network and the results would be identical to the tests done on the physical servers. To start a container in host mode we simply add *--network host* along with the docker run command [17].

Running a container in host modes womes with risk, if the user is not careful the changes made inside the container regarding the networking can effect the host.As we are using the host interface, changing iptales, iproute, default routes means we are also adjusting the same thing on the physical servers. It is recommended to be careful during these steps if the project is being run on a device which is also used for other purposes in the lab.

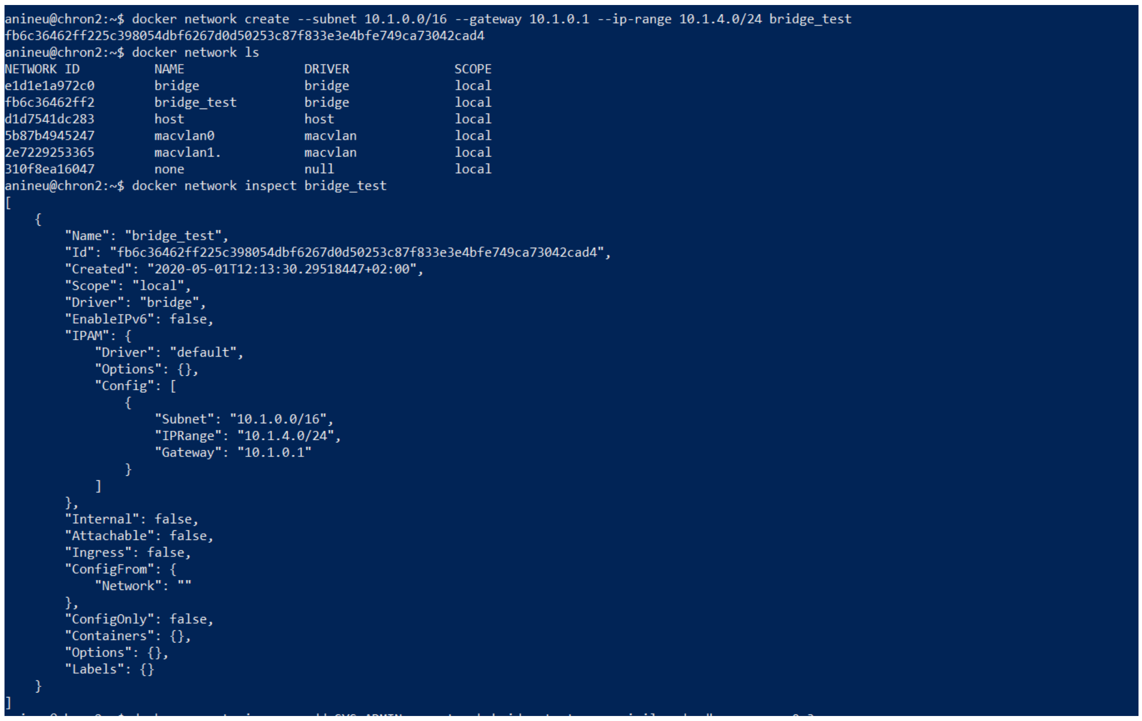
So, we will be creating our own docker bridge that will be used on the containers for SDN controllers and we shall use host networking for all the other containers that need to connect to a different host.

To create a docker bridge of our own we follow the instructions provided by the official docker pages [17]:

docker network create --subnet 10.1.0.0/16 --gateway 10.1.0.1 --ip-range 10.1.4.0/24 bridge_test

The above command creates our own docker-bridge network named *bridge_test* and the rest of the parameters on the bridge are self-explanatory. To start a container using this bridge networking we just add *--network bridge_test* along with the *docker run* command when we start the container. Just like the "docker0" bridge, this *bridge_test* has automatic configured connection to all interfaces on the device and the internet.

To check if the bridge has been created correctly, we can type the command *docker network inspect bridge_test* as shown in Figure 29.



```
anineu@chron2:~$ docker network create --subnet 10.1.0.0/16 --gateway 10.1.0.1 --ip-range 10.1.4.0/24 bridge_test
fb6c36462ff225c398054dbf6267d0d50253c87f833e3e4bfe749ca73042cad4
anineu@chron2:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
e1d1e1a972c0        bridge              bridge              local
fb6c36462ff2        bridge_test         bridge              local
d1d7541dc283        host                host                local
5b87b4945247        macvlan0            macvlan             local
2e7229253365        macvlan1.           macvlan             local
310f8ea16047        none                null                local
anineu@chron2:~$ docker network inspect bridge_test
[
    {
        "Name": "bridge_test",
        "Id": "fb6c36462ff225c398054dbf6267d0d50253c87f833e3e4bfe749ca73042cad4",
        "Created": "2020-05-01T12:13:30.29518447+02:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "10.1.0.0/16",
                    "IPRange": "10.1.4.0/24",
                    "Gateway": "10.1.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Figure 29 : docker bridge bridge_test

3.4 Conclusion

In this chapter, we ran iperf tests with iperf2 and iperf3 on the physical servers, investigated improving the throughput performance and finally did the same tests on the docker containers. We also did a brief research on docker in general and what type of networking we shall use in the further steps of this project. We investigated few important docker commands, created few containers, created our own docker-bridge network. For more information about docker and docker commands we refer to the official docker documentation on the website [17].

4 SDN Scenario in the server

4.1 Introduction

In this chapter we will carry out an analysis on the state of the art related to SDN technologies in PON networks, to later describe the implementation of our specific SDN scenario. Then, the steps to follow to implement our SDN solution will be described sequentially, from the implementation of a router and DHCP server in a container, to the configuration of virtual switch (OVS). We will finish with the connection and start-up of the OVS with the OpenDayLight controller, and ONOS necessary for the configuration and management of OpenFlow flows. In this way the management and the configuration of services and data traffic will be implemented in the GPON network using the OpenFlow protocol.

4.2 Global Description of the SDN scenario deployed in the server

We propose an SDN-GPON solution that permits to configure a GPON by means of OpenFlow using an external SDN controller and several OpenFlow switches. The architectural solution is shown in Fig. 30. To turn legacy OLTs and ONTs into OpenFlow controllable devices, an SDN hardware abstraction layer must be implemented. Therefore, we propose to use OpenFlow Virtual Switches (OVS) connected to the OLT and to ONTs, and an OpenDayLight (ODL) [20] controller (Fig. 30). However, other SDN controllers, such as ONOS [21]could also be used. In this network scenario, the SDN controller will be able to dynamically modify services according to real network traffic or user requirements, allowing a flexible control of the GPON capabilities. Therefore, the SDN controller belongs to ISPs/Network Operators that provide Internet connections and services to their customers. Moreover, as GPONs operate in two channels with different wavelengths, the downstream channel (from the OLT to ONTs) and the upstream channel (from ONTs to the OLT), the SDN controller has to deal with the traffic in both channels, so that the contracted services of the network subscribers comply with the corresponding QoS requirements (e.g., guaranteed bandwidth).

Ideally, the OpenFlow switches (OVS) should be integrated inside the OLT and the ONTs (SDN based OLT/ ONTs). However, as SDN-based OLTs and ONTs are not yet available, the OVSs can be embedded in an external hardware like Raspberry Pi, Banana Pi, mini-computer or a computer. One OVS should be implemented in a computer logically co-located with the OLT (Central OVS, COVS) to emulate the SDN layer of the OLT, and thus to control the downstream traffic of the entire GPON (Fig. 30). On the other hand, at the users' side, an OVS (Remote OVS, ROVS) lies beside each ONT to control the upstream traffic requirements of each subscriber. These devices will handle a lower computation load than the COVS, so cheaper devices than computers can be used for ROVS, like Raspberry Pi, as shown in ONT1 and ONT2 in Fig. 30.

To set an SDN configuration, the ODL controller sends OpenFlow messages to every ROVS through the conventional GPON channels and to the COVS through a direct connection. First, the SDN controller sends flow tables to the virtual switches (COVS, ROVS) to configure the services (Internet, HDTV, VoIP) and their QoS requirements. In fact, the OpenFlow tables are programmed by the SDN controller (that belongs to ISPs/Network Operators) and their entries are modified in real-time each time residential users demand new services or any modification in their QoS requirements. Then, two flows are created for each service as the GPON operates in two channels: one for managing the downstream QoS requirements and another for the upstream QoS requirements. The former flow is created in the COVS and the later in the ROVS. The match instructions and fields of each flow differ depending on the channel and on the developed functionalities, as it will be explained in the next sections. For example, if the maximum bandwidth is going to be controlled by means of SDN, the bandwidth rate assigned to each service (at both channels) is measured with OpenFlow meters. A meter measures the rate of packets assigned to it and enables controlling the rate of those packets. Meters are attached directly to flow entries. In our proposal each meter entry consists of a meter identifier and a meter band. The meter band specifies the maximum rate associated with that flow (band rate) and the way to process the packets of the flow (band type). Besides, some band types have optional arguments (drop, dscp remark) called type specific arguments and we use the drop option. Then, the data rate of each meter band is continuously measured, and if the rate is larger than the value defined in the band rate drop, packets are discarded, so this choice can be used to define a rate limiter band. Therefore, to control the maximum bandwidth of one specific service by means of meters, it is necessary to attach one meter to each flow, one for the maximum downstream bandwidth and another for the maximum upstream bandwidth of the service. Then, the band rate defines the maximum bandwidth associated with the service at both channels (upstream, downstream). Finally, it should be noted that since residential users are not expected to know the concept of flows and meters, this low-level process is totally abstracted into higher-level commands that can be understand by users, such as guaranteed bandwidth levels.

On the client side, GPONs can employ L3-model ONTs, which integrate router functionalities, and L2-model ONTs, without routing functionalities. L2 ONTs act as switches and they are transparent for network devices, so that it is necessary to add an L3 element to route the packets towards the residential network and to provide unique IP address to the different devices connected to the ONT. For the implementation of the proposed SDN-GPON solution, L2 ONTs are preferred due to flexibility, as they are not limited by the L3 vendor ONTs routing options. Thus, we deploy OVS and configure flows differentiated by IP addresses, acting similarly to a router although they are L2. Then, every ROVS needs a local DHCP server to provide IP addresses to the devices connected to its associated L2 ONT. First, the global DHCP server of the service providers or network operators assigns a range of IP addresses to each local DHCP server in the ROVSs, and then each ROVS assigns IP addresses to each connected device. It is worth noting that the IP addresses are no longer local to the residential network. Thus, every device will have a unique IP address in the GPON.

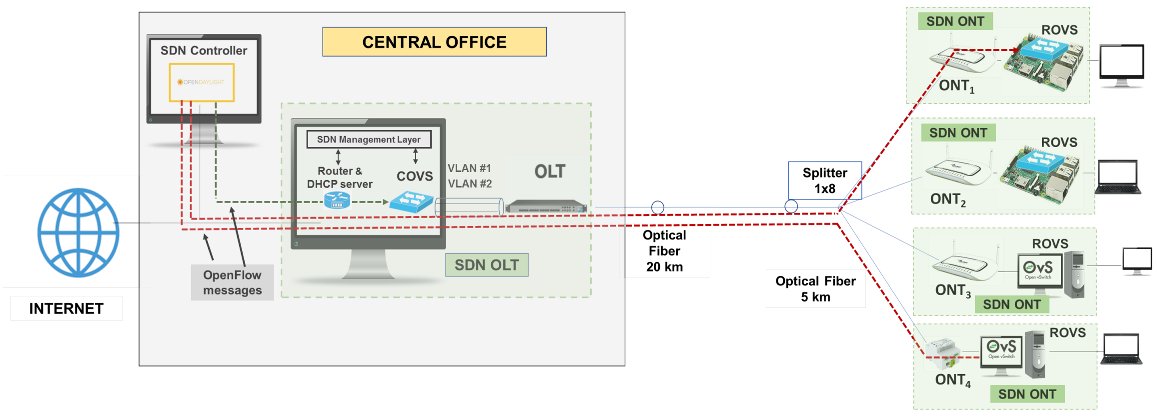


Figure 30: Classic GPON Scenario

The purpose of this project is to translate the applications i.e. OVS, Router, DHCP Server and SDN Controller, previously installed on the central computer to the Server (Chron2) in order to control the global GPON using both, the 1 Gpbs and the 10 Gbps ports on the OLT and ONT.

This implementation will be done by using the Docker technology to containerize the above-mentioned applications, by building Docker images to create a lightweight and fast bootable version of the applications on the Docker (see Chapter 2 Section 3 for information regarding Docker images, containerizing and virtualizing).

Therefore, two containers will be created on the Docker as shown in Figure 31. The first container will serve the function of DHCP server, the OVS and the Router. The second container will be used as the SDN controller. If the initial tests with two containers succeeds and the project is on schedule, the next step would be to separate each application on different containers and run the tests on them.

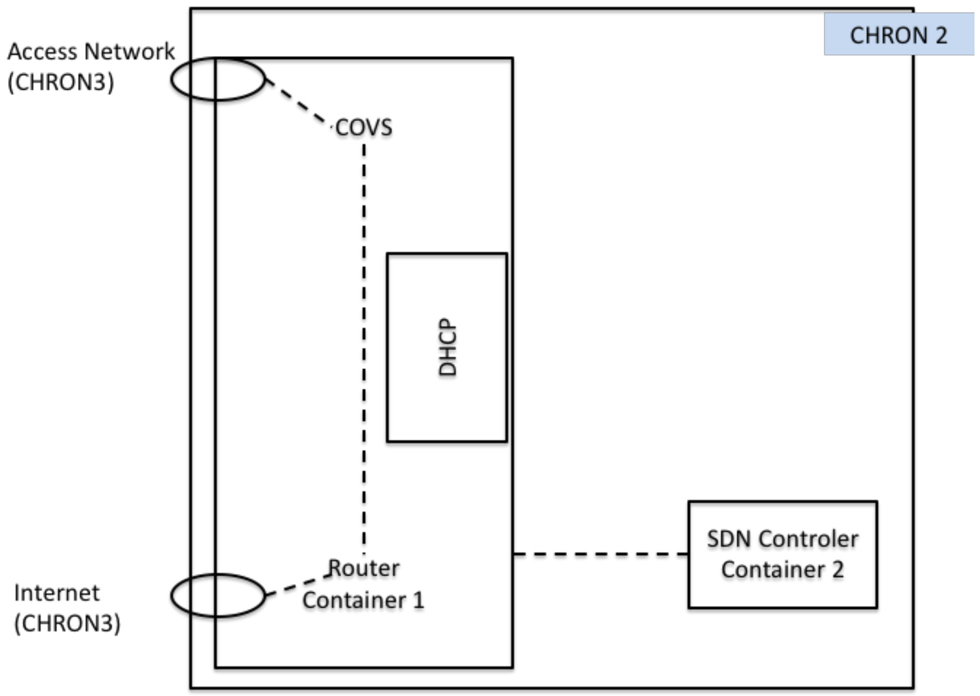


Figure 31: Network Topology with two docker containers

4.3 Router and DHCP server deployment

4.3.1 *Programming Linux Kernel Routing and a DHCP server*

The first step to achieve the desired scenario is to program a router on a computer with Linux. To do this, we will use the Linux kernel utility of forwarding IP packets ([22] between several networks (that is, that the Chron2 acts as a router). We need to perform this task to control the network that reaches the GPON model, and not depend on the School router for the different necessary configurations. Subsequently, a DHCP (Dynamic Host Configuration Protocol) server will be installed to give the network configuration parameters to the machines that connect to the central computer

4.3.2 *Deployment of the router and DHCP server*

Before everything can be started a docker container must be created to configure the router and a DHCP server.

The very first step into this process is to download the official Debian image from the Docker Hub. This can be done by typing the command docker pull Debian [23]. This pulls the latest version of the official Debian image from the Docker hub. The images are updated frequently [24].

The download image is renamed for convenience. This can be done with the command: *docker tag <old-image-name:tag> <new-image-name>*

In this case this would be *docker tag debain:latest router:0.1*. The image has been renamed to *router* and it has *0.1* as its version. The images name can only contain lower case characters. After the image has been renamed, the old image can be removed with *docker rmi <old image name:tag>* command. It can also be left alone as the image can be used to build controller and other different containers in the future. Refer to Chapter 2 Section 4 about Docker to learn more about Docker images and containers.

To run the image and start a container with right privileges and correct networking features the following command is needed: *docker run -t -i --cap-add SYS_ADMIN --network host --name=<container-name>   --privileged router:0.1*

Now on, all the instructions described will be executed inside the container. The official Debian image that was pulled from the Docker hub does not contain many packages that will be needed during the installation of the DHCP server and the router configuration. After the container has been started, the necessary packages can be installed. For starters the following packets are very compulsory [18]:

- net-tools
- text editor (nano)
- isc-dhcp-server
- iproute2
- iputils-ping
- net-tools
- ifupdown
- vlan
- iperf
- iperf3

These are the needed packages that permit us to do the configurations and the tests in the machine related to the project. There might be need of more but that could be installed anytime later with the command:

apt install <package-name> -y

Since the central computer has Linux installed, the router will be based on IP packet forwarding that is already implemented in Linux systems. This can be done in a very simple way by writing a '1' in the file *ip_forward* which can be found in the path */proc/sys/net/ipv4/ip_forward*. This makes the central computer behave like a router, that is, it will be able to forward packets from a subnet (the subnet that connects Chron3 and Chron2) to the subnet that is going to be connected the OLT (as well as the subnets implemented behind each ONT), as it can be observed in Figure 32.

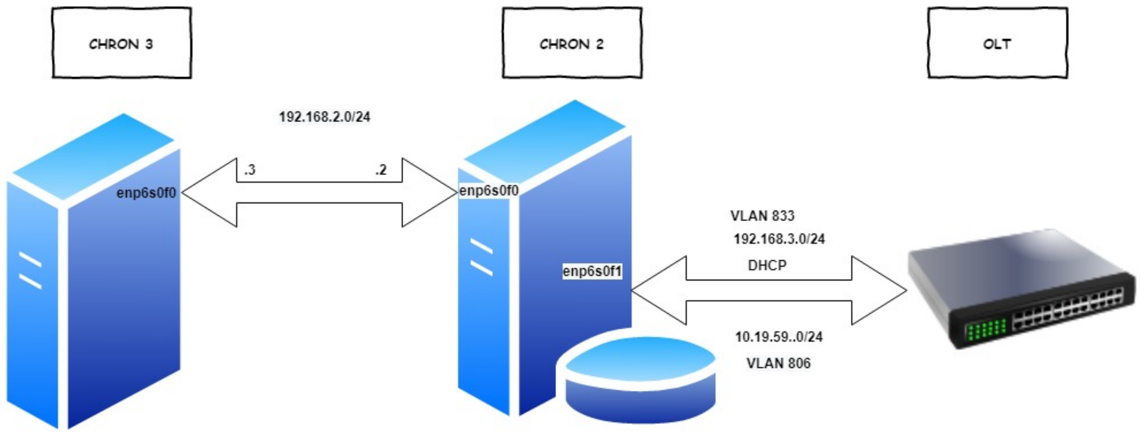


Figure 32 : Topology of the connection of the Router to the OLT

For our router to act as intended, we must configure the tables and interfaces. It is recommended to act wisely and carefully when configuring the IP tables using the host mode, as the container is being run on the host mode, any changes mades inside the container regarding the networking also means that these changes take place on the physical server's configuration. We are going to use the iptables utility [25], although it is true that it is usually used more as a utility to establish firewall rules, it also helps us to establish static rules on how we want our kernel to forward packets.

These rules are specified through the command line, using the necessary parameters that we will see later in this section, and they are usually limited to specifying from which subnet to which subnet we want to take the packages, and if we want to treat them one way or another. Specifically, we are going to apply two rules:

iptables -t nat -A POSTROUTING -s 192.168.3.0/24 -o enp6s0f0 -j MASQUERADE

iptables -t nat -A POSTROUTING -s 10.19.59.0/24 -o enp6s0f0 -j MASQUERADE

iptables -A FORWARD -i enp6s0f0 -j ACCEPT

iptables -A FORWARD -o enp6s0f0 -j ACCEPT

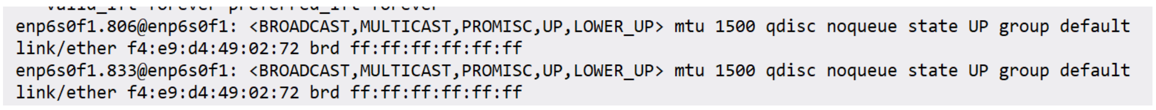
In these rules, the different options can be described as:

- <u>The –t option</u> specifies which table this command should refer to. In this case, it refers to the NAT (Network Address Translation) table. This table is consulted when an incoming package creates a new connection, and has two possibilities, PREROUTING (to act on packages as soon as they reach our kernel) and OUTPUT (to act on the packages that are generated locally before it is routed).
- <u>The –A option</u> specifies which rule we want to add in the table previously specified.
- <u>The –S rule</u> specifies the destination of the packets to which we want to apply the rule. In this case, we will add the subnet 192.168.3.0/24, and 10.19.59.0/24 since that will be the subnet in which our machine will be responsible for routing.
- <u>The –o option</u> is the interface by which we want to take out the packages that meet the above rules. The interface that is connected to the router, that is the interface enp6s0f0 which has IP address of 192.168.2.2/24.
- <u>The –j rule</u> indicates what to do with packages that match everything specified above.

Once the routing configurations are completed as shown above, the next step is to install and configure the DHCP server. The VLANs needs to be configured, so the VLANs that appear in Figure 33 are added. The first VLAN will have the tag 833 and contain the subnet 192.168.3.0/24 and the second one will have tag 806 with 10.19.59.0/24 as its subnet. VLANs are added with vconfig commands [26], that is:

vconfig add enp6s0f1 833

vconfig add enp6s0f1 806



```
enp6s0f1.806@enp6s0f1: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
link/ether f4:e9:d4:49:02:72 brd ff:ff:ff:ff:ff:ff
enp6s0f1.833@enp6s0f1: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
link/ether f4:e9:d4:49:02:72 brd ff:ff:ff:ff:ff:ff
```

Figure 33 : Creation of VLANs

Finally, the address of each virtual interface (192.168.0.1/24 and 10.19.59.1/24) are added with the commands:

ifconfig enp6s0f1.833 192.168.3.1 netmask 255.255.255.0

ifconfig enp6s0f1.806 10.19.59.1 netmask 255.255.255.0

After the above steps are completed, we must provide IP address to all connected devices on those interfaces. To provide IP address to the connected devices, the DHCP server will be used. The DHCP server can be installed with the command *apt-get install -y isc-dhcp-server* [27].

After the DHCP server has been installed few files needs to be configured to get the dhcp server working.

The first file can be found in path */etc/dhcp/dhcpd.conf* and few rules needs to be added to it. In this file we add different subnets to configure, being able to create more. The features to be configured are the following:

- range (range)
- domain name servers (DNS)
- domain name (domain-name)
- router subnet (router)
- broadcast address (broadcast)
- default lease time (default-lease-time)
- and maximum grant time (max-lease-time)

In fact, the exact configuration of this file is the next:

```
subnet 192.168.3.0 netmask 255.255.255.0 {
range 192.168.3.100 192.168.3.250;
option domain-name-servers 157.88.129.90;
option domain-name "RouterTFGLab7-833";
option routers 192.168.3.1;
option broadcast-address 192.168.3.255;
default-lease-time 600;
max-lease-time 7200;
}
subnet 10.19.59.0 netmask 255.255.255.0 {
range 10.19.59.100 10.19.59.200;
option domain-name-servers 157.88.129.90;
option domain-name "RouterTFGLab7-806";
option routers 10.19.59.1;
option broadcast-address 10.19.59.255;
default-lease-time 600;
max-lease-time 7200;
}
```

The second file is found in the path */etc/default/isc-dhcp-server* and the he following rules are added to the file:

```
INTERFACESv4="enp6s0f1.833 enp6s0f1.806 "
```

With this command, the server is being asked to accept DHCP requests on those virtual interfaces and the dhcp server is ready. After all the above changes has been made the DHCP server on the chron2can be started with command *service isc-dhcp-server start*.

Due to unforeseen circumstances (covid-19) it is impossible to go to the lab and configure OLT so the DHCP server and routing cannot be tested that way. Therefore, we decided to alter our topology in order to be able to test the working of the configurations without any trouble. The altered version of the topology can be found on Figure 34.

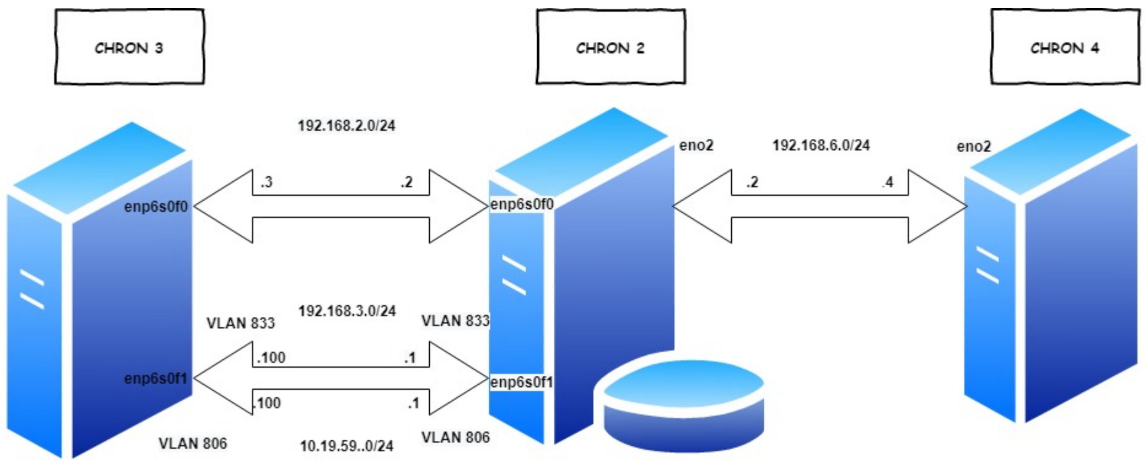


Figure 34 : New topology to test the routing and DCHP server.

In order to test this configuration, the same VLANs needs to be created on Chron3 and IPTABLE rules regarding the new *eno2* connection to Chron4 needs to be added to the iptables. This can be done the same way as done earlier with the next commands.

On the terminal of Chron 3 we type:

- *vconfig add enp6s0f1* 833 and *vconfig add enp6s0f1 806*
- route add -net 192.168.6.0 netmask 255.255.255.0 gw 192.168.2.3

The first command creates the virtual interfaces that will be enabled as DHCP clients later and the second command adds a static route to the routing table of CHRON3 asking the CHRON3 to use 192.168.2.3 (enp6s0f0) as its gateway to reach the network 192.168.6.0 that connects the CHRON 2 and CHRON 4 to each other. The above information is easier to visualize when observed together with figure 5.

On the terminal of CHRON2 we type:

- *iptables -t nat -A POSTROUTING -s192.168.6.0/24 -o enp6s0f0 -j MASQUERADE* on CHRON 2

The 192.168.6.0/24 will be the subnet that the new interface *eno2* will contain. The iptables rule is like the ones added earlier for the virtual interfaces.

4.3.2.1 DHCP Server and DHCP Client tests

In figure 35 we once again repeat the topology that has been adjusted to run the tests (as previously mention, because we cannot access the labs anymore). To test the DHCP server, we type on the Chron3 (so it emulates a real device such as the OLT in the GPON) and send the dhcp server (located in Chron 2) for DHCP request on the interface enp6s0f1.833 with the command: *dhclient enp6s0f1.833*

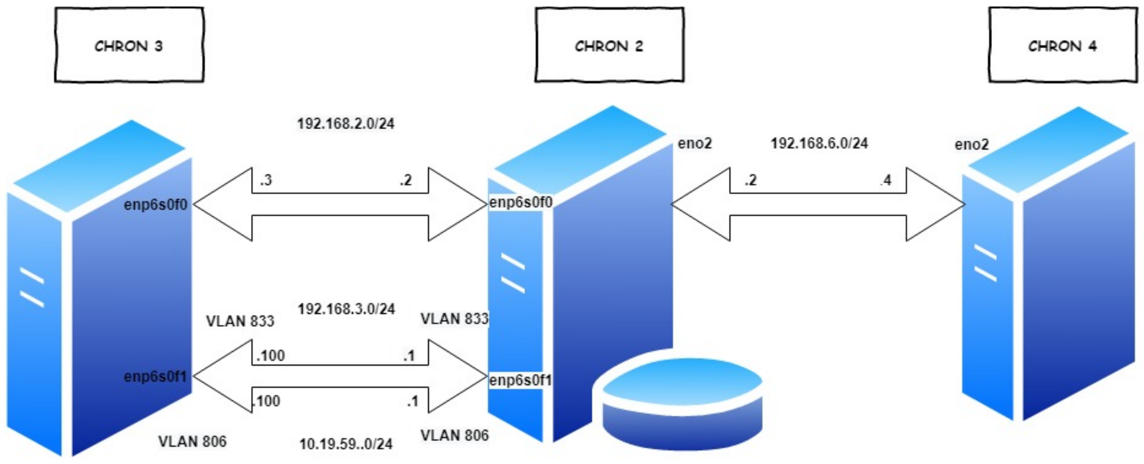
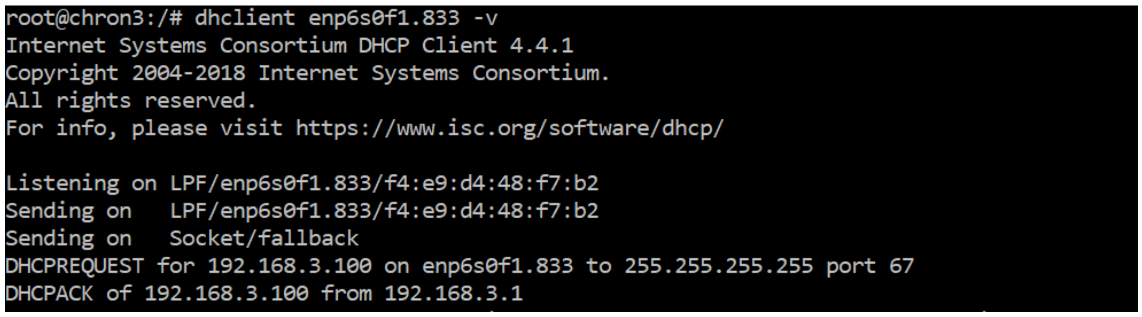


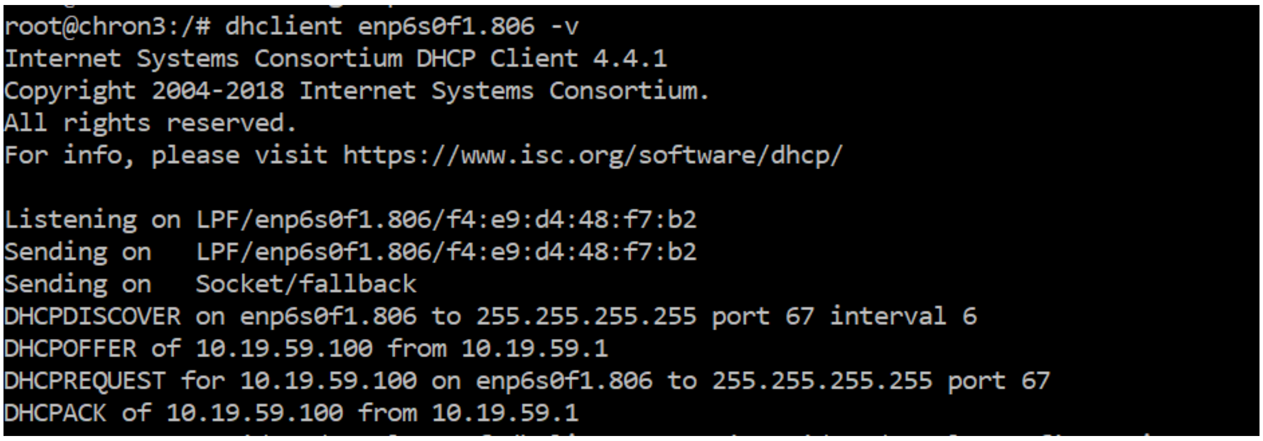
Figure 35 : Topology created to test the DHCP server and the router



```
root@chron3:/# dhclient enp6s0f1.833 -v
Internet Systems Consortium DHCP Client 4.4.1
Copyright 2004-2018 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/

Listening on LPF/enp6s0f1.833/f4:e9:d4:48:f7:b2
Sending on   LPF/enp6s0f1.833/f4:e9:d4:48:f7:b2
Sending on   Socket/fallback
DHCPREQUEST for 192.168.3.100 on enp6s0f1.833 to 255.255.255.255 port 67
DHCPACK of 192.168.3.100 from 192.168.3.1
```

Figure 36: DHCP ACK 833

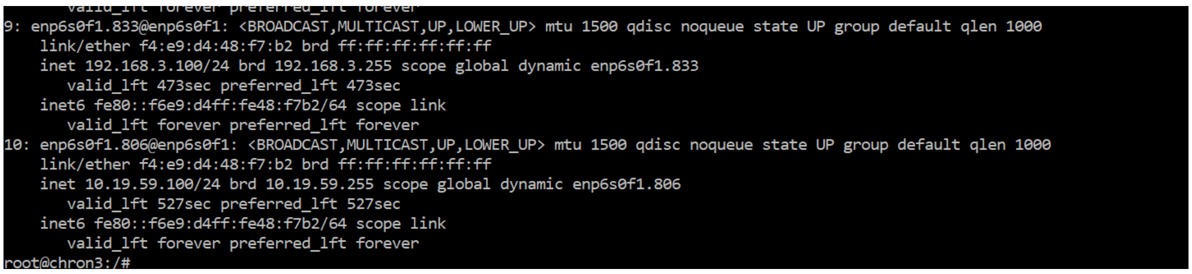
Figure 36 shows that the 833 interface's request for IP address has been acknowledged and has been given 192.168.3.100. This is within the range, that was configured earlier on the dhcp server. Moreover, Figure 37 shows that the DHCP request from the interface 806 (with the command *dhclient enp6s0f1.806*) has also been accepted as dhcp client and given the IP address 10.19.59.100.



```
root@chron3:/# dhclient enp6s0f1.806 -v
Internet Systems Consortium DHCP Client 4.4.1
Copyright 2004-2018 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/

Listening on LPF/enp6s0f1.806/f4:e9:d4:48:f7:b2
Sending on   LPF/enp6s0f1.806/f4:e9:d4:48:f7:b2
Sending on   Socket/fallback
DHCPDISCOVER on enp6s0f1.806 to 255.255.255.255 port 67 interval 6
DHCPOFFER of 10.19.59.100 from 10.19.59.1
DHCPREQUEST for 10.19.59.100 on enp6s0f1.806 to 255.255.255.255 port 67
DHCPACK of 10.19.59.100 from 10.19.59.1
```

Figure 37: DHCP ACK 806

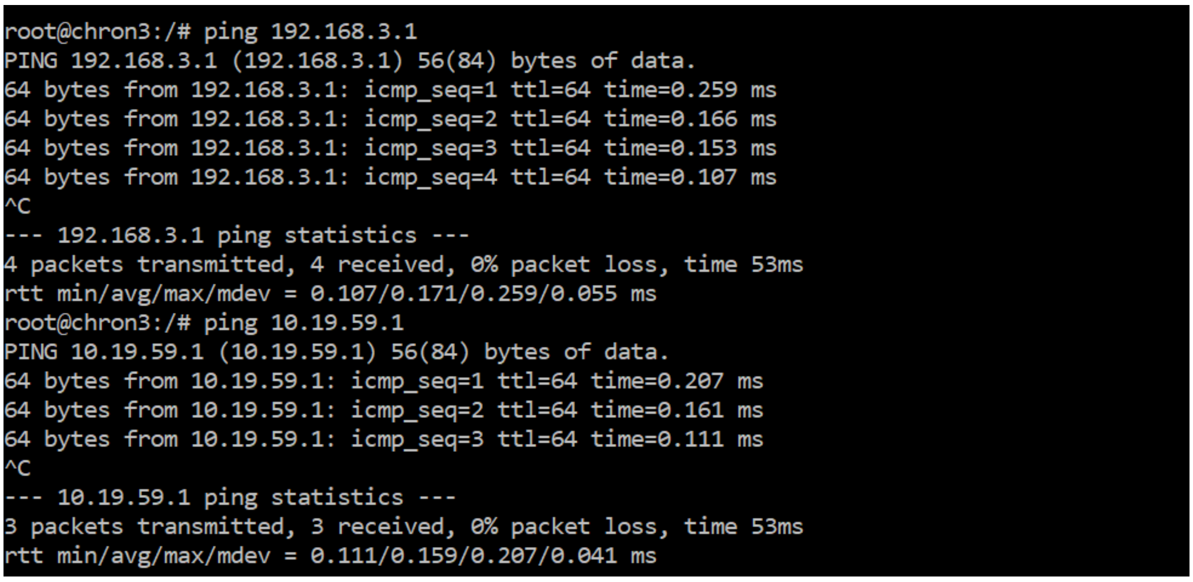
To make sure if the IPs are configured, it is done with the command *ip -a* as it is shown in Figure 38.



```
       valid_lft forever preferred_lft forever
9: enp6s0f1.833@enp6s0f1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether f4:e9:d4:48:f7:b2 brd ff:ff:ff:ff:ff:ff
    inet 192.168.3.100/24 brd 192.168.3.255 scope global dynamic enp6s0f1.833
       valid_lft 473sec preferred_lft 473sec
    inet6 fe80::f6e9:d4ff:fe48:f7b2/64 scope link
       valid_lft forever preferred_lft forever
10: enp6s0f1.806@enp6s0f1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether f4:e9:d4:48:f7:b2 brd ff:ff:ff:ff:ff:ff
    inet 10.19.59.100/24 brd 10.19.59.255 scope global dynamic enp6s0f1.806
       valid_lft 527sec preferred_lft 527sec
    inet6 fe80::f6e9:d4ff:fe48:f7b2/64 scope link
       valid_lft forever preferred_lft forever
root@chron3:/#
```

Figure 38: IP Check on the 806 and 833 interfaces

The next step is to check the connection between the Chron 3 and Chron 2 via the virtual interfaces. The interfaces on the Chron 2 are pinged from Chron 3 as shown in Figure 39.



```
root@chron3:/# ping 192.168.3.1
PING 192.168.3.1 (192.168.3.1) 56(84) bytes of data.
64 bytes from 192.168.3.1: icmp_seq=1 ttl=64 time=0.259 ms
64 bytes from 192.168.3.1: icmp_seq=2 ttl=64 time=0.166 ms
64 bytes from 192.168.3.1: icmp_seq=3 ttl=64 time=0.153 ms
64 bytes from 192.168.3.1: icmp_seq=4 ttl=64 time=0.107 ms
^C
--- 192.168.3.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 53ms
rtt min/avg/max/mdev = 0.107/0.171/0.259/0.055 ms
root@chron3:/# ping 10.19.59.1
PING 10.19.59.1 (10.19.59.1) 56(84) bytes of data.
64 bytes from 10.19.59.1: icmp_seq=1 ttl=64 time=0.207 ms
64 bytes from 10.19.59.1: icmp_seq=2 ttl=64 time=0.161 ms
64 bytes from 10.19.59.1: icmp_seq=3 ttl=64 time=0.111 ms
^C
--- 10.19.59.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 53ms
rtt min/avg/max/mdev = 0.111/0.159/0.207/0.041 ms
```

Figure 39 : Connection between virtual interfaces on chron3 and chron2

To make sure and check if the right interfaces are being pinged, the *command tcpdump -i enp6s0f1.833* can be done on the Chron2.

4.3.2.2 Routing Test

The DHCP server is working as intended, but we must check the performance of the router. For this, the Gigabit interface *eno2* will be used on the Chron2 and Chron 4. Then, both interfaces are given an IP address with the command:

- On Chron2: *Ifconfig eno2 192.168.6.2 netmask 255.255.255.0*
- On Chron 4: *ifconfig eno2 192.168.6.4 netmask 255.255.255.0*

After the ifconfigs, the connection is tested with pings. Then, the three Chron servers are connected to each other as shown in Figure 40 in order to test the performance.

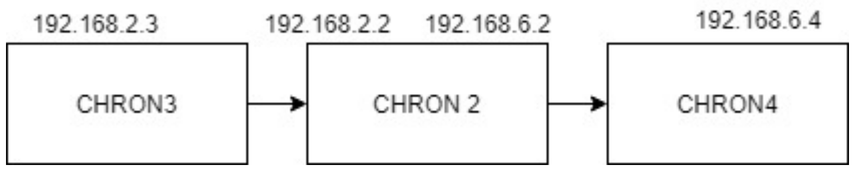
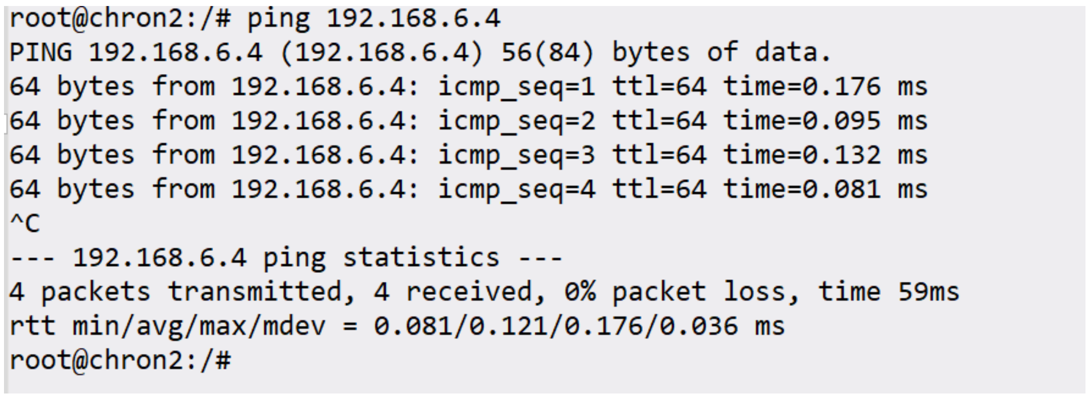


Figure 40: Topology for testing the Router performance

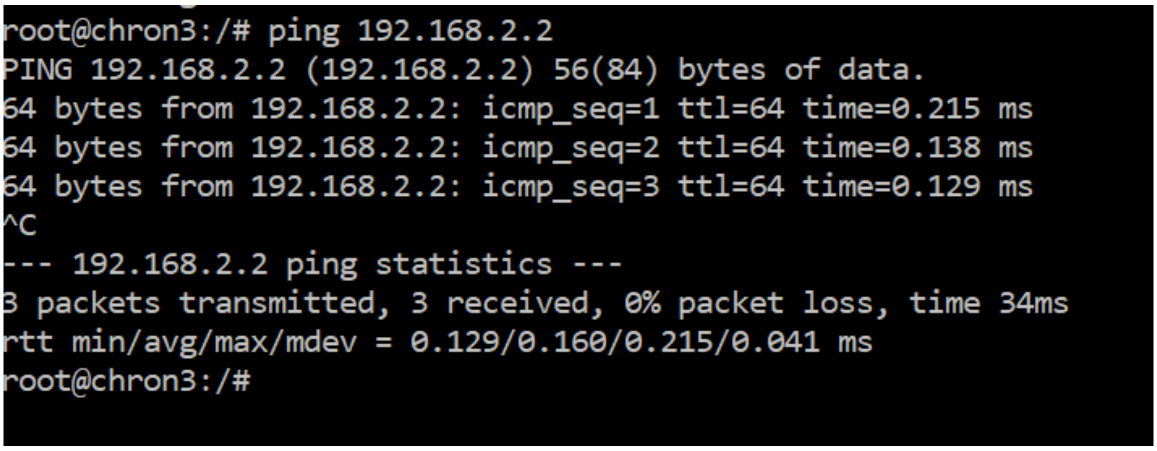
First, the Chron 4 is pinged from Chron 2 (router) as it is shown in Figure 41 and we can observe that the connection is working.



```
root@chron2:/# ping 192.168.6.4
PING 192.168.6.4 (192.168.6.4) 56(84) bytes of data.
64 bytes from 192.168.6.4: icmp_seq=1 ttl=64 time=0.176 ms
64 bytes from 192.168.6.4: icmp_seq=2 ttl=64 time=0.095 ms
64 bytes from 192.168.6.4: icmp_seq=3 ttl=64 time=0.132 ms
64 bytes from 192.168.6.4: icmp_seq=4 ttl=64 time=0.081 ms
^C
--- 192.168.6.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 59ms
rtt min/avg/max/mdev = 0.081/0.121/0.176/0.036 ms
root@chron2:/#
```

Figure 41: Chron 4 pinged from Chron 2

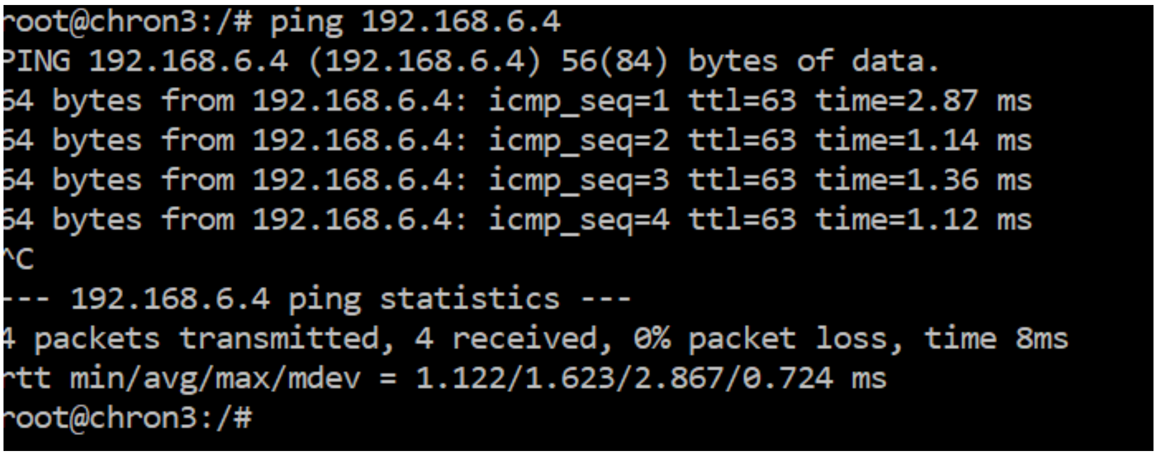
To test the complete how the router performs, the Chron 2 will be pinged from Chron 3 first as shown in Figure 42 and the connection is working as intended.



```
root@chron3:/# ping 192.168.2.2
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
64 bytes from 192.168.2.2: icmp_seq=1 ttl=64 time=0.215 ms
64 bytes from 192.168.2.2: icmp_seq=2 ttl=64 time=0.138 ms
64 bytes from 192.168.2.2: icmp_seq=3 ttl=64 time=0.129 ms
^C
--- 192.168.2.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 34ms
rtt min/avg/max/mdev = 0.129/0.160/0.215/0.041 ms
root@chron3:/#
```

Figure 42 :Chron 2 pinged from Chron 3

Finally, the Chron 4 server will be pinged from the Chron 3 as it can be seen on Figure 43 and it can be noticed that the connection is working as intended. Therefore, the DHCP server and the Router are working perfectly, and we can move on to the next steps of the project.



```
root@chron3:/# ping 192.168.6.4
PING 192.168.6.4 (192.168.6.4) 56(84) bytes of data.
64 bytes from 192.168.6.4: icmp_seq=1 ttl=63 time=2.87 ms
64 bytes from 192.168.6.4: icmp_seq=2 ttl=63 time=1.14 ms
64 bytes from 192.168.6.4: icmp_seq=3 ttl=63 time=1.36 ms
64 bytes from 192.168.6.4: icmp_seq=4 ttl=63 time=1.12 ms
^C
--- 192.168.6.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 8ms
rtt min/avg/max/mdev = 1.122/1.623/2.867/0.724 ms
root@chron3:/#
```

Figure 43 :Chron 4 pinged from Chron 3.

4.4 Installation and deployment of the Open vSwitch in the server

The Open vSwitch will be installed on the same container as the DHCP server. After the container has been started, the necessary packages the switch needs be able to install and implement the Open vSwitch inside a container will be installed. The list and steps can be found on the official documentation of the Open vSwitch [28]. All packages can be installed with one command:

apt-get install -y wget net-tools nano iproute2 iputils-ping ifupdown vlan iperf iperf3 git curl python-simplejson python-qt4 python-twisted-conch python3.6 automake autoconf gcc uml-utilities libtool build-essential pkg-config libssl-dev iproute2 tcpdump

It is possible that some packets have been installed in the earlier steps, but this command will make sure that the latest version are updated. This command downloads and installs all the dependencies and necessary tools that are necessary to install implement and test the Open vSwitch. After all the preparations are done, the latest version of open virtual switch is cloned from the official GIT repository https://github.com/openvswitch/ovs.git. The link clones the files from the master branch from the GitHub. During this project the supported version is ovs.2.13. To make the OVS works properly, we need to make sure the folder "build" exists in the path "*lib/modules/VERSION_KERNEL/build*". The VERSION_KERNEL in the path should match the output of the command *"uname -r"*. This was not the case for us, and this problem can be solved by installing the correct Linux-headers with the command: *apt-get install -y ′uname -r′* or just *apt-get install -y linux-headers-4.19.0-6-amd64* in this case. When the correct headers have been installed and the git repository of the OVS has been cloned, navigating to the folder with the command *cd ovs* will be the next step.

Running the *./boot.sh* will build the configuration script that is then executed afterwards with the command *./configure --with-linux=/lib/modules/$(uname -r)/build*. Then, the apply the *make* command and then the *make install* command. It is recommended to install the modules (*make modules install*) if the kernel version was built in the earlier steps, but the docker container lacking kernels of their own and the need for them to use the host kernel to be able to run any kernel-based software's caused unsolvable complexity in this situation. The user modules can be installed with the command *make install*.

The kernel modules was primarily thought to be needed to make the meters on the switch work properly but because the installation process of the kernel module on the container had few problems and the meters worked fine on the user space, we decided to move on with the user modules. To make sure if the meter can be configured to work better than performance achieved during this project, more research needs to be done regarding kernel module installation of OVS in a Docker Container.

Open vSwitch includes a shell script, and helpers, called *ovs-ctl* which automates much of the tasks for starting and stopping ovsdb-server, and ovs-vswitchd. After installation, the daemons can be started by using the *ovs-ctl* utility. This will take care to setup initial conditions and start the daemons in the correct order. Before starting *ovs-vswitchd* itself, it is necessary to start its configuration database, *ovsdb-server*. Each machine on which Open vSwitch is installed should run its own copy of *ovsdb-server*. Before *ovsdb-server* itself can be started, we must configure a database that it can use [28]:

```
$ mkdir -p /usr/local/etc/openvswitch
$ ovsdb-tool create /usr/local/etc/openvswitch/conf.db \
  vswitchd/vswitch.ovsschema
```

The *ovsdb-server* is then configured to use the database just created with the next commands:

$ mkdir -p /usr/local/var/run/openvswitch

*$ ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock *

*--remote=db:Open_vSwitch,Open_vSwitch,manager_options *

*--private-key=db:Open_vSwitch,SSL,private_key *

*--certificate=db:Open_vSwitch,SSL,certificate *

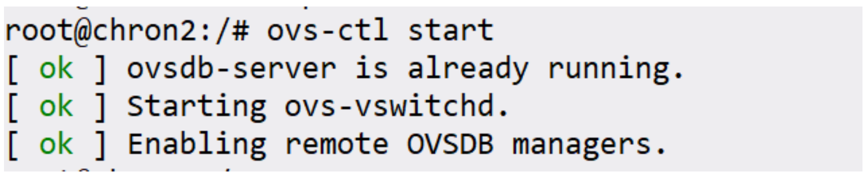
*--bootstrap-ca-cert=db:Open_vSwitch,SSL,ca_cert *

--pidfile --detach --log-file

Once the previous steps are done, the ovs can be finally started with the next commands and the result is shown in Figure 44.

$ export PATH=$PATH:/usr/local/share/openvswitch/scripts

$ ovs-ctl start



```
root@chron2:/# ovs-ctl start
[ ok ] ovsdb-server is already running.
[ ok ] Starting ovs-vswitchd.
[ ok ] Enabling remote OVSDB managers.
```

Figure 44: OVS start

For the next steps it is necessary to have a good understanding of how the open virtual switch works. Virtual switches act as bridges to which different interfaces can be connected and the packets passing through this bridge then can be managed and manipulated before getting passed through, as it can be observed in Figure 45.

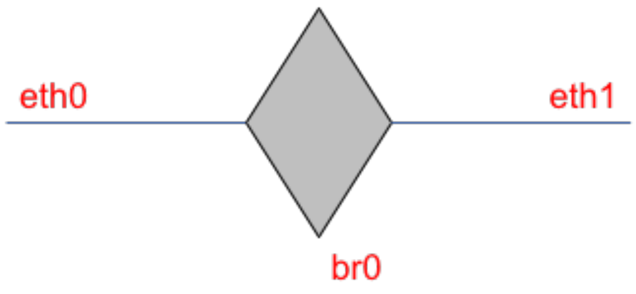


Figure 45 : Working of OVS bridge

Open vSwitch consists two sets of commands, one of which starts with *ovs-vsctl* and the other one with *ovs-ofctl*. The functionality of each of them can be summarized as:

- *ovs-vsctl*: This tool is used for configuration and viewing of switch operations. Bridge additions/deletions are just some of the options that are available with this command.
- *ovs-ofctl*: This tool is used for the administration and monitorization of the OpenFlow switches. Even if the switch has not been centralized with a controller, this command can be used to show the current state and table entries when they are configured in the OVS [29].

As mentioned earlier the two interfaces of the devices where the switch has been installed can be connected via the bridge which allows the two interfaces to belong of the same subnet and there is no need for a router between the two interfaces for the packets. This might seems like an advantage but in our case as the central server, it is a drawback because the connection of *enp6s0f0* interface (this gives us access to the internet from Chron3) and the virtual interfaces *enp6s0f1.833* (supposed to be connected to OLT) will bypass our router implemented in the server and the network will stop working. This conflict is also visible on Figure 46.

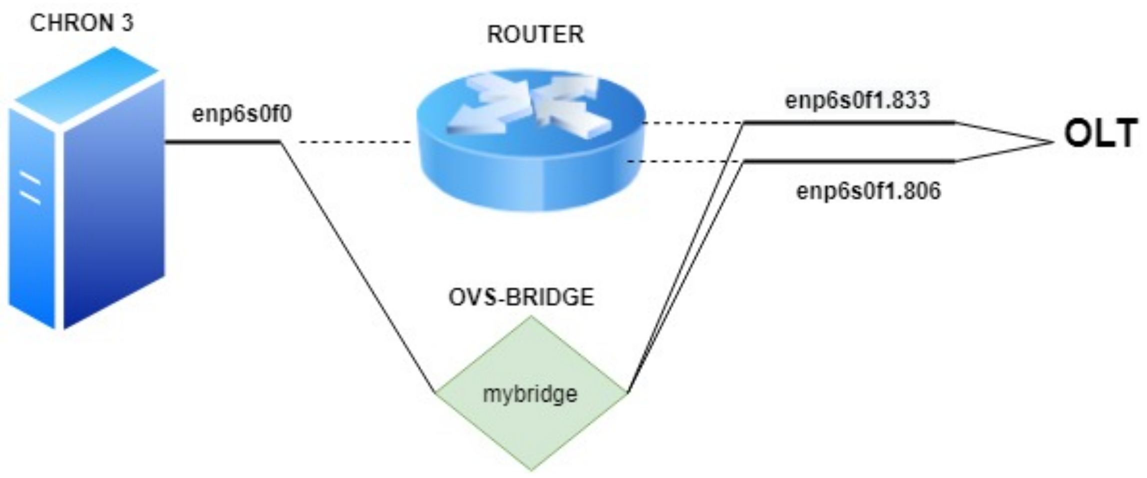


Figure 46 : Wrong implementation of the virtual switch

This conflict can be avoided by only adding the virtual interfaces to the bridge and letting the router route the incoming packets on the interface enp6s0f0 to the bridge instead of adding the interface *enp6s0f0* to the bridge. The new workaround to make sure the packets from Chron 3 destined for the OLT pass through the Router before it passes through the bridge can be seen on the Figure 47.

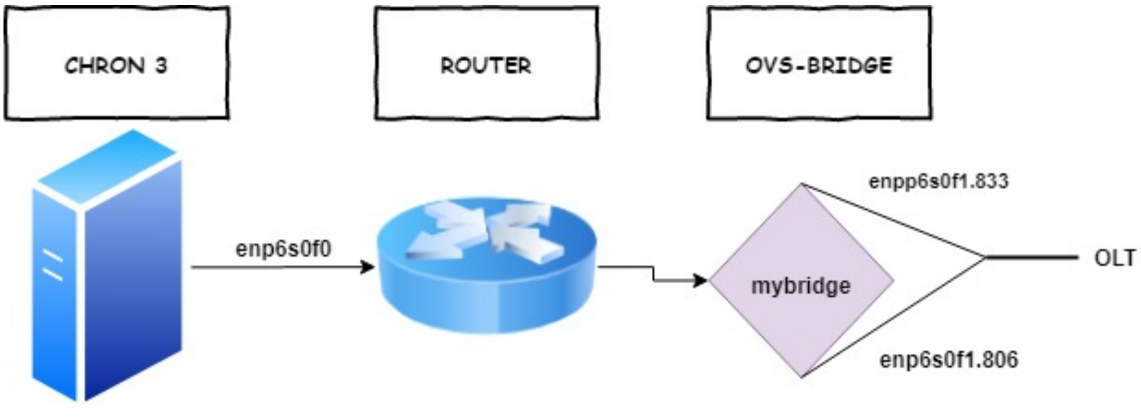


Figure 47 : Correct implementation of the virtual switch

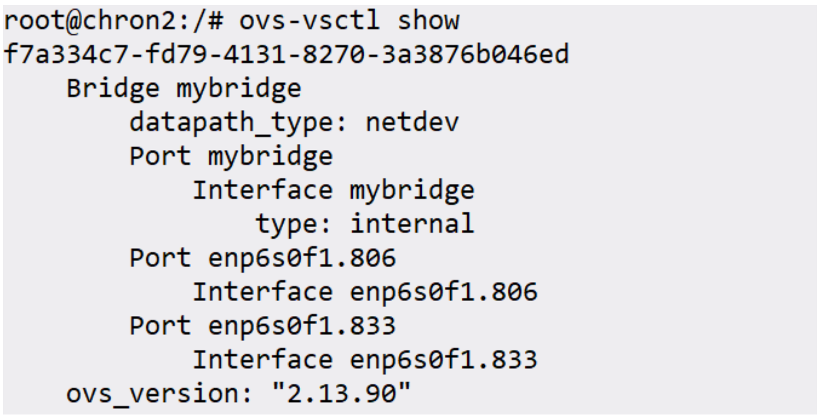
After the proper study about the virtual switches and bridges the building process can start. The next lines of commands make the configuration on the Figure 47 possible:

ovs-vsctl add-br mybridge

ovs-vsctl add-port mybridge enp6s0f1.833

ovs-vsctl add-port mybridge enp6s0f1.806

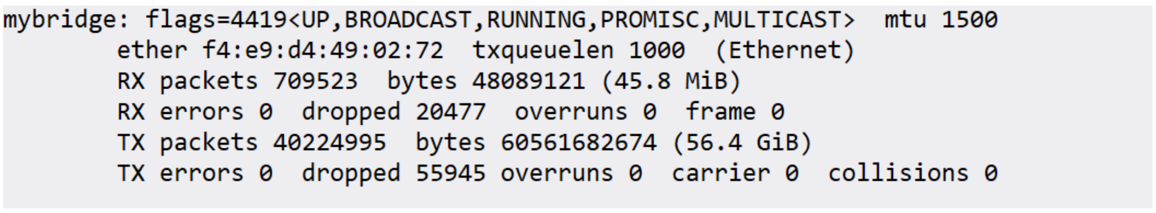
The above configuration creates an OVS bridge (mybridge) and adds the virtual interfaces 833 and 806 to the bridge. To make sure the ports have been added to the newly created ovs-bridge, the command *ovs-vsctl show* can be used to display the result [30]. The results are shown in Figure 48.



```
root@chron2:/# ovs-vsctl show
f7a334c7-fd79-4131-8270-3a3876b046ed
    Bridge mybridge
        datapath_type: netdev
        Port mybridge
            Interface mybridge
                type: internal
        Port enp6s0f1.806
            Interface enp6s0f1.806
        Port enp6s0f1.833
            Interface enp6s0f1.833
    ovs_version: "2.13.90"
```

Figure 48 : Display all ports on mybridge

At this point, it is also necessary to understand another important operation of the bridge that generates the virtual switches, since the bridge has been created but it still lacks any IP configurations, as it can be observed in Figure 49. The IP information needs to be added to it.



```
mybridge: flags=4419<UP,BROADCAST,RUNNING,PROMISC,MULTICAST>  mtu 1500
        ether f4:e9:d4:49:02:72  txqueuelen 1000  (Ethernet)
        RX packets 709523  bytes 48089121 (45.8 MiB)
        RX errors 0  dropped 20477  overruns 0  frame 0
        TX packets 40224995  bytes 60561682674 (56.4 GiB)
        TX errors 0  dropped 55945 overruns 0  carrier 0  collisions 0
```

Figure 49 : Newly created OVS bridge without any IP configurations.

At the network level the bridge overrides or replaces the addressing of the interfaces. This means that we will have to delete the IP configurations on those interfaces and add the same IP configurations to the bridge. The enp6s0f1.833 and enp6s0f1.806 interfaces are assigned different network addresses i.e. 192.168.3.0/24 and 10.19.59.0/24. To be able to add two different network interfaces to the same bridge interfaces, it can be done with the following commands. Firstly, the IP information on the virtual interfaces are deleted with commands:

ifconfig enp6s0f1.833 0

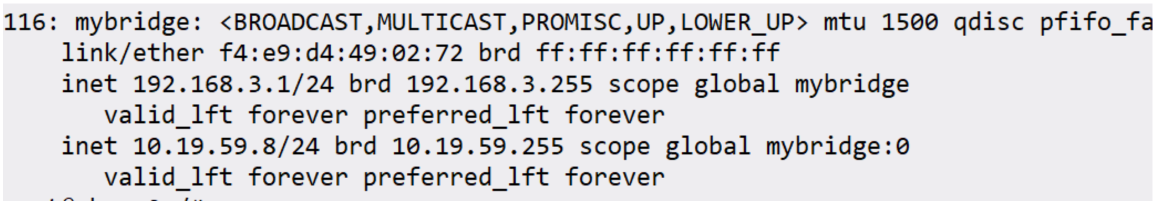
ifconfig enp6s0f1.806 0

Then the IP deleted from the virtual interfaces needs to be added to the bridge:

ifconfig mybridge 192.168.3.1 netmask 255.255.255.0

ifconfig mybridge:0 10.19.59.8 netmask 255.255.255.0

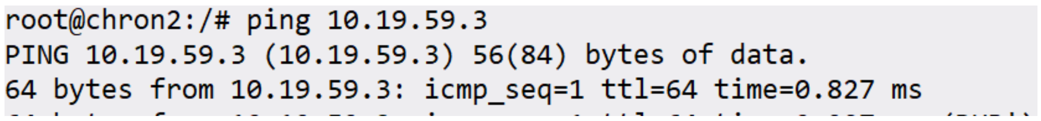
It can be seen in the Figure 50 that the bridge has two different addresses with same MAC address. The bridge has replaced the virtual interfaces and can now connect to the other side of the interfaces.



```
116: mybridge: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500 qdisc pfifo_fa
    link/ether f4:e9:d4:49:02:72 brd ff:ff:ff:ff:ff:ff
    inet 192.168.3.1/24 brd 192.168.3.255 scope global mybridge
       valid_lft forever preferred_lft forever
    inet 10.19.59.8/24 brd 10.19.59.255 scope global mybridge:0
       valid_lft forever preferred_lft forever
```

Figure 50 : Newly created OVS bridge with IP configurations

Then, in this case the interfaces are connected to the Chron 3 *enp6s0f1* port. To check the connection a simple ping will suffice, as it is shown in Figure 51.



```
root@chron2:/# ping 10.19.59.3
PING 10.19.59.3 (10.19.59.3) 56(84) bytes of data.
64 bytes from 10.19.59.3: icmp_seq=1 ttl=64 time=0.827 ms
```

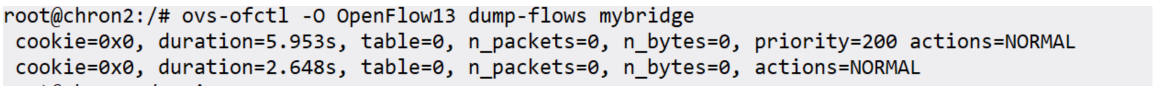
Figure 51 : connection check through the bridge

For the next part it is recommended to have good understanding about the OpenFlow switches and OpenFlow entries. The creation and configurations of OpenFlow entries is one of the actions of SDN controllers but for the purpose of testing and learning about the flows, they are often created on the switch in the begin. This process will also help to create default templates that can be later used to create new flow entries using the SDN controller. Then, the next code shows some flow two examples of flows configured on the bridge:

ovs-ofctl -O OpenFlow13 add-flow mybridge actions=normal

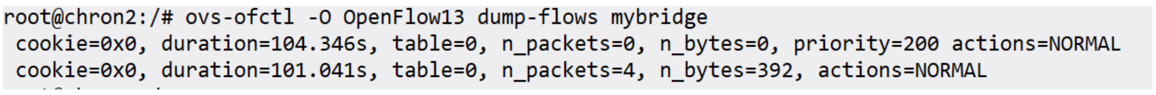
ovs-ofctl -O OpenFlow13 add-flow mybridge priority=200, actions=normal

In the above commands two flows are created with same actions, *normal*. It represents the traditional non-OpenFlow pipeline of the switch. With the action set to normal the switch forwards the packets from the OpenFlow pipeline to the normal pipeline. The second flow also has *priority=200* added to it, but the first command indicates no priority, which means that it is assigned the default priority of 32767. Then, the first flow will be used when the packets passes through this bridge during the packet transfer, since it has the highest priority. This can be tested by pinging the interfaces and look out for which flow has been hit. Before doing the connection test with the ping feature, to make sure the flows have been added, the state of the recently created flows can be printed by the command *ovs-ofctl -O OpenFlow13 dump-flows mybridge* which outputs the flow information and flow statistics. The flow statistics before and after the pings are shown in Figure 52 and Figure 53.



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x0, duration=5.953s, table=0, n_packets=0, n_bytes=0, priority=200 actions=NORMAL
 cookie=0x0, duration=2.648s, table=0, n_packets=0, n_bytes=0, actions=NORMAL
```

Figure 52: Flow lists with different priority before the pings



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x0, duration=104.346s, table=0, n_packets=0, n_bytes=0, priority=200 actions=NORMAL
 cookie=0x0, duration=101.041s, table=0, n_packets=4, n_bytes=392, actions=NORMAL
```

Figure 53 : Flow statistics after the pings

It can be seen in Figure 53 that the flow with the maximum priority has been used because it has the statistics of the packets that went through the bridge using that specific protocol. In contrast, the flow with the lowest priority (priority=200) has no packets registered in its statistics.

Furthermore, flows can also be created to manipulate packets received through specific mac addresses or network addresses (IPs). The next example shows a newly created flow where the packets entering the bridge through the source mac address *f4:e9:d4:49:02:70* and wanting to reach the destination mac address *f4:e9:d4:49:02:72* will be ported out through the port enp6s0f1.833:

ovs-ofctl -O OpenFlow13 add-flow mybridge dl_src:f4:e9:d4:49:02:70,dl_dst=f4:e9:d4:49:02:72,actions=output:enp6s0f1.833

The flow’s working can be tested by pinging the interface with the above give Mac address, as it was done earlier to test the priority.

On the other hand, meters can also be configured with the *ovs-ofctl* functionality on the switch with the next command:

ovs-ofctl -O OpenFlow13 add-meter mybridge meter=1, kbps, band=type=drop,rate=30000

In this way, a meter measures the rate of packets assigned to it and enables controlling the rate of those packets. Meters are attached directly to flow entries. Any flow entry can specify a meter in its instruction set the meter measures and controls the rate of the aggregate of all flow entries to which it is attached [31]. In the above example, a meter is created with a meter id of 1 and it only lets packets through up to 30 Mbps. Therefore, if the data rate is higher than 30 Mbps, packets are discarded, and the meter is then called a limiter because it limits the throughput on the bridge.

Even though the flow-entries and meter configurations are handled by the sdn-controller in the further development of the project, it is highly recommended to create at least one flow and a meter on the switch to use them as the templates in the future. Not all default templates available on the official website are compatible with the switch that has been installed, so testing the configurations by creating the flows using the traditional way is a good learning experience. In this way, in order that the switch can be controlled by one controller we have to insert the command:

ovs-vsctl set-controller mybridge tcp:10.1.4.0:6633

This command indicates that we want to establish connection with the OpenFlow controller located on the given ip address and the switch is ready to be centralized.

4.5 Installation of OpenDayLight (ODL) in docker

The OpenDayLight SDN controller will be installed on a separate container. The base image Debian is used to build this container as it was used earlier. The container is started with the command:

docker run -t -i --network <docker-bridge name > --name=<container-name> <imagename:tag>

To keep the container lightweight only the necessary packages will be installed when necessary. For starters, the packages *wget* and *tar* are installed with the command *apt-get install -y wget tar*. The *wget* lets the user download the opendaylight and *tar* lets the user open the tar folder (tape archiver).

The open Daylight releases are named after the elements of the periodic table. The version used in this project is named Sodium and it is the 11th release. Without prior knowledge, the flows configuration can be complex to learn, and it is recommended to use the supported latest releases as the bugs that still exist on the older version might cause problems in the future. The graphical user interface to create and configure flows on the ODL is not supported on any versions released after the 8th release (we are currently on the 11th and 12th), however although older versions support the GUI to configure the flows, it is recommended to adapt to the newer releases and use the POSTMAN application. The very first step is to browse the official Open Daylight website and read through the documentation regarding the installation and configuration [20]. Then, the *tar* file can be then installed with the command:

wget

https://nexus.opendaylight.org/content/repositories/opendaylight.release/org/opendaylight/integration/opendaylight/0.11.2/opendaylight-0.11.2.tar.gz

The above-mentioned link is provided by ODL official documentation [20] that links the user to the installation files of OpenDayLight directly.

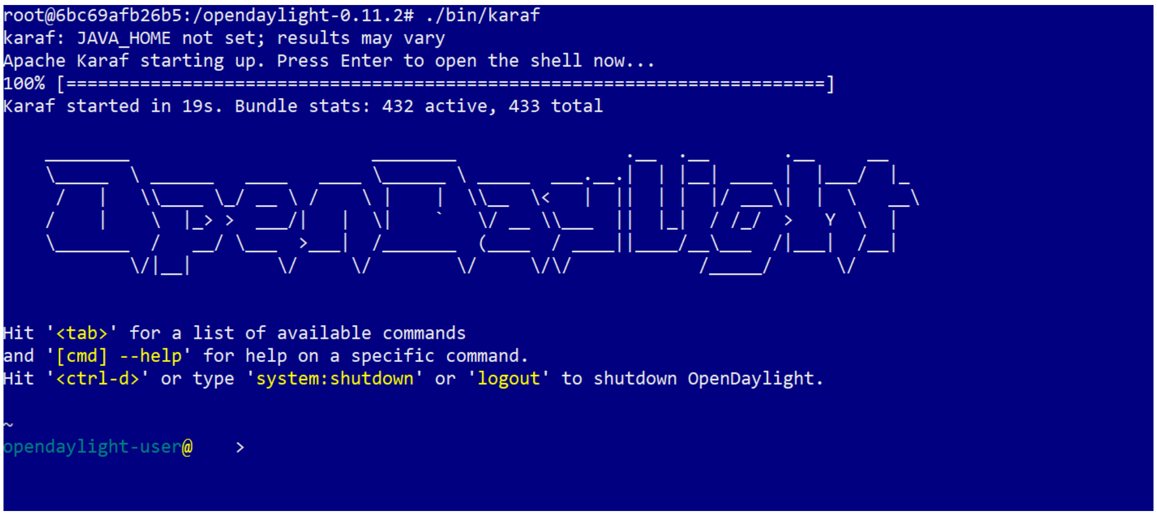
This wget combined with the link downloads the tar file of sodium version of OpenDayLight inside the working folder. It can be seen on the link as 0.11.2, which means that it is the 11th version and the 11th element of the periodic table i.e. Sodium. After the download, the tar file needs to be extracted with the command:

Tar -xf opendaylight-0.11.2.tar.gz

Then navigate to the directory with command *cd opendaylight-0.11.2* . This is the directory where all the installations files and other important files along with the startup script regarding the ODL can be found. The ODL start script will be run here.

On the other hand, the latest version of java needs to be installed on the machine for the latest version of ODL. The latest version can be installed with command *apt install -y default-jre*.

After the latest version of java has been installed, run *./bin/karaf* inside the current directory and the script starts and the Open Daylight will be installed, started and ready to be configured further, as it can be observed in Figure 54.



```
root@6bc69afb26b5:/opendaylight-0.11.2# ./bin/karaf
karaf: JAVA_HOME not set; results may vary
Apache Karaf starting up. Press Enter to open the shell now...
100% [========================================================================]
Karaf started in 19s. Bundle stats: 432 active, 433 total

    ________                       ________                .__  .__       .__     __
    \_____  \ ______   ____   ____ \______ \ _____  ___.__.|  | |__| ____ |  |___/  |_
     /   |   \\____ \_/ __ \ /    \ |    |  \\__  \<   |  ||  | |  |/ ___\|  |  \   __\
    /    |    \  |_> >  ___/|   |  \|    `   \/ __ \\___  ||  |_|  / /_/  >   Y  \  |
    \_______  /   __/ \___  >___|  /_______  (____  / ____||____/__\___  /|___|  /__|
            \/|__|        \/     \/        \/     \/\/            /_____/      \/


Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

~
opendaylight-user@    >
```

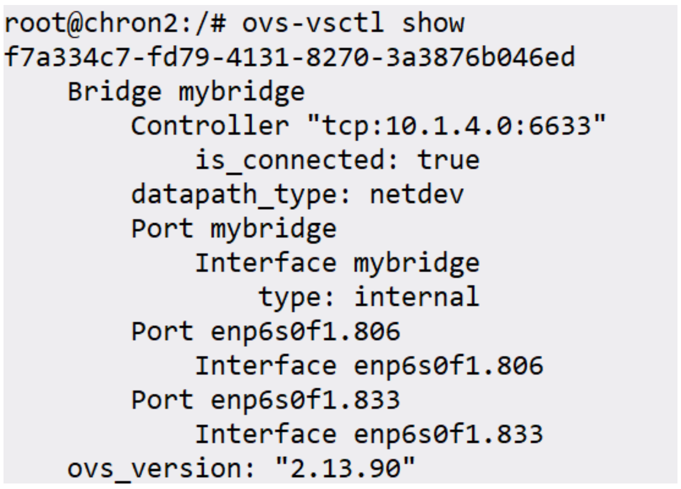
Figure 54 : First Start of the Open Daylight

4.6 Connection between the Open vSwitch and ODL

The connection between the controller and the OVS can be done in two steps:

- First step: On the OVS command line we set up the IP of the ODL container as its master. This is done with the command *ovs-vsctl set-controller mybridge tcp:10.1.4.0:6633*.
- Second Step: We start the ODL inside the container by navigating to the directory *opendaylight-0.11.2* and running the script *./bin/karaf* inside the directory the ODL has been installed into. This starts the ODL as it can also be seen in Figure 54.

After this process the connection between controller and the switch needs to be tested by just pinging to each other. If the SDN controller can ping to the interface (host interface in this case) this means the controller has connected to the OVS and taken over the functions from the switch. This can be checked by using the *ovs-vsctl show* command on the open vSwitch, as it is shown in Figure 55. In this figure the *"is_connected"* option is *true*, which means the controller is now the master of the switch and responsible for further flows creations and configurations.



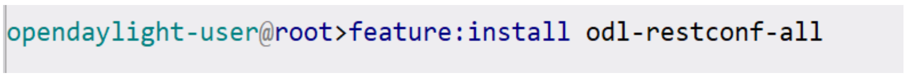
```
root@chron2:/# ovs-vsctl show
f7a334c7-fd79-4131-8270-3a3876b046ed
    Bridge mybridge
        Controller "tcp:10.1.4.0:6633"
            is_connected: true
        datapath_type: netdev
        Port mybridge
            Interface mybridge
                type: internal
        Port enp6s0f1.806
            Interface enp6s0f1.806
        Port enp6s0f1.833
            Interface enp6s0f1.833
    ovs_version: "2.13.90"
```

Figure 55 : Controller accepted by the switch

To be able to configure flows with ODL, some few features need to be installed on the controller. The list of available features can be seen with *feature:list* command on the ODL and the pre-installed features with *feature:list -i*. Moreover, the search for a specific feature can be done with *feature:list | grep <feature-name>*. In this case we only need the open-flow plugin and the restconf [32], that provides the next functionalities:

- The *OpenFlow-plugin* provides the following functions: flow management, group management, meter management and statistics polling
- The *restconf* allows us to access to the RESTCONF API. The RESTCONF protocol allows web applications to be used to access data inside a network element. In this case the Open vSwitch.

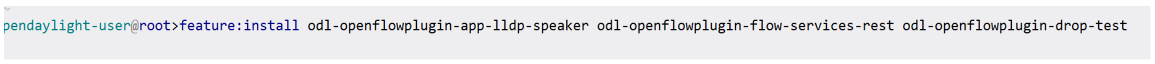
The installation of *restconf* is simple using the command *feature:install odl-restconf-all* as it is done in Figure 56.



```
opendaylight-user@root>feature:install odl-restconf-all
```

Figure 56 : feature installation restconf

The OpenFlow plugin can be installed in a similar way with the command: *feature:install odl-openflowplugin-app-lldp-speaker odl-openflowplugin-flow-services-rest odl-openflowplugin-drop-test* as it is done in Figure 57.



```
pendaylight-user@root>feature:install odl-openflowplugin-app-lldp-speaker odl-openflowplugin-flow-services-rest odl-openflowplugin-drop-test
```

Figure 57: feature installation OpenFlow plugin

After the installation of these features, it is possible to view the topology and the current flow entries that have been created on the switch by surfing to the controller IP address. Normally, when the controller is being run on a virtual machine or in a physical machine, the switch information and flow configuration could be accessed on any web browser with the controller ip address. In contrast, in our case the controller is being run on a docker container and the docker container are isolated, so it needs one host port to be open. For more information about docker networking and opening ports refer to the Chapter 2 section about Docker. Then, in order to open one of host port for the docker container, extra port information is needed during the start process [17]. Then, in the next example, port 8181 is being opened on the host and the IP that will be used to surf during the switch configuration is also the port of the host that is connected to the internet. The IP address *10.1.4.0* will still be used as the controller address on the OVS but to view the REST API data on the browser, the IP address *10.0.103.73* will be used because this is the IP on the port/interface that the container will use to be able to launch on the web browser. Refer to Chapter 2 about Docker on more regarding opening host ports for containers.

```
docker run -t -i --cap-add SYS_ADMIN --network <docker-bridge-name> --
name=<container-name> -p 10.0.103.73:8181:8181 --privileged <image-name>
```

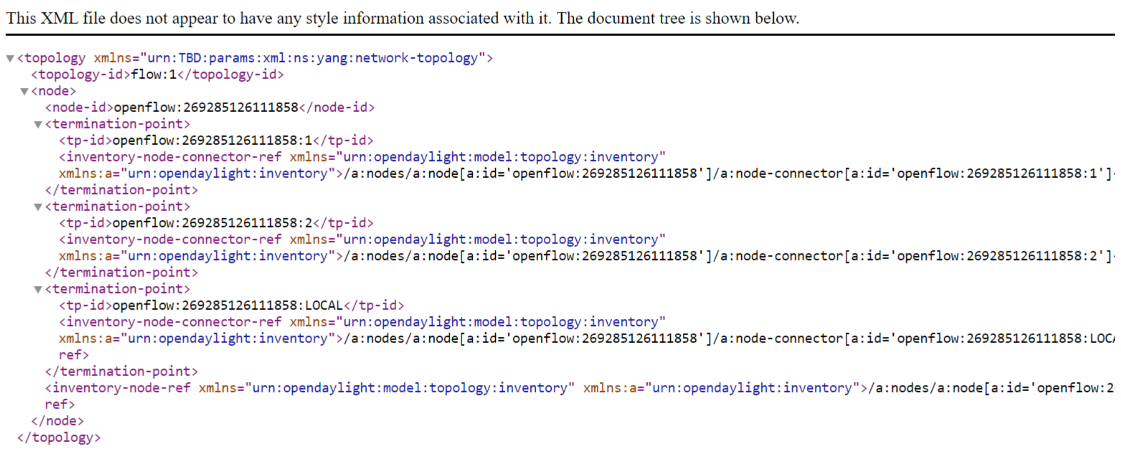
Therefore, the OVS still has 10.1.4.0 as its controller and the master but the container where the odl is installed uses the port with IP 10.0.103.73 to connect to the outside world so the IP 10.0.103.73 will be used during the flow configurations and other tests.

Now the connection part has been figured out, the next part is to find out the switch id or the node id. Most official documents indicate it has the *id openflow:1* but this is not always the case. Therefore, to access to the topology data of the switch the following link needs to be posted on any browser:

```
http://10.0.103.73:8181/restconf/operational/network-topology:network-topology/topolo
gy/flow:1
```

Putting the link on any browser (i.e. google chrome, Firefox, edge) uses the GET function and outputs the xml-format data of the switch topology, as it can be seen in Figure

58. This lets the user to see basic switch information that will be used in the further steps. If the user is asked for authentication *username: admin* and *password: admin* will suffice.



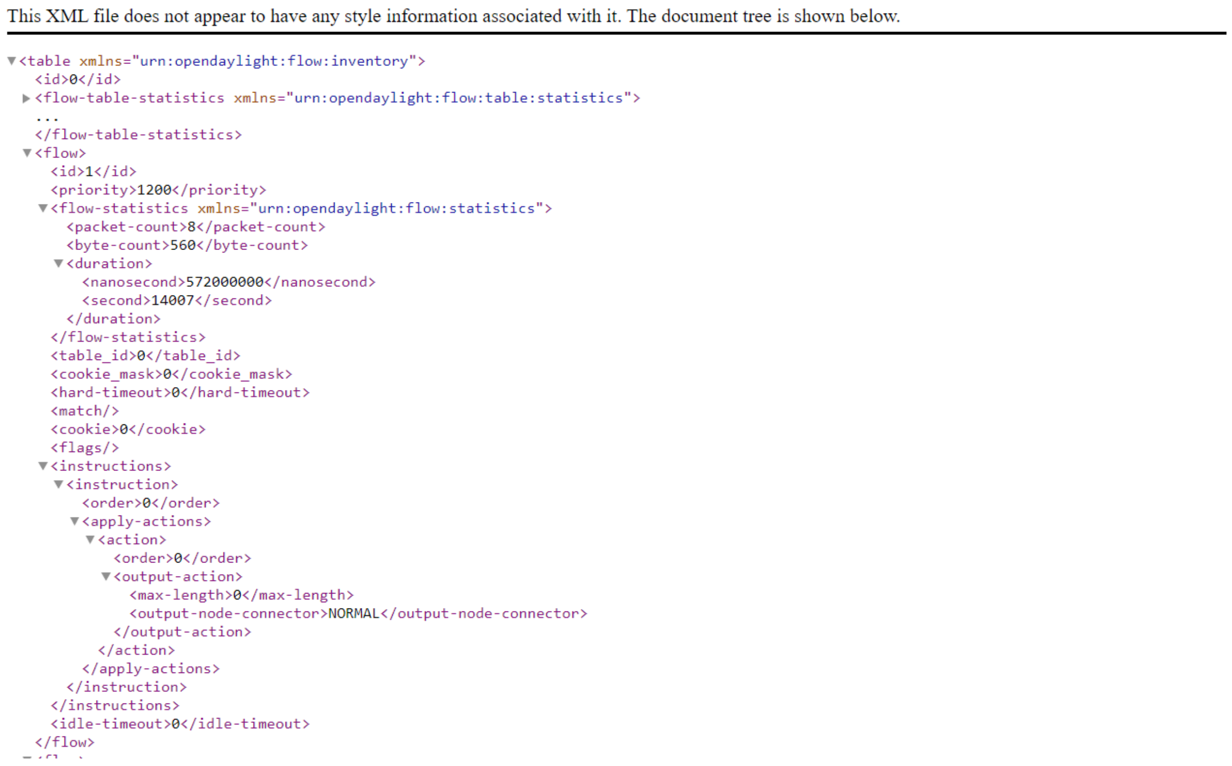
This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<topology xmlns="urn:TBD:params:xml:ns:yang:network-topology">
   <topology-id>flow:1</topology-id>
 ▼<node>
     <node-id>openflow:269285126111858</node-id>
   ▼<termination-point>
       <tp-id>openflow:269285126111858:1</tp-id>
       <inventory-node-connector-ref xmlns="urn:opendaylight:model:topology:inventory"
       xmlns:a="urn:opendaylight:inventory">/a:nodes/a:node[a:id='openflow:269285126111858']/a:node-connector[a:id='openflow:269285126111858:1']
     </termination-point>
   ▼<termination-point>
       <tp-id>openflow:269285126111858:2</tp-id>
       <inventory-node-connector-ref xmlns="urn:opendaylight:model:topology:inventory"
       xmlns:a="urn:opendaylight:inventory">/a:nodes/a:node[a:id='openflow:269285126111858']/a:node-connector[a:id='openflow:269285126111858:2']
     </termination-point>
   ▼<termination-point>
       <tp-id>openflow:269285126111858:LOCAL</tp-id>
       <inventory-node-connector-ref xmlns="urn:opendaylight:model:topology:inventory"
       xmlns:a="urn:opendaylight:inventory">/a:nodes/a:node[a:id='openflow:269285126111858']/a:node-connector[a:id='openflow:269285126111858:LOC
       ref>
     </termination-point>
     <inventory-node-ref xmlns="urn:opendaylight:model:topology:inventory" xmlns:a="urn:opendaylight:inventory">/a:nodes/a:node[a:id='openflow:2
     ref>
   </node>
 </topology>
```

Figure 58 : Switch topology data

The screen capture of the Figure 58 provides many valuable information, such as the node-id is *openflow:269285126111858* and there are two ports connected to this node (833 and 806 interfaces). The switch labels them as connector 1 and connector 2 in its database. Regarding each node named with the node id, the information about the flows that were created during the open vSwitch configuration can be accessed. This is done with the next command that displays all flow-entries on the switch as the output and it can be observed in Figure 59.

```
http://10.0.103.73:8181/restconf/operational/opendaylight-inventory:nodes/node/openflo
w:269285126111858/table/0
```



```
This XML file does not appear to have any style information associated with it. The document tree is shown below.

▼<table xmlns="urn:opendaylight:flow:inventory">
   <id>0</id>
  ▶<flow-table-statistics xmlns="urn:opendaylight:flow:table:statistics">
   ...
   </flow-table-statistics>
  ▼<flow>
     <id>1</id>
     <priority>1200</priority>
    ▼<flow-statistics xmlns="urn:opendaylight:flow:statistics">
       <packet-count>8</packet-count>
       <byte-count>560</byte-count>
      ▼<duration>
         <nanosecond>572000000</nanosecond>
         <second>14007</second>
       </duration>
     </flow-statistics>
     <table_id>0</table_id>
     <cookie_mask>0</cookie_mask>
     <hard-timeout>0</hard-timeout>
     <match/>
     <cookie>0</cookie>
     <flags/>
    ▼<instructions>
      ▼<instruction>
         <order>0</order>
        ▼<apply-actions>
          ▼<action>
             <order>0</order>
            ▼<output-action>
               <max-length>0</max-length>
               <output-node-connector>NORMAL</output-node-connector>
             </output-action>
           </action>
         </apply-actions>
       </instruction>
     </instructions>
     <idle-timeout>0</idle-timeout>
   </flow>
```

Figure 59 : Display flow entries in one specific node of the topology

In the screen capture of Figure 59 we can see one of the flows that is currently running on the switch with flow id: 1 and the related parameters of the flow i.e. priority :1200, table-id: 0.

On the other hand, the link that was used to access this data *restconf/operational/...* means that all data are operational, and they are running on the switch. This tree is read-only and to add new configurations to the switch, the *operational* in the link needs to be replaced by *config*. Then, the *config* tree allows the user to PUT, DEL and GET the information on the switch whereas the *operational* tree provides read-only feature [32]. Finally, in order to configure the flows and meters we will use the application POSTMAN [33].

4.7 Configuration of Postman with ODL

To be able to configure the flow entries on flow tables, it is very important to have a good understanding of how POSTMAN works. This part will only cover the installation and the configuration of the software so it can be used as we intend with the ODL controller. The application can be downloaded by simply clicking on the download button on the official website of POSTMAN as it can be seen on Figure 60. The installation process runs simple and smooth as any other application being installed on a windows operating system.

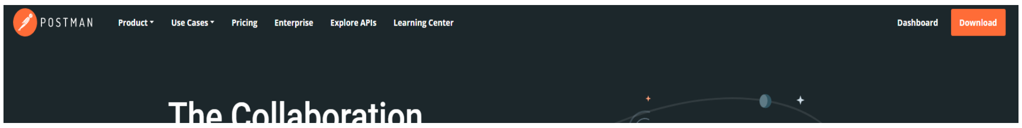


Figure 60 : Postman Homepage

After the installation has been successful, a new collection can be created by clicking on the new option available on the top left corner of the application as seen on Figure 61.

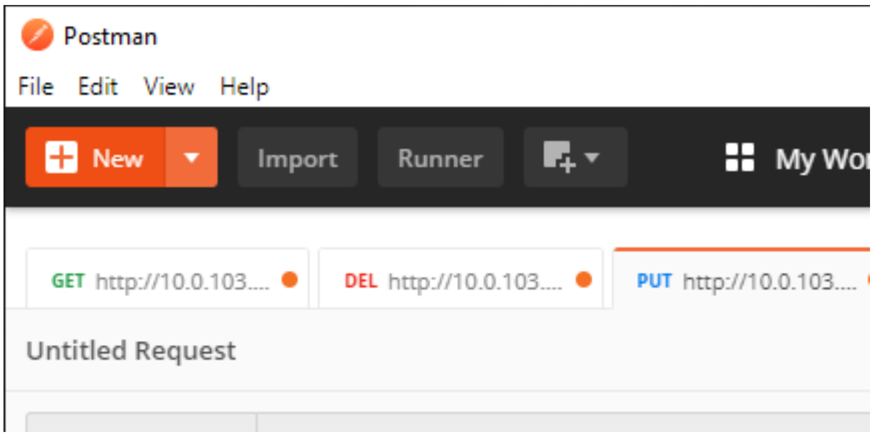
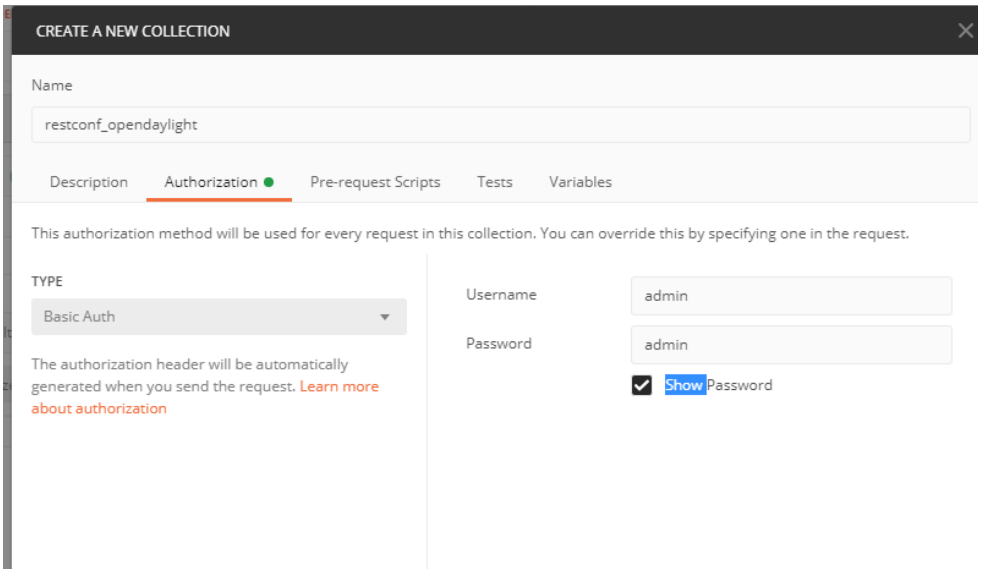


Figure 61 : Postman create new collection

The next step is to choose a name for the collection and select a basic authentication type, as it is done in Figure 62.


CREATE A NEW COLLECTION

Name: restconf_opendaylight

Description | Authorization | Pre-request Scripts | Tests | Variables

This authorization method will be used for every request in this collection. You can override this by specifying one in the request.

TYPE: Basic Auth

The authorization header will be automatically generated when you send the request. Learn more about authorization

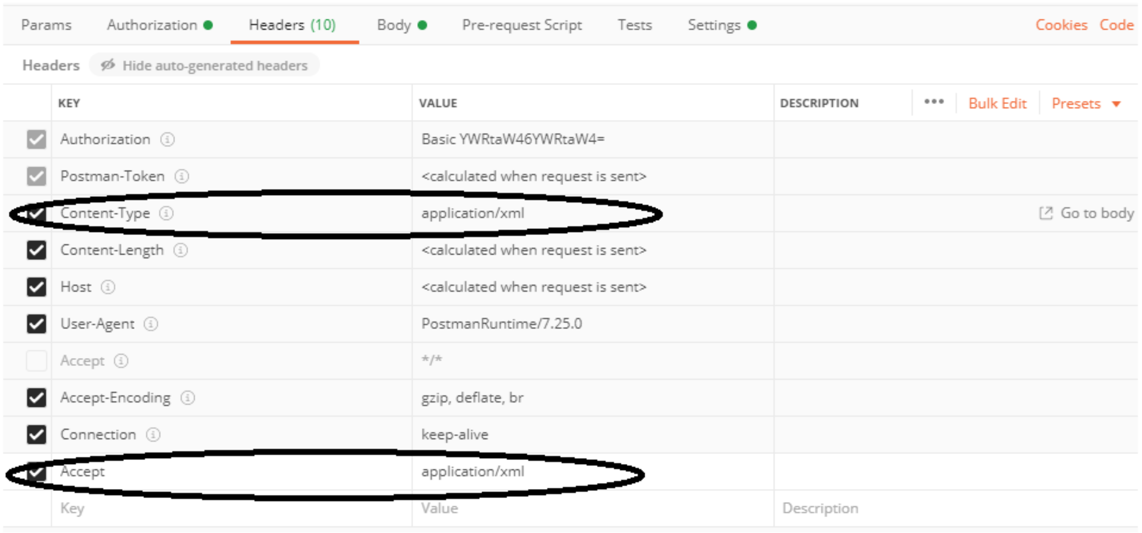
Username: admin

Password: admin

☑ Show Password

Figure 62 : Postman collection name Authentication

The final step to get the Postman working is to adjust the headers. Postman already contains some default headers, but two more needs to be added. As seen in Figure 63 *Content-Type: application/xml* and *Accept: application/xml* have been added to the headers. The *Content-type* tells Postman that the user will send data to the rest API in XML format and the *Accept* lets the Postman know that the user is accepting any result it acquires from the REST API using the GET function in XML format as well.


Params | Authorization | Headers (10) | Body | Pre-request Script | Tests | Settings | Cookies | Code

Headers | Hide auto-generated headers

	KEY	VALUE	DESCRIPTION
✓	Authorization	Basic YWRtaW46YWRtaW4=	
✓	Postman-Token	<calculated when request is sent>	
✓	Content-Type	application/xml	
✓	Content-Length	<calculated when request is sent>	
✓	Host	<calculated when request is sent>	
✓	User-Agent	PostmanRuntime/7.25.0	
	Accept	*/*	
✓	Accept-Encoding	gzip, deflate, br	
✓	Connection	keep-alive	
✓	Accept	application/xml	
	Key	Value	Description

Figure 63 : Postman Headers

After the previous steps, Postman is ready to be used and as it can be noticed in the Figure 64, we can insert the URL that was used earlier on a browser to GET, PUT, or DELETE any data to the switch. Together with the field for the link, Postman shows two bodies as it can be seen in Figure 64. The upper body is where the data that needs to be inserted to the API is typed, and the lower body is where the GET functions outputs the data. Basic information such as the time the function takes to execute or status messages (200 ok, 404 errors) are also visible on the screen capture.

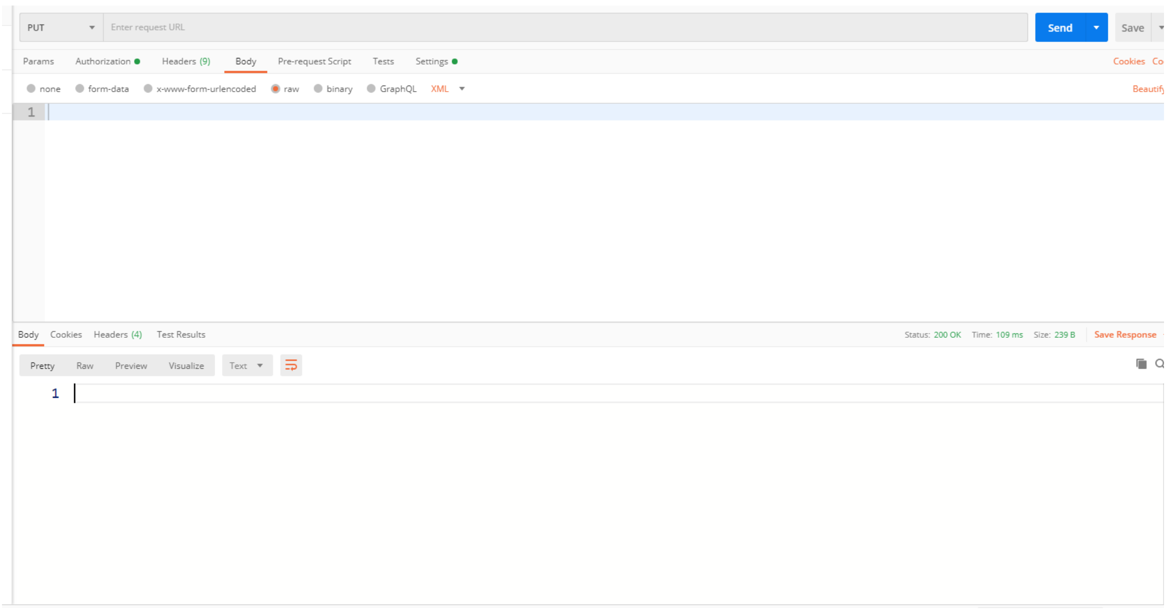


Figure 64 : Post man upper and lower Body

4.8 Configuration of flows and meters in ODL with Postman

Before creating flow entries, it is necessary to understand how a flow entry is built. Therefore, each flow table entry contains [34] the next components:

- *Match fields*: to match against packets. These consist of the ingress port and packet headers, and optionally metadata specified by a previous table.
- *Priority*: matching precedence of the flow entry.
- *Counters*: updated when packets are matched.
- *Instructions*: to modify the action set or pipeline processing.
- timeouts: maximum amount of time or idle time before one flow is expired by the switch.

- *Cookie*: opaque data value chosen by the controller. May be used by the controller to filter flow statistics, flow modification and flow deletion. It is not used when processing packets.

A flow table entry is identified by its match fields and priority, so the match fields and the priority taken together identify a unique flow entry in the flow table [34]. The proper working of flow can also be observed in the flowchart available as Figure 65.

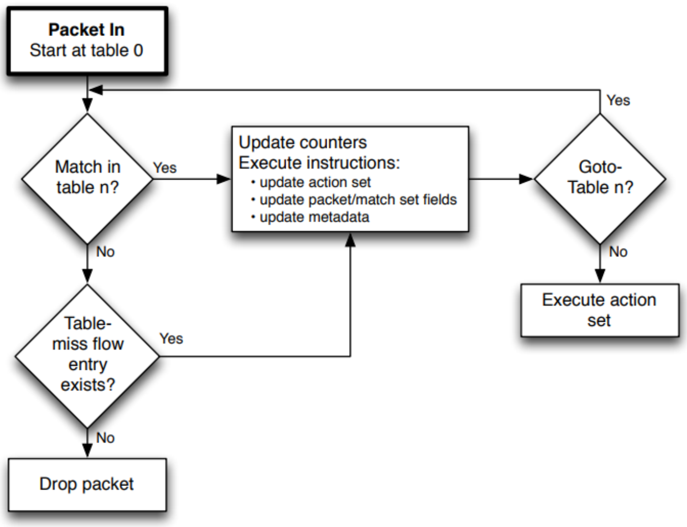


Figure 65 : Flowchart detailing packet flow through an OpenFlow switch

In the next steps the user will create some flow entries and send them via postman to the switch.

4.8.1 *Configuration of normal flows*

During the early stages we talked about the *operational* tree and the *config* tree to access the flow data and configure the flows. *Operational* tree allows us to view the flows currently running on the switch and *config* allows the user to configure or add new flows. Using GET function to output those operational flows on the Postman can be very useful so those flows created on the switch can be used as templates to create new flows via the *config* tree. In order to create a *normal* flow with XML format using Postman we must insert some code as the next one:

```
<flow xmlns="urn:opendaylight:flow:inventory">
  <id>1</id>
```

```
<priority>1200</priority>
<table_id>0</table_id>
<cookie_mask>0</cookie_mask>
<hard-timeout>0</hard-timeout>
<match/>
<cookie>0</cookie>
<flags/>
<instructions>
  <instruction>
    <order>0</order>
    <apply-actions>
      <action>
        <order>0</order>
        <output-action>
          <max-length>0</max-length>
          <output-node-connector>NORMAL</output-node-connector>
        </output-action>
      </action>
    </apply-actions>
  </instruction>
 </instructions>
<idle-timeout>0</idle-timeout>
     </flow>
```

The above flow, also known as the normal flow entry, does nothing special other than letting all packets through its port. Its destination is flow 1 in the table 0. It has 1200 priority value and no meters or timeouts associated. The xml-data can be entered in the upper body on the postman, then the PUT option is selected and the send button will forward the flow to the switch as it can be observed in Figure 66:

http://10.0.103.73:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:269285126111858/table/0/flow/1

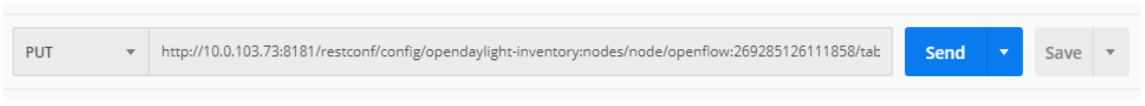
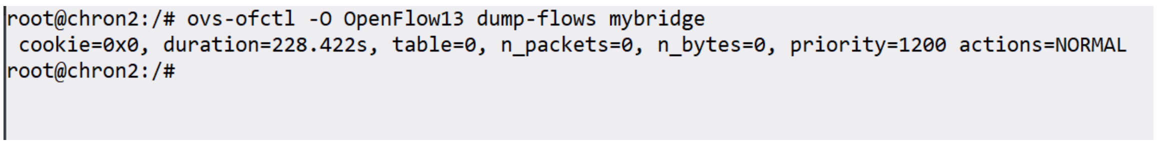

PUT
http://10.0.103.73:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:269285126111858/tab
Send
Save

Figure 66 : PUT flow into the table

To check if the switch accepts the flow, we simply ask for the open-vSwitch to display its current flows with the command *ovs-ofctl -O OpenFlow13 dump-flows mybridge* and we can observe the existing flow (Figure 67) of priority 1200.



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x0, duration=228.422s, table=0, n_packets=0, n_bytes=0, priority=1200 actions=NORMAL
root@chron2:/#
```

Figure 67 : Dump all flows on the bridge

In this case, the switch has accepted the configuration sent from the controller. It is possible that the switch could be incompatible with templates found on the official ODL documents, but it could be solved with the creation of a flow entry from the switch itself like it was done earlier with the command *ovs-ofctl -O OpenFlow13 add-flow mybridge actions=normal*. This command creates a flow in the *operational* tree of the restconfig which then can be used as a default template for the further creation of flows in the *config* tree (See the section about the configuration of the OVS on Chapter3 to learn the difference between operational and config tree). This process is mentioned again because this is very important part during the creation of the flows and can cause for huge loss of time if not done correctly.

4.8.2 *Configuration of flows to forward packets for specific ports*

The normal flow template is used again as a template and the match and instructions fields are replaced with the corresponding xml-code. The flow id in the xml-code and the postman link also needs to be changed accordingly. For example, if the flow id is 2, the link should look like this:

```
http://10.0.103.73:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:26
9285126111858/table/0/flow/2
```

Then, the example of this flow is to send packets from connector 1 to connector 2, in this case enp6s0f1.833 to enp6s0f1.806 in. Then, the next flow entry forwards any packets from 833 interface to 806:

```
<match>
      <in-port>1</in-port>
    </match>
    <cookie>0</cookie>
    <flags/>
    <instructions>
      <instruction>
        <order>0</order>
        <apply-actions>
          <action>
            <order>0</order>
            <output-action>
              <max-length>0</max-length>
              <output-node-connector>2</output-node-connector>
            </output-action>
          </action>
        </apply-actions>
      </instruction>
```

```
</instructions>
```

A similar flow entry needs to be created if we want the packets coming through the 806 interface to be forwarded to the 833 interface. This can be done by creating a new flow with same input but changing the flow id to 3 in-port to 2 and the output-node-connector to 1 in the previous template, in the next way:

```
<id>3</id>
<in-port>2</in-port>
<output-node-connector>1</output-node-connector>
```

4.8.3 *Configuration of flows from one specific Mac Address*

The same template from normal flow is used again, but the match and the action fields are replaced using the next xml code:

```
<match>
    <ethernet-match>
        <ethernet-source>
            <address>f4:e9:d4:49:02:72</address>
        </ethernet-source>
        <ethernet-destination>
            <address>f4:e9:d4:48:f7:b2</address>
        </ethernet-destination>
    </ethernet-match>
</match>
<cookie>0</cookie>
<flags/>
<instructions>
    <instruction>
        <order>0</order>
        <apply-actions>
            <action>
```

```
                <order>0</order>
                <output-action>
                    <max-length>0</max-length>
                    <output-node-connector>1</output-node-connector>
                </output-action>
            </action>
        </apply-actions>
    </instruction>
</instructions>
```

In the above flow all packets with the MAC source address f4:e9:d4:49:02:72 to destination f4:e9:d4:48:f7:b2 are forwarded to the connector 1, as the flow instructs the switch to output/forward the packets intended for that specific MAC address through node-connector 1. The Switch database saves the connected ports as Connector 1, 2 and so on in its database. The connector 1 is equivalent to port enp6s0f1.833 in this case. Finally, the priority can be changed if needed and the hard timeout can be added too. Hard timeout makes sure that the flow is deleted from the switch after specific amount of time in seconds.

4.8.4 *Configuration of flows for specific Network Address*

The template from normal flow is used again and the match field and instructions field are replaced in the xml-code. In fact, the flow applies to any packets destined for the network address 192.168.3.0/24, as this condition is inserted in the *ethernet-match* of the flow below:

```
<match>
    <ethernet-match>
        <ethernet-type>
            <type>2048</type>
        </ethernet-type>
    </ethernet-match>
    <ipv4-destination>192.168.3.0/24</ipv4-destination>
</match>
<cookie>1</cookie>
<flags/>
<instructions>
    <instruction>
        <order>0</order>
        <apply-actions>
            <action>
                <order>0</order>
                <output-action>
                    <max-length>0</max-length>
                    <output-node-connector>1</output-node-connector>
                </output-action>
            </action>
        </apply-actions>
    </instruction>
```

</instructions>

At this moment the flow instructs the switch to output the packets destined for any IP's in the network 192.168.3.0/24 through the node-connector 1 (this is the same node as in the previous example). However, more instructions like meters and timeouts can be added in the future to test the working of it in details.

All the flows created earlier can be edited and changed if the priority, timeout or flow id needs to be changed. The PUT function on the Postman replaces the older one with the newer flow entry if the link matches. Finally, deleting a flow entry is also possible by selecting the DELETE option on postman as it can be observed in Figure 68.

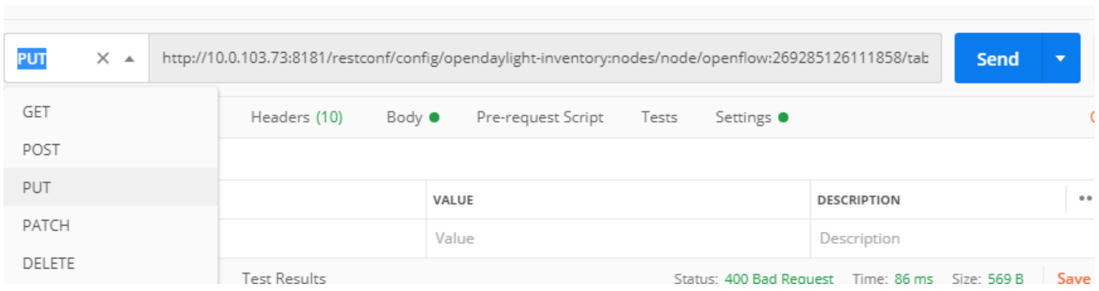


Figure 68 : Different options available i.e. put, delete and get

4.8.5 *Configuration of meters in ODL*

To create a meter a different link is used on the Postman. The PUT function is still at work but the link goes as follows:

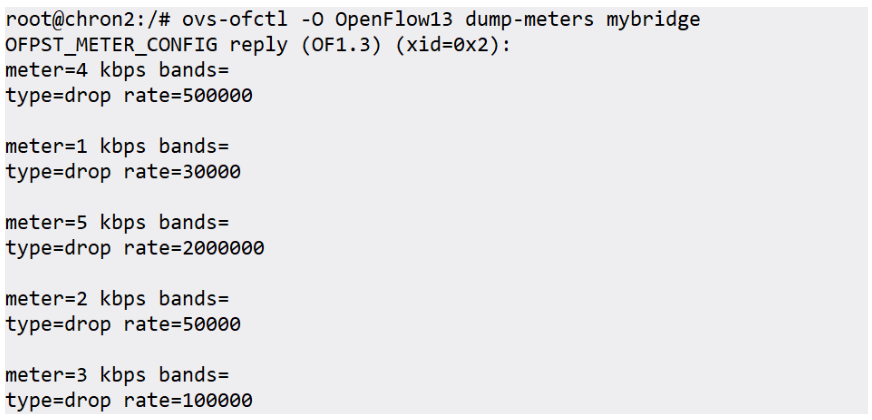
http://10.0.103.73:8181/restconf/config/opendaylight-inventory:nodes/node/openflow:269285126111858/meter/1

Then, the corresponding xml-code to create a meter is the next one:

```
<meter xmlns="urn:opendaylight:flow:inventory">
    <meter-id>1</meter-id>
    <meter-band-headers>
      <meter-band-header>
        <band-id>0</band-id>
        <meter-band-types>
          <flags>ofpmbt-drop</flags>
```

```
            </meter-band-types>
            <drop-rate>30000</drop-rate>
            <drop-burst-size>30000</drop-burst-size>
        </meter-band-header>
    </meter-band-headers>
    <flags>meter-kbps</flags>
</meter>
```

The above meter has been configured with the id "1" and it is limiting any packets going through it to 30000 kbps (30 Mbps). It is also necessary to check if the switch has accepted the new meter with the command *ovs-ofctl -O OpenFlow13 dump-meters mybridge* and it is shown in Figure 69.



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-meters mybridge
OFPST_METER_CONFIG reply (OF1.3) (xid=0x2):
meter=4 kbps bands=
type=drop rate=500000

meter=1 kbps bands=
type=drop rate=30000

meter=5 kbps bands=
type=drop rate=2000000

meter=2 kbps bands=
type=drop rate=50000

meter=3 kbps bands=
type=drop rate=100000
```

Figure 69 : Newly created meters check on the vSwitch

For the test purposes few more meters with higher drop-rate shall be created. It is necessary to change the meter id in the xml-code and the link on the postman in order to create new meters. The meter 2 will have */meter/2* in its link and so on. Five meters have been created for the test purpose starting from 50 Mbps and going up to 5 Gbps.

Unlike the flow entries, changing data inside a meter cannot be done with the PUT function. The meter must be deleted with DELETE function on the postman first, if changes needs to be made on the meter. Once the existing meter has been deleted, we must change the xml-code and create the new meter with correct the inputs using the PUT option.

4.8.6 *Adding a meter to a configured flow*

To add the recently created meter to a flow entry, the following xml-snippet needs to be added in the instruction field:

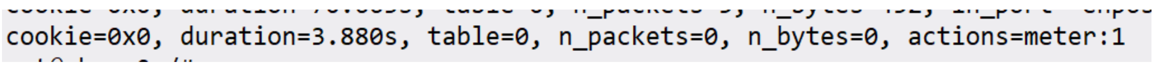
```
<instruction>
        <order>1</order>
                <meter>
                <meter-id>1</meter-id>
        </meter>
  </instruction>
```

Therefore, a normal flow with meter id “1” will look as follows the meter has been added to it (in that case the meter that has been configured previously):

```
<flow xmlns="urn:opendaylight:flow:inventory">
  <id>1</id>
 <priority>32768</priority>
  <table_id>0</table_id>
  <cookie_mask>0</cookie_mask>
  <hard-timeout>0</hard-timeout>
  <match/>
  <cookie>0</cookie>
  <flags/>
  <instructions>
    <instruction>
      <order>0</order>
      <apply-actions>
        <action>
          <order>0</order>
          <output-action>
```

```
                        <max-length>0</max-length>
                        <output-node-connector>NORMAL</output-node-connector>
                      </output-action>
                    </action>
                  </apply-actions>
                </instruction>
                <instruction>
                        <order>1</order>
                        <meter>
                   <meter-id>1</meter-id>
                        </meter>
                        </instruction>
              </instructions>
              <idle-timeout>0</idle-timeout>
            </flow>
```

If the flow has been configured with meter "1" it can also be checked on the open vSwitch with the command *ovs-ofctl -O OpenFlow13 dump-flows mybridge* as it can be observed in Figure 70. This confirms that the normal flow has been configured with meter '1' (*actions=meter:1*) as it can be observed in Figure 70.



```
cookie=0x0, duration=3.880s, table=0, n_packets=0, n_bytes=0, actions=meter:1
```

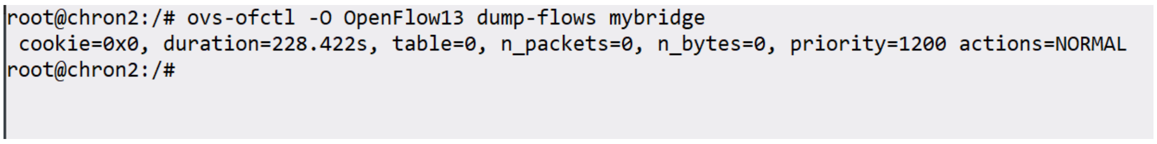
Figure 70 : Output with dump-flows command

4.9 Tests with flows and meters in ODL

In the next steps, we will test the performance of a set of flows and meters. This will be done step by step in the same order as in the previous section.

4.9.1 *Test 1: Normal flows created in the OVS*

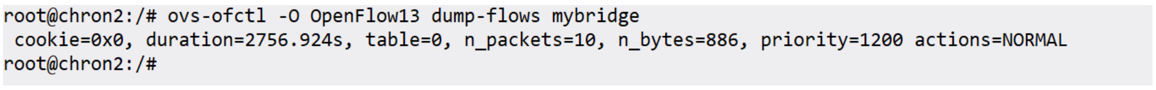
Firstly, it is necessary to checked if the switch has accepted the flow entry as it is shown in Figure 71, in that case a normal flow with priority 1200.



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x0, duration=228.422s, table=0, n_packets=0, n_bytes=0, priority=1200 actions=NORMAL
root@chron2:/#
```

Figure 71 : normal flow test

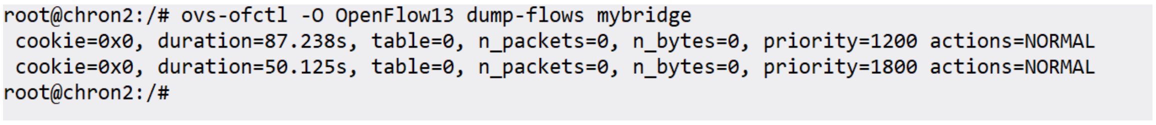
After the flow entry has been confirmed, a test can be done by pinging the other side of the interface to check the statistics of packets in the flow (Figure 72). In that figure it can be noticed that the flow entry has been hit and 10 packets passed through the bridge.



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x0, duration=2756.924s, table=0, n_packets=10, n_bytes=886, priority=1200 actions=NORMAL
root@chron2:/#
```

Figure 72 : normal flow statistics after the ping

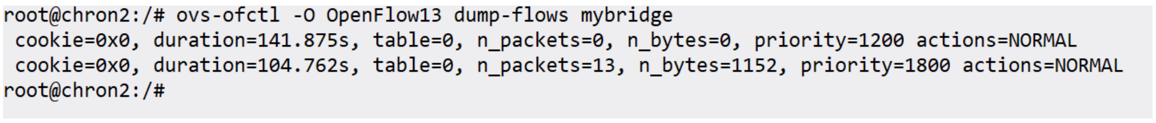
In the next step the priority of a flow entry will be checked. A normal flow has been created (copy of the first one), but this time the flow has a higher priority of 1800, so we have two different flows with different priority (Figure 73).



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x0, duration=87.238s, table=0, n_packets=0, n_bytes=0, priority=1200 actions=NORMAL
 cookie=0x0, duration=50.125s, table=0, n_packets=0, n_bytes=0, priority=1800 actions=NORMAL
root@chron2:/#
```

Figure 73 : Two flows created with different priority

The same interface is pinged again to look for changes. After the pings, it can be seen on Figure 74 that the flow with the higher priority (1800) is being used since packets are being only transferred through this flow entry.

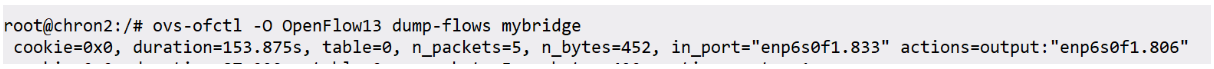


```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x0, duration=141.875s, table=0, n_packets=0, n_bytes=0, priority=1200 actions=NORMAL
 cookie=0x0, duration=104.762s, table=0, n_packets=13, n_bytes=1152, priority=1800 actions=NORMAL
root@chron2:/#
```

Figure 74 : Flow stats with different priority

4.9.2 *Test 2: Flow entry for specific ports*

In this flow entry, the switch was asked to forward packets from *enp6s0f1.833* through the *enp6s0f1.806* port. Then, it is necessary to check if the switch has accepted the flow. After the previous condition has been confirmed, the port is then pinged, and the results are shown in Figure 75. Therefore, it can be seen on flow the statistics that the flow entry has been hit (there are packets though this flow), so the flow entry configuration is correct.

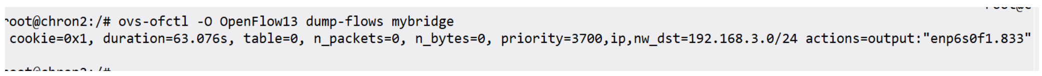


```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x0, duration=153.875s, table=0, n_packets=5, n_bytes=452, in_port="enp6s0f1.833" actions=output:"enp6s0f1.806"
```

Figure 75 : Flow entry check for specific port

4.9.3 *Test 3: Flow entry for specific network address*

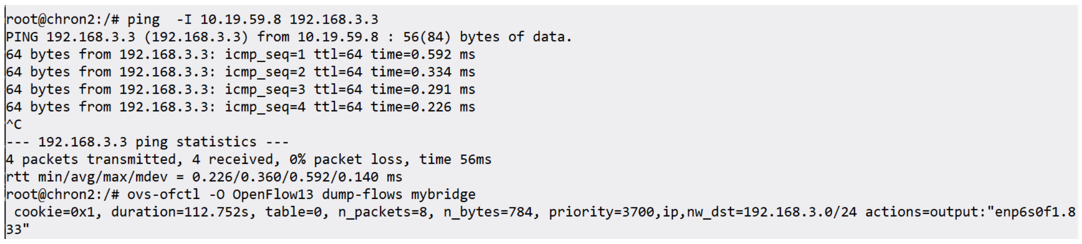
In the first step we must check if the flow entry has been accepted by the switch using the command shown in Figure 76.



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x1, duration=63.076s, table=0, n_packets=0, n_bytes=0, priority=3700,ip,nw_dst=192.168.3.0/24 actions=output:"enp6s0f1.833"
```

Figure 76 : Flow entry for specific Network Address

After the flow entry has been conformed, the network 192.168.3.0 is pinged to see if the flow entry works as intended. Figure 77 confirms that the flow entry works as intended since the flow permits packets to pass through.



```
root@chron2:/# ping  -I 10.19.59.8 192.168.3.3
PING 192.168.3.3 (192.168.3.3) from 10.19.59.8 : 56(84) bytes of data.
64 bytes from 192.168.3.3: icmp_seq=1 ttl=64 time=0.592 ms
64 bytes from 192.168.3.3: icmp_seq=2 ttl=64 time=0.334 ms
64 bytes from 192.168.3.3: icmp_seq=3 ttl=64 time=0.291 ms
64 bytes from 192.168.3.3: icmp_seq=4 ttl=64 time=0.226 ms
^C
--- 192.168.3.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 56ms
rtt min/avg/max/mdev = 0.226/0.360/0.592/0.140 ms
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x1, duration=112.752s, table=0, n_packets=8, n_bytes=784, priority=3700,ip,nw_dst=192.168.3.0/24 actions=output:"enp6s0f1.8
33"
```

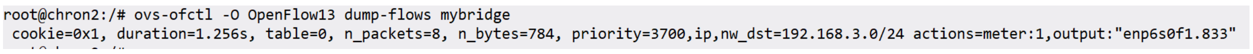
Figure 77 : Flow statistics check after the pings

4.9.4 *Test 4: A flow entry with different meters*

As mentioned earlier, five meters have been created on the switch to do several tests. In the next steps the flow is configured with different meters and the performance of meters are tested. The drop rate of meters is configured with meter ID 1, 2, 3, 4, 5 with limited rates of 30 Mbps, 50 Mbps, 100 Mbps, 500 Mbps and 5 Gbps, respectively.

4.9.4.1 Performance of meters of 30 Mbps

The flow entry has been configured with meter with identifier 1 (id) that correspond with a limited drop rate of 30 Mbps as it can be observed in Figure 78.

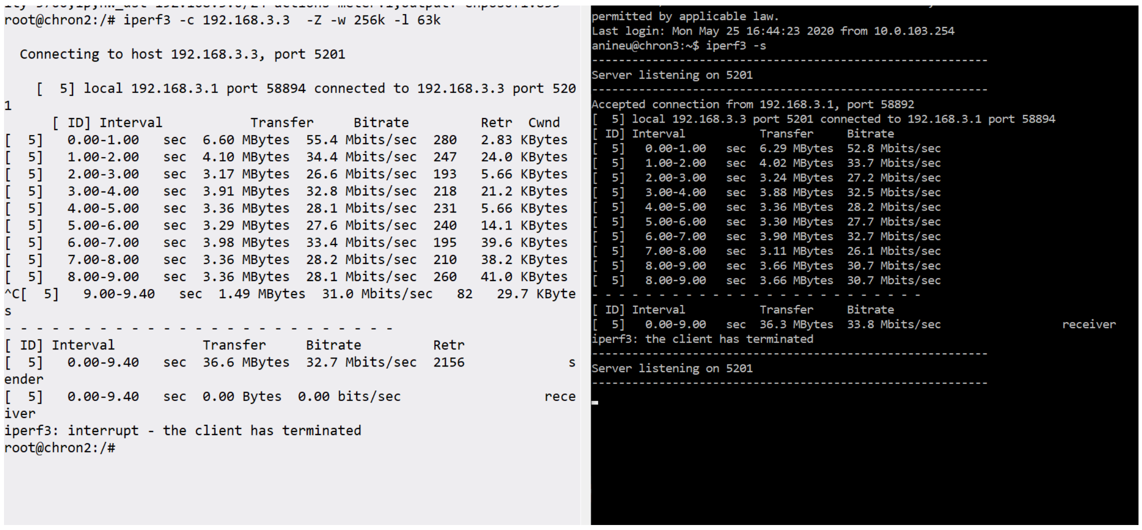


```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x1, duration=1.256s, table=0, n_packets=8, n_bytes=784, priority=3700,ip,nw_dst=192.168.3.0/24 actions=meter:1,output:"enp6s0f1.833"
```

Figure 78 : Flow entry with Meter 1

The throughput is then tested with a tool named iperf3. To test the through put with iperf3 we need a client side and the server side. The virtual interface on chron2 will be used as the client-side to start the iperf and the interfaces on the Chron3 will be used as the server-side to receive the data sent from the client (Figure 79). Refer to Chapter 3 on more information regarding the working of Iperf and its configurations.

The meters are not precise as it is still a new technology in development, but it is working as intended, since the throughput is around 30 Mbps as it is expected from the first meter (Figure 79.



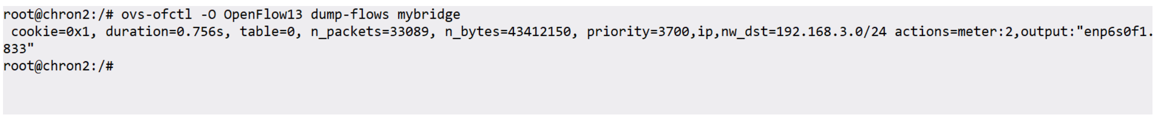
```
root@chron2:/# iperf3 -c 192.168.3.3  -Z -w 256k -l 63k

  Connecting to host 192.168.3.3, port 5201

    [  5] local 192.168.3.1 port 58894 connected to 192.168.3.3 port 520
1
      [ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  6.60 MBytes  55.4 Mbits/sec  280   2.83 KBytes
[  5]   1.00-2.00   sec  4.10 MBytes  34.4 Mbits/sec  247   24.0 KBytes
[  5]   2.00-3.00   sec  3.17 MBytes  26.6 Mbits/sec  193   5.66 KBytes
[  5]   3.00-4.00   sec  3.91 MBytes  32.8 Mbits/sec  218   21.2 KBytes
[  5]   4.00-5.00   sec  3.36 MBytes  28.1 Mbits/sec  231   5.66 KBytes
[  5]   5.00-6.00   sec  3.29 MBytes  27.6 Mbits/sec  240   14.1 KBytes
[  5]   6.00-7.00   sec  3.98 MBytes  33.4 Mbits/sec  195   39.6 KBytes
[  5]   7.00-8.00   sec  3.36 MBytes  28.2 Mbits/sec  210   38.2 KBytes
[  5]   8.00-9.00   sec  3.36 MBytes  28.1 Mbits/sec  260   41.0 KBytes
^C[  5]   9.00-9.40   sec  1.49 MBytes  31.0 Mbits/sec   82   29.7 KByte
s
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-9.40   sec  36.6 MBytes  32.7 Mbits/sec  2156             s
ender
[  5]   0.00-9.40   sec  0.00 Bytes  0.00 bits/sec                  rece
iver
iperf3: interrupt - the client has terminated
root@chron2:/#
```

```
permitted by applicable law.
Last login: Mon May 25 16:44:23 2020 from 10.0.103.254
anineu@chron3:~$ iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 192.168.3.1, port 58892
[  5] local 192.168.3.3 port 5201 connected to 192.168.3.1 port 58894
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  6.29 MBytes  52.8 Mbits/sec
[  5]   1.00-2.00   sec  4.02 MBytes  33.7 Mbits/sec
[  5]   2.00-3.00   sec  3.24 MBytes  27.2 Mbits/sec
[  5]   3.00-4.00   sec  3.88 MBytes  32.5 Mbits/sec
[  5]   4.00-5.00   sec  3.36 MBytes  28.2 Mbits/sec
[  5]   5.00-6.00   sec  3.30 MBytes  27.7 Mbits/sec
[  5]   6.00-7.00   sec  3.90 MBytes  32.7 Mbits/sec
[  5]   7.00-8.00   sec  3.11 MBytes  26.1 Mbits/sec
[  5]   8.00-9.00   sec  3.66 MBytes  30.7 Mbits/sec
[  5]   8.00-9.00   sec  3.66 MBytes  30.7 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-9.00   sec  36.3 MBytes  33.8 Mbits/sec                  receiver
iperf3: the client has terminated
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 79 : perf3 test with Meter 1 (30 Mbps)

4.9.4.2 Performance of meters of 50 Mbps

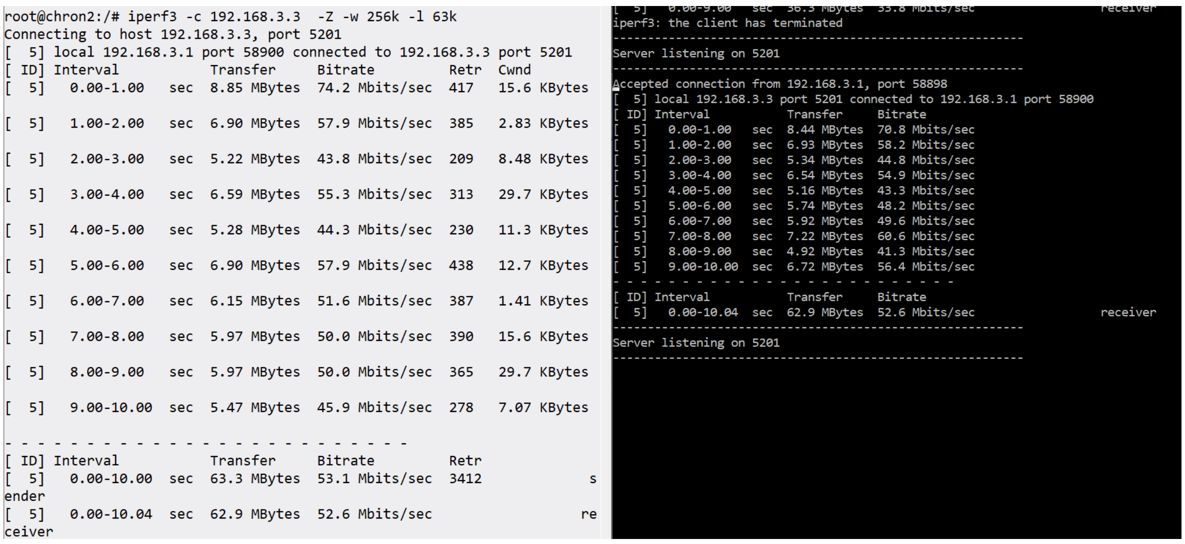
The flow is then configured with meter 2 of a higher data rate of 50Mbps, as it can be observed in Figure 80.



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x1, duration=0.756s, table=0, n_packets=33089, n_bytes=43412150, priority=3700,ip,nw_dst=192.168.3.0/24 actions=meter:2,output:"enp6s0f1.
833"
root@chron2:/#
```

Figure 80 : Flow entry with Meter 2 (50 Mbps)

We make sure if the switch has accepted the changes on the flow. After the conformation we can run the same iperf process again. The meter 2 works as intended limiting the throughput around 50 Mbps, as this performance can be observed in Figure 81.



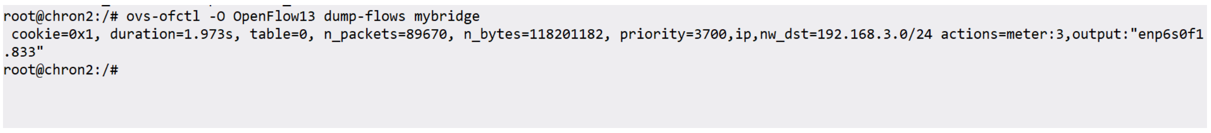
```
root@chron2:/# iperf3 -c 192.168.3.3  -Z -w 256k -l 63k
Connecting to host 192.168.3.3, port 5201
[  5] local 192.168.3.1 port 58900 connected to 192.168.3.3 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  8.85 MBytes  74.2 Mbits/sec  417   15.6 KBytes

[  5]   1.00-2.00   sec  6.90 MBytes  57.9 Mbits/sec  385   2.83 KBytes

[  5]   2.00-3.00   sec  5.22 MBytes  43.8 Mbits/sec  209   8.48 KBytes

[  5]   3.00-4.00   sec  6.59 MBytes  55.3 Mbits/sec  313   29.7 KBytes

[  5]   4.00-5.00   sec  5.28 MBytes  44.3 Mbits/sec  230   11.3 KBytes

[  5]   5.00-6.00   sec  6.90 MBytes  57.9 Mbits/sec  438   12.7 KBytes

[  5]   6.00-7.00   sec  6.15 MBytes  51.6 Mbits/sec  387   1.41 KBytes

[  5]   7.00-8.00   sec  5.97 MBytes  50.0 Mbits/sec  390   15.6 KBytes

[  5]   8.00-9.00   sec  5.97 MBytes  50.0 Mbits/sec  365   29.7 KBytes

[  5]   9.00-10.00  sec  5.47 MBytes  45.9 Mbits/sec  278   7.07 KBytes

- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  63.3 MBytes  53.1 Mbits/sec  3412             s
ender
[  5]   0.00-10.04  sec  62.9 MBytes  52.6 Mbits/sec                  re
ceiver
```

```
[  5]   0.00-9.00   sec  36.3 MBytes  33.8 Mbits/sec                  receiver
iperf3: the client has terminated
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 192.168.3.1, port 58898
[  5] local 192.168.3.3 port 5201 connected to 192.168.3.1 port 58900
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  8.44 MBytes  70.8 Mbits/sec
[  5]   1.00-2.00   sec  6.93 MBytes  58.2 Mbits/sec
[  5]   2.00-3.00   sec  5.34 MBytes  44.8 Mbits/sec
[  5]   3.00-4.00   sec  6.54 MBytes  54.9 Mbits/sec
[  5]   4.00-5.00   sec  5.16 MBytes  43.3 Mbits/sec
[  5]   5.00-6.00   sec  5.74 MBytes  48.2 Mbits/sec
[  5]   6.00-7.00   sec  5.92 MBytes  49.6 Mbits/sec
[  5]   7.00-8.00   sec  7.22 MBytes  60.6 Mbits/sec
[  5]   8.00-9.00   sec  4.92 MBytes  41.3 Mbits/sec
[  5]   9.00-10.00  sec  6.72 MBytes  56.4 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.04  sec  62.9 MBytes  52.6 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 81 : Iperf3 test with Meter 2 (50 Mbps)

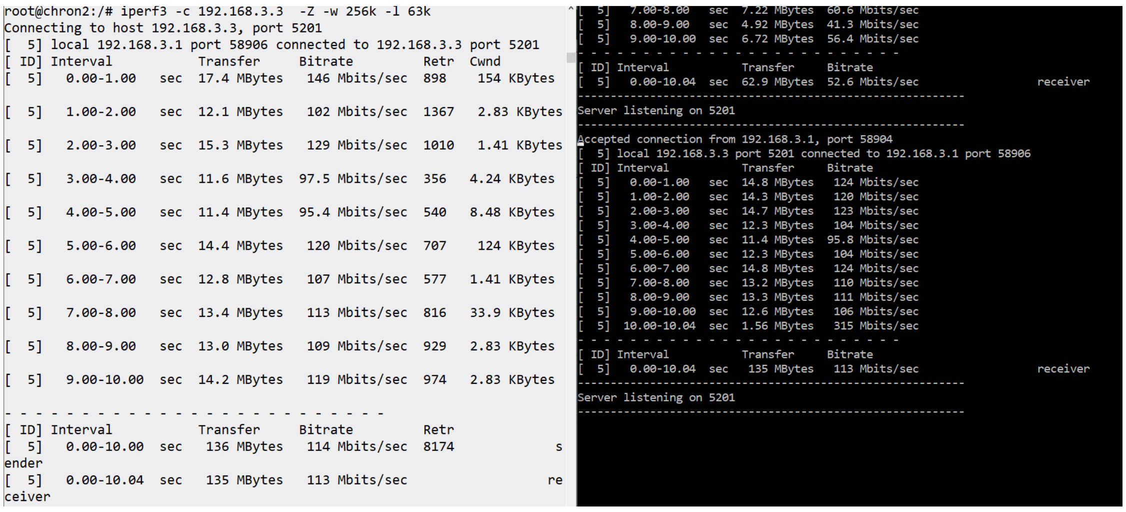
4.9.4.3 Performance of meters of 100 Mbps

The flow entry is configured with meter 3, related to 100Mbps (see Figure 82). The iperf is being run on the interface as it can be noticed in Figure 83. Moreover, Figure 83 shows that the throughput is around 100 Mbps as it was configured on the meter 3.



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x1, duration=1.973s, table=0, n_packets=89670, n_bytes=118201182, priority=3700,ip,nw_dst=192.168.3.0/24 actions=meter:3,output:"enp6s0f1
.833"
root@chron2:/#
```

Figure 82 : Flow entry with Meter 3 (100 Mbps)



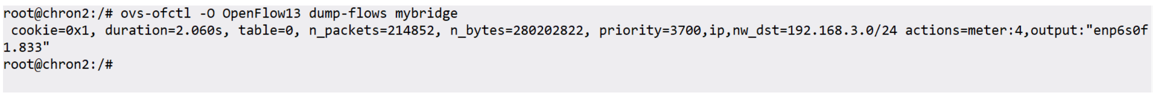
```
root@chron2:/# iperf3 -c 192.168.3.3  -Z -w 256k -l 63k
Connecting to host 192.168.3.3, port 5201
[  5] local 192.168.3.1 port 58906 connected to 192.168.3.3 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  17.4 MBytes   146 Mbits/sec  898    154 KBytes

[  5]   1.00-2.00   sec  12.1 MBytes   102 Mbits/sec  1367   2.83 KBytes

[  5]   2.00-3.00   sec  15.3 MBytes   129 Mbits/sec  1010   1.41 KBytes

[  5]   3.00-4.00   sec  11.6 MBytes  97.5 Mbits/sec  356    4.24 KBytes

[  5]   4.00-5.00   sec  11.4 MBytes  95.4 Mbits/sec  540    8.48 KBytes

[  5]   5.00-6.00   sec  14.4 MBytes   120 Mbits/sec  707    124 KBytes

[  5]   6.00-7.00   sec  12.8 MBytes   107 Mbits/sec  577    1.41 KBytes

[  5]   7.00-8.00   sec  13.4 MBytes   113 Mbits/sec  816    33.9 KBytes

[  5]   8.00-9.00   sec  13.0 MBytes   109 Mbits/sec  929    2.83 KBytes

[  5]   9.00-10.00  sec  14.2 MBytes   119 Mbits/sec  974    2.83 KBytes

- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec   136 MBytes   114 Mbits/sec  8174             s
ender
[  5]   0.00-10.04  sec   135 MBytes   113 Mbits/sec                  re
ceiver
```

```
[  5]   7.00-8.00   sec  7.22 MBytes  60.6 Mbits/sec
[  5]   8.00-9.00   sec  4.92 MBytes  41.3 Mbits/sec
[  5]   9.00-10.00  sec  6.72 MBytes  56.4 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.04  sec  62.9 MBytes  52.6 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 192.168.3.1, port 58904
[  5] local 192.168.3.3 port 5201 connected to 192.168.3.1 port 58906
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  14.8 MBytes   124 Mbits/sec
[  5]   1.00-2.00   sec  14.3 MBytes   120 Mbits/sec
[  5]   2.00-3.00   sec  14.7 MBytes   123 Mbits/sec
[  5]   3.00-4.00   sec  12.3 MBytes   104 Mbits/sec
[  5]   4.00-5.00   sec  11.4 MBytes  95.8 Mbits/sec
[  5]   5.00-6.00   sec  12.3 MBytes   104 Mbits/sec
[  5]   6.00-7.00   sec  14.8 MBytes   124 Mbits/sec
[  5]   7.00-8.00   sec  13.2 MBytes   110 Mbits/sec
[  5]   8.00-9.00   sec  13.3 MBytes   111 Mbits/sec
[  5]   9.00-10.00  sec  12.6 MBytes   106 Mbits/sec
[  5]  10.00-10.04  sec  1.56 MBytes   315 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.04  sec   135 MBytes   113 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 83 : Iperf3 with Meter 3 (100 Mbps)

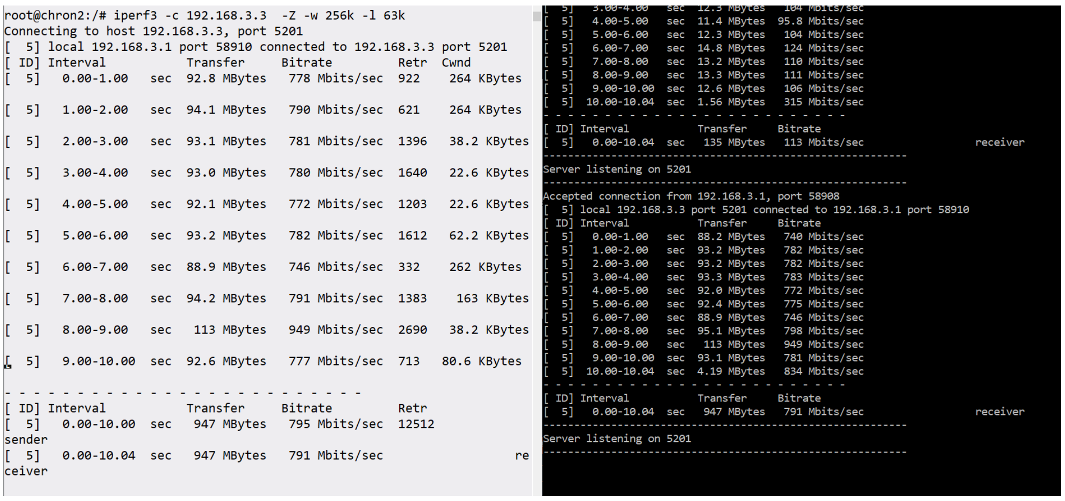
4.9.4.4 Performance of meters of 500 Mbps

The flow entry is configured with meter 4, that correspond with a maximum data rate of 500 Mbps, as it can be observed in Figure 84. The same iperf test is being run again and Figure 85 shows that meter permits more data through it (790 Mbps) that it is allowed. The meters start to get more and more inaccurate as the bit rate goes higher.



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x1, duration=2.060s, table=0, n_packets=214852, n_bytes=280202822, priority=3700,ip,nw_dst=192.168.3.0/24 actions=meter:4,output:"enp6s0f
1.833"
root@chron2:/#
```

Figure 84 : Flow entry with Meter 4 (500 Mpbs)



```
root@chron2:/# iperf3 -c 192.168.3.3  -Z -w 256k -l 63k
Connecting to host 192.168.3.3, port 5201
[  5] local 192.168.3.1 port 58910 connected to 192.168.3.3 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  92.8 MBytes   778 Mbits/sec  922    264 KBytes

[  5]   1.00-2.00   sec  94.1 MBytes   790 Mbits/sec  621    264 KBytes

[  5]   2.00-3.00   sec  93.1 MBytes   781 Mbits/sec  1396   38.2 KBytes

[  5]   3.00-4.00   sec  93.0 MBytes   780 Mbits/sec  1640   22.6 KBytes

[  5]   4.00-5.00   sec  92.1 MBytes   772 Mbits/sec  1203   22.6 KBytes

[  5]   5.00-6.00   sec  93.2 MBytes   782 Mbits/sec  1612   62.2 KBytes

[  5]   6.00-7.00   sec  88.9 MBytes   746 Mbits/sec  332    262 KBytes

[  5]   7.00-8.00   sec  94.2 MBytes   791 Mbits/sec  1383    163 KBytes

[  5]   8.00-9.00   sec   113 MBytes   949 Mbits/sec  2690   38.2 KBytes

[  5]   9.00-10.00  sec  92.6 MBytes   777 Mbits/sec  713   80.6 KBytes

- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec   947 MBytes   795 Mbits/sec  12512
sender
[  5]   0.00-10.04  sec   947 MBytes   791 Mbits/sec                  re
ceiver
```

```
[  5]   3.00-4.00   sec  12.3 MBytes   104 Mbits/sec
[  5]   4.00-5.00   sec  11.4 MBytes  95.8 Mbits/sec
[  5]   5.00-6.00   sec  12.3 MBytes   104 Mbits/sec
[  5]   6.00-7.00   sec  14.8 MBytes   124 Mbits/sec
[  5]   7.00-8.00   sec  13.2 MBytes   110 Mbits/sec
[  5]   8.00-9.00   sec  13.3 MBytes   111 Mbits/sec
[  5]   9.00-10.00  sec  12.6 MBytes   106 Mbits/sec
[  5]  10.00-10.04  sec  1.56 MBytes   315 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.04  sec   135 MBytes   113 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 192.168.3.1, port 58908
[  5] local 192.168.3.3 port 5201 connected to 192.168.3.1 port 58910
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  88.2 MBytes   740 Mbits/sec
[  5]   1.00-2.00   sec  93.2 MBytes   782 Mbits/sec
[  5]   2.00-3.00   sec  93.2 MBytes   782 Mbits/sec
[  5]   3.00-4.00   sec  93.3 MBytes   783 Mbits/sec
[  5]   4.00-5.00   sec  92.0 MBytes   772 Mbits/sec
[  5]   5.00-6.00   sec  92.4 MBytes   775 Mbits/sec
[  5]   6.00-7.00   sec  88.9 MBytes   746 Mbits/sec
[  5]   7.00-8.00   sec  95.1 MBytes   798 Mbits/sec
[  5]   8.00-9.00   sec   113 MBytes   949 Mbits/sec
[  5]   9.00-10.00  sec  93.1 MBytes   781 Mbits/sec
[  5]  10.00-10.04  sec  4.19 MBytes   834 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.04  sec   947 MBytes   791 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 85 : Iperf3 with Meter 4 (500 Mbps)

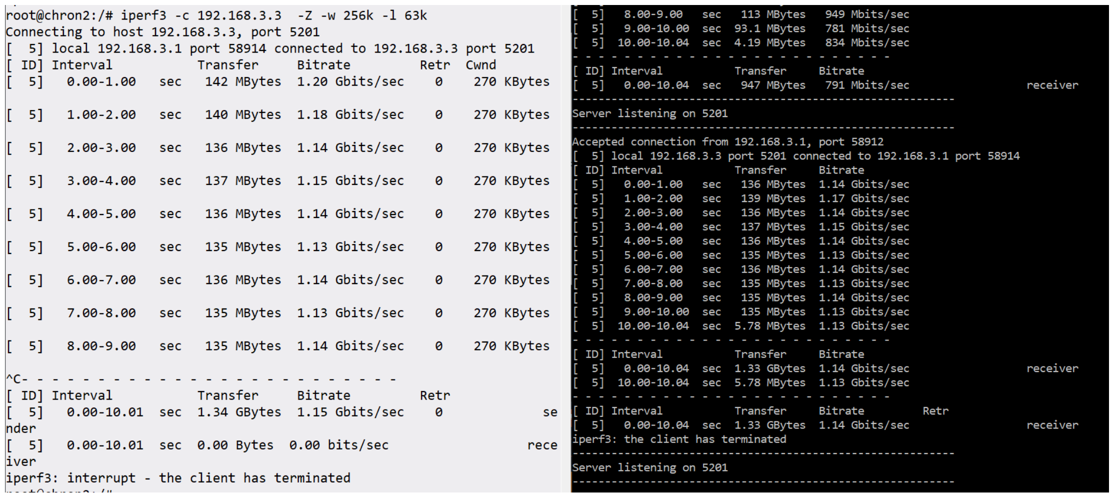
To check at what point the meters starts to act incorrectly, few more meters are created using the same methodology as earlier and the results are shown in the table below.

METER (Mbps)	**IPERF3: RESULTs (Mbps)**
30	33.8 Mbps
50	55 Mbps
100	118 Mbps
150	206 Mbps
200	329 Mbps
250	439 Mbps
300	561 Mbps
350	675 Mbps
400	790 Mbps
450	870 Mbps
500	940 Mbps
550	954 Mbps
600	1,10 Gbps

The results are good and precise in the lower bitrate but after 150Mbps they start getting less and less accurate. If this is related to the switch or the meters being new technology in general needs to be investigated.

4.9.4.5 Performance of meters of 5 Gbps

The flow entry is configured with meter 5 (5 Gbps). Figure 86 shows that when the meter is asked to let only 5 Gbps through the bridge, the throughput is capped at 1.14 Gbps. After multiple tests with higher bitrate meters (up to 10 Gbps) it can be confirmed that the switch itself does not allow higher throughput than 1.15 Gbps through its bridge. When there are no meters configured on the switch, the max throughput is still stuck at 1.15 Gbps. Whether the switch being run on the user space module is the cause behind it needs to be researched.



```
root@chron2:/# iperf3 -c 192.168.3.3  -Z -w 256k -l 63k
Connecting to host 192.168.3.3, port 5201
[  5] local 192.168.3.1 port 58914 connected to 192.168.3.3 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   142 MBytes  1.20 Gbits/sec    0    270 KBytes

[  5]   1.00-2.00   sec   140 MBytes  1.18 Gbits/sec    0    270 KBytes

[  5]   2.00-3.00   sec   136 MBytes  1.14 Gbits/sec    0    270 KBytes

[  5]   3.00-4.00   sec   137 MBytes  1.15 Gbits/sec    0    270 KBytes

[  5]   4.00-5.00   sec   136 MBytes  1.14 Gbits/sec    0    270 KBytes

[  5]   5.00-6.00   sec   135 MBytes  1.13 Gbits/sec    0    270 KBytes

[  5]   6.00-7.00   sec   136 MBytes  1.14 Gbits/sec    0    270 KBytes

[  5]   7.00-8.00   sec   135 MBytes  1.13 Gbits/sec    0    270 KBytes

[  5]   8.00-9.00   sec   135 MBytes  1.14 Gbits/sec    0    270 KBytes

^C- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.01  sec  1.34 GBytes  1.15 Gbits/sec    0             se
nder
[  5]   0.00-10.01  sec  0.00 Bytes  0.00 bits/sec                  rece
iver
iperf3: interrupt - the client has terminated
```

```
[  5]   8.00-9.00   sec   113 MBytes   949 Mbits/sec
[  5]   9.00-10.00  sec  93.1 MBytes   781 Mbits/sec
[  5]  10.00-10.04  sec  4.19 MBytes   834 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.04  sec   947 MBytes   791 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 192.168.3.1, port 58912
[  5] local 192.168.3.3 port 5201 connected to 192.168.3.1 port 58914
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec   136 MBytes  1.14 Gbits/sec
[  5]   1.00-2.00   sec   139 MBytes  1.17 Gbits/sec
[  5]   2.00-3.00   sec   136 MBytes  1.14 Gbits/sec
[  5]   3.00-4.00   sec   137 MBytes  1.15 Gbits/sec
[  5]   4.00-5.00   sec   136 MBytes  1.14 Gbits/sec
[  5]   5.00-6.00   sec   135 MBytes  1.13 Gbits/sec
[  5]   6.00-7.00   sec   136 MBytes  1.14 Gbits/sec
[  5]   7.00-8.00   sec   135 MBytes  1.13 Gbits/sec
[  5]   8.00-9.00   sec   135 MBytes  1.14 Gbits/sec
[  5]   9.00-10.00  sec   135 MBytes  1.13 Gbits/sec
[  5]  10.00-10.04  sec  5.78 MBytes  1.13 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.04  sec  1.33 GBytes  1.14 Gbits/sec                  receiver
[  5]  10.00-10.04  sec  5.78 MBytes  1.13 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.04  sec  1.33 GBytes  1.14 Gbits/sec                  receiver
iperf3: the client has terminated
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 86 : Iperf3 with Meter 5 (5 Gbps)

4.10 Installation and Configuration of ONOS in docker

In this step another SDN controller named ONOS (Open Network Operating System) will be installed inside a container. The same flows and meters will be tested with this controller. This can later be used to differentiate between the working of two controllers and which might be optimal and user friendly to be utilized for the future projects. The new container can also be observed on Figure 87.

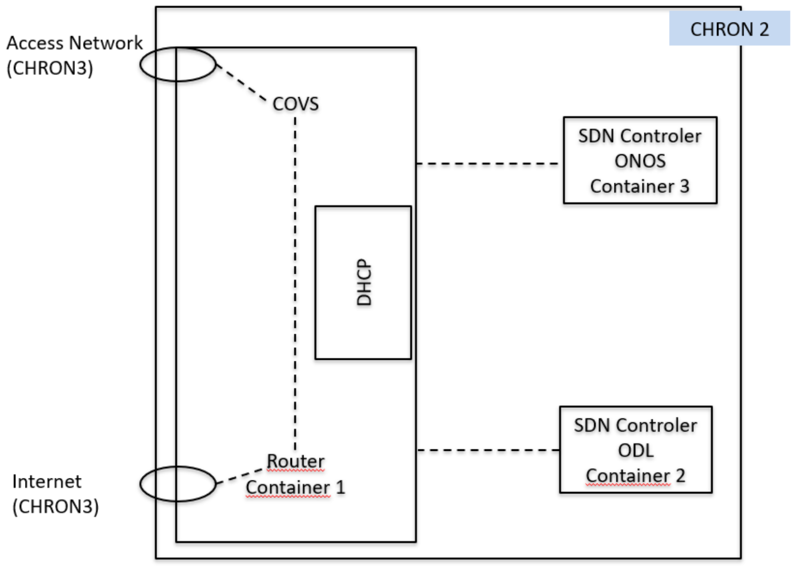


Figure 87 :Docker containers with ONOS.

Unlike ODL, installation of ONOS inside a docker container is simpler and faster. The official ONOS developers provide good documentation on the Docker hub to make it easier for any user that wants to run ONOs inside a container. There are two ways of doing this process. We can either clone the pre-built ONOS image by the official ONOS on Docker hub or build our own ONOS image with the information provided by the ONOS community [21]:

- *First Option*: It can be done with the command *docker pull onosproject/onos.* This command pulls/clones the latest ONOS image and it can be run as any other image we did earlier to start a container that has ONOS installed into it.
- *Second Option*: This step can take very long depending on the size of our image and it is recommended to use the first option if it is available. We build our own image with the information provided by ONOSPROJECT on Docker hub. To build an image we need to create a DOCKERFILE and install every packets and software that is needed. This method was used during the OVS and the ODL installations. Refer to Chapter 2 on how to build and run an image on Docker [21].

We used the first method to clone the image provided by the ONOS community, the image can be run to start a ONOS container with the command:

docker run -t -i -d --cap-add SYS_ADMIN --network <docker-bridge-network> --name=<container-name> -p 8101:8101 -p 5005:5005 -p 830:830 -p 10.0.103.73:8181:8181 --privileged <image-name:tag>

The above command starts a container from of the ONOS image. The IP address 10.0.103.73 is the IP of the host interface, the docker container will use it to surf to the onos GUI. This works the same as we did with the ODL configurations, i.e. opening the host port with IP 10.0.103.73 to be able to configure the controller inside a container. The ONOS is running now and the next step is to surf to its GUI and connect it to our OVS. To access the ONOS GUI, we surf to the following link on any browser (google chrome, Firefox etc.):

http://10.0.103.73:8181/onos/ui/#/

Upon going to the link, we need to provide the default authentication to access the onos user interface with username: *onos* and password: *rocks*, as it can be observed in the figure 88

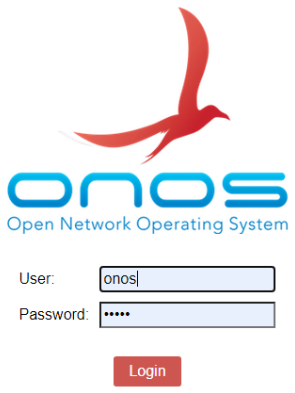


Figure 88 : ONOS authentication

After the authentication succeeds, the following screen shown in Figure 89 appears on the browser.

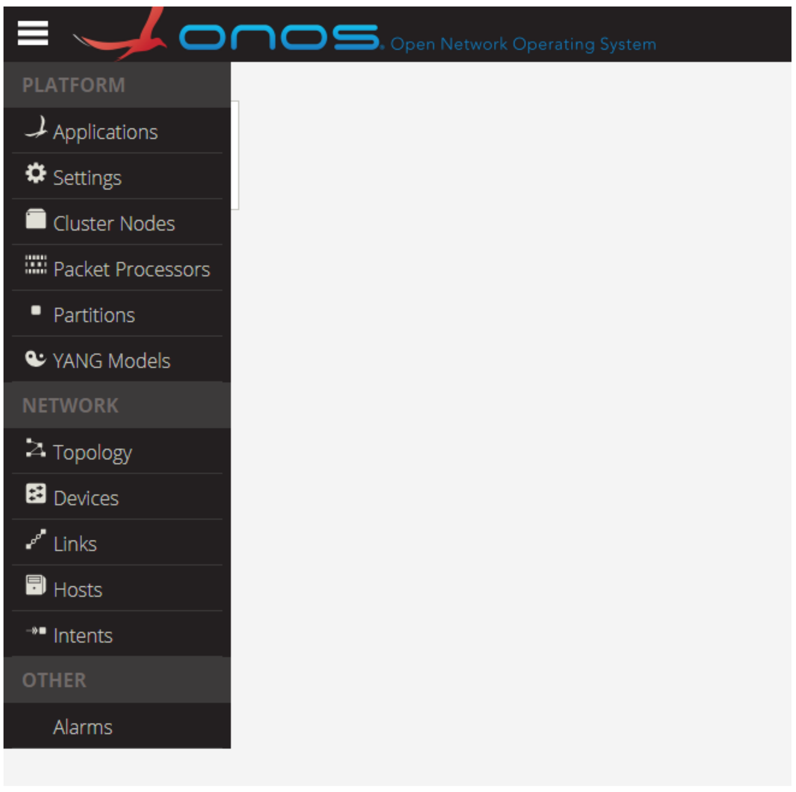
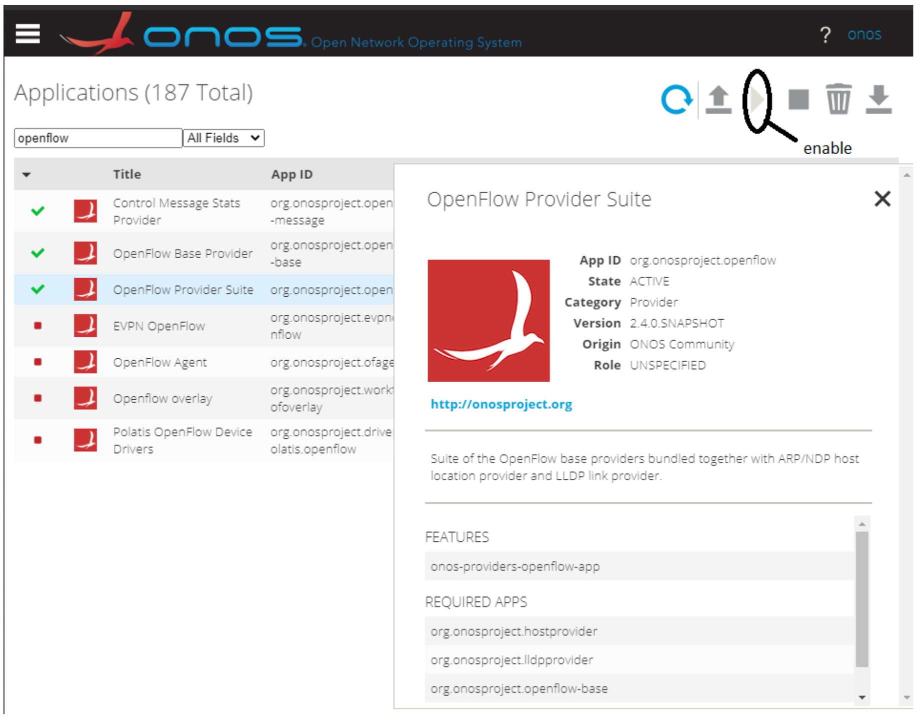


Figure 89 : ONOS frontpage

The onos GUI has multiple options available, the topology shows the connected switches, the devices show all connected devices and the applications tab helps us to activate applications that will be necessary to create flow entries. The next step is to connect the OVS to the ONOS, and it can be done with two different steps:

1. Step 1: On the OVS command line we set up the IP of the ONOS container as its master. This is done with the command *ovs-vsctl set-controller mybridge tcp:10.1.4.0:6633*. This is like the step that was done during the ODL connection to the switch.
2. Step 2: We navigate to the *Application*s option that can be seen on Figure 89 (on the left) and enable OpenFlow application. To enable the OpenFlow application, we search OpenFlow in the list of all available and installed application as it can be observed in Figure 90.



Applications (187 Total)

openflow | All Fields

		Title	App ID
✓		Control Message Stats Provider	org.onosproject.open -message
✓		OpenFlow Base Provider	org.onosproject.open -base
✓		OpenFlow Provider Suite	org.onosproject.open
■		EVPN OpenFlow	org.onosproject.evpn nflow
■		OpenFlow Agent	org.onosproject.ofage
■		Openflow overlay	org.onosproject.work ofoverlay
■		Polatis OpenFlow Device Drivers	org.onosproject.drive olatis.openflow

OpenFlow Provider Suite

App ID	org.onosproject.openflow
State	ACTIVE
Category	Provider
Version	2.4.0.SNAPSHOT
Origin	ONOS Community
Role	UNSPECIFIED

http://onosproject.org

Suite of the OpenFlow base providers bundled together with ARP/NDP host location provider and LLDP link provider.

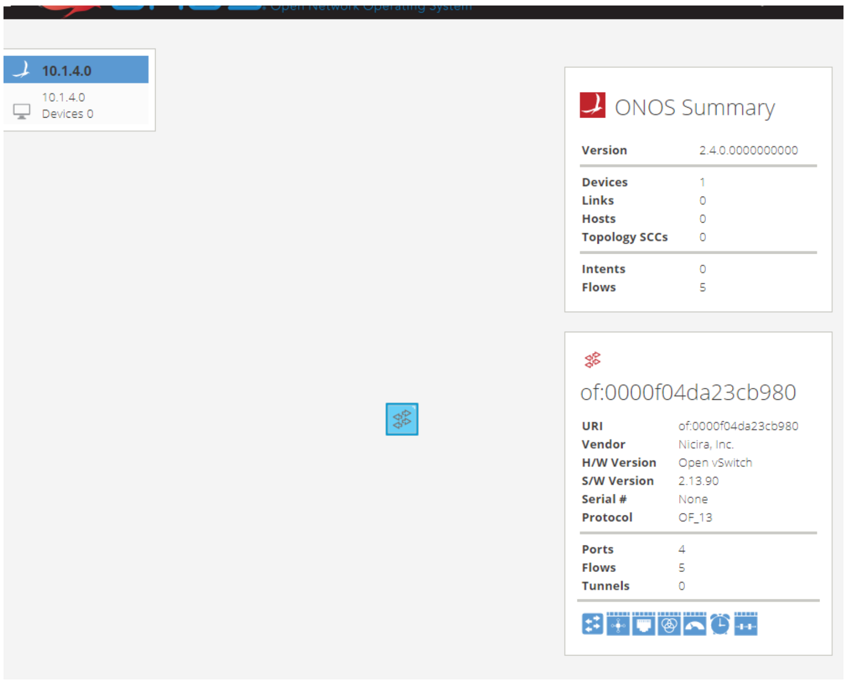
FEATURES
onos-providers-openflow-app

REQUIRED APPS
org.onosproject.hostprovider
org.onosproject.lldpprovider
org.onosproject.openflow-base

Figure 90 : Enabling an application.

In Figure 90 the green tick means that the application has been enabled, and the red box means it has not been enabled yet. On the same figure we can also see an enable button on the top right, so selecting the application we need and clicking on the enable button the selected OpenFlow application is enabled, which will automatically connect the onos controller and the open vSwitch. To check the connection, we click on the *topology* functionality that was seen earlier in Figure 89.

On the other hand, in Figure 91, the switch with a specific switch-id has been connected to the controller and it has few ports connected to it. In this case the ports are *enp6s0f1.833*, *enp6s0f1.806*, *eno2* and the internal port of the ovs-bridge. The switch used OpenFlow protocol 13 and has few flows enabled on it.



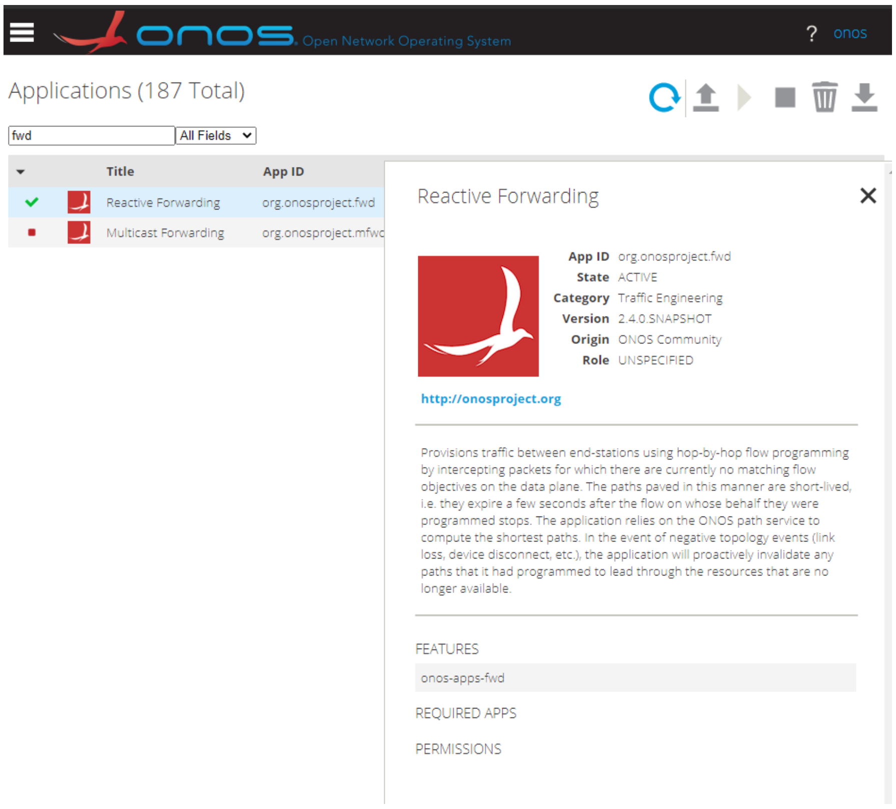
ONOS Summary

Version	2.4.0.0000000000
Devices	1
Links	0
Hosts	0
Topology SCCs	0
Intents	0
Flows	5

of:0000f04da23cb980

URI	of:0000f04da23cb980
Vendor	Nicira, Inc.
H/W Version	Open vSwitch
S/W Version	2.13.90
Serial #	None
Protocol	OF_13
Ports	4
Flows	5
Tunnels	0

Figure 91 : Topology after the connection has been made.



		Title	App ID
✓		Reactive Forwarding	org.onosproject.fwd
■		Multicast Forwarding	org.onosproject.mfwd

Reactive Forwarding

App ID	org.onosproject.fwd
State	ACTIVE
Category	Traffic Engineering
Version	2.4.0.SNAPSHOT
Origin	ONOS Community
Role	UNSPECIFIED

http://onosproject.org

Provisions traffic between end-stations using hop-by-hop flow programming by intercepting packets for which there are currently no matching flow objectives on the data plane. The paths paved in this manner are short-lived, i.e. they expire a few seconds after the flow on whose behalf they were programmed stops. The application relies on the ONOS path service to compute the shortest paths. In the event of negative topology events (link loss, device disconnect, etc.), the application will proactively invalidate any paths that it had programmed to lead through the resources that are no longer available.

FEATURES

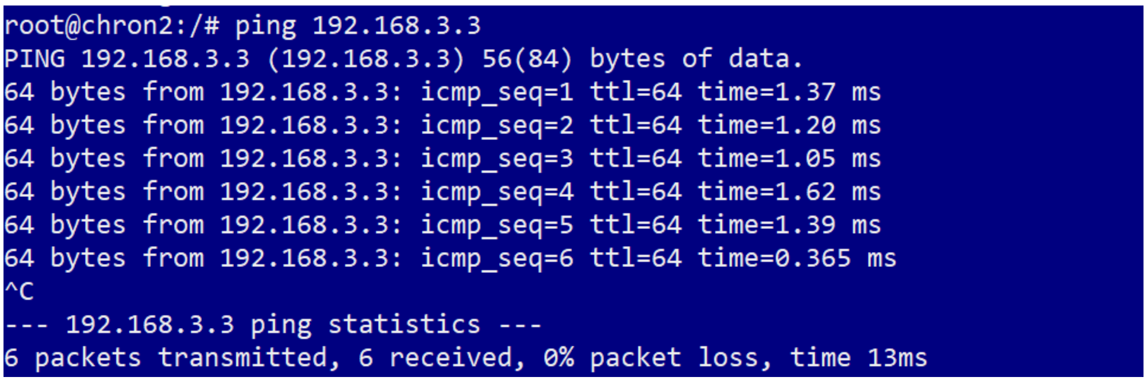
onos-apps-fwd

REQUIRED APPS

PERMISSIONS

Figure 92 : Application fwd has been installed

Once the installation and connection has been enabled, we need to enable one more application on the ONOS that will allow the connections between the hosts through the bridge. With ODL this connection was done by creating a normal flow and on the ONOS we can enable the application named FWD on the applications list as it can be seen in Figure 92. After the application has been installed, the connection can be tested by pinging to the other side of the interface connected to the OVS bridge (enp6s0f1.833 on Chron 3). The connection test is shown in Figure 93.



```
root@chron2:/# ping 192.168.3.3
PING 192.168.3.3 (192.168.3.3) 56(84) bytes of data.
64 bytes from 192.168.3.3: icmp_seq=1 ttl=64 time=1.37 ms
64 bytes from 192.168.3.3: icmp_seq=2 ttl=64 time=1.20 ms
64 bytes from 192.168.3.3: icmp_seq=3 ttl=64 time=1.05 ms
64 bytes from 192.168.3.3: icmp_seq=4 ttl=64 time=1.62 ms
64 bytes from 192.168.3.3: icmp_seq=5 ttl=64 time=1.39 ms
64 bytes from 192.168.3.3: icmp_seq=6 ttl=64 time=0.365 ms
^C
--- 192.168.3.3 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 13ms
```

Figure 93 : Connection test through the ovs bridge with FWD application enabled.

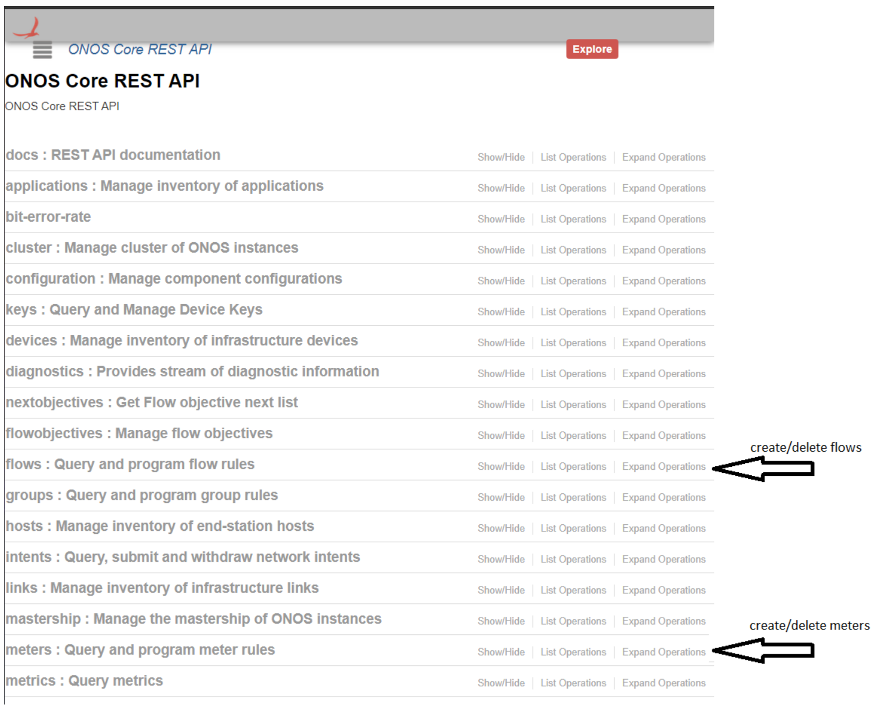
After the normal connection has been configured on the bridge the next step is to create flow entries and test the meters.

4.11 Configuration of flows and meters in ONOS

Unlike ODL, with ONOS it is possible to create flows via the graphical user interface. Both use the REST API so most of the parameters during the creation and configuration of flow entries remains the same.

The flow entries will be created using the GUI that is available on the APIDOCS page. The page to create and view the flows on the onos can be opened with the link http://10.0.103.73:8181/onos/v1/docs/#/ on any browser such as Google Chrome or Firefox.

The same authentication is used here (username: *onos* password: *rocks*). After the authentication succeeds, a list of available options appears on the page as it can be seen on Figure 94. Out of all the available features, only two features with the arrows pointed to them on Figure 94 will be used. They are used to configure flows and meters.



ONOS Core REST API

ONOS Core REST API

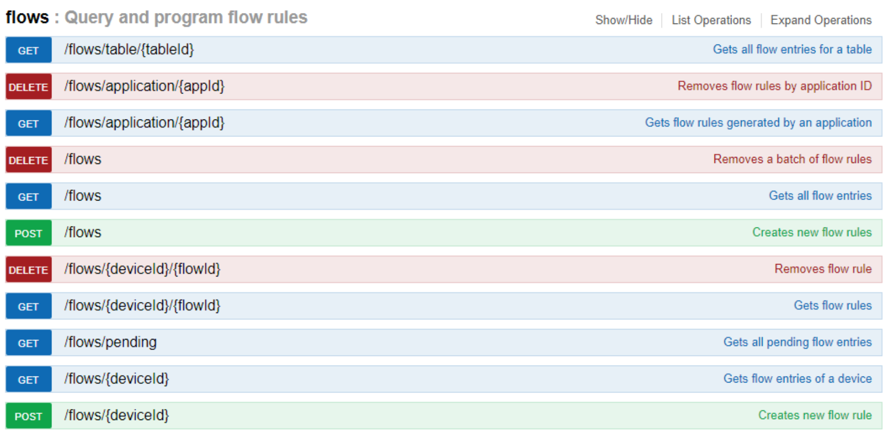
docs : REST API documentation	Show/Hide | List Operations | Expand Operations
applications : Manage inventory of applications	Show/Hide | List Operations | Expand Operations
bit-error-rate	Show/Hide | List Operations | Expand Operations
cluster : Manage cluster of ONOS instances	Show/Hide | List Operations | Expand Operations
configuration : Manage component configurations	Show/Hide | List Operations | Expand Operations
keys : Query and Manage Device Keys	Show/Hide | List Operations | Expand Operations
devices : Manage inventory of infrastructure devices	Show/Hide | List Operations | Expand Operations
diagnostics : Provides stream of diagnostic information	Show/Hide | List Operations | Expand Operations
nextobjectives : Get Flow objective next list	Show/Hide | List Operations | Expand Operations
flowobjectives : Manage flow objectives	Show/Hide | List Operations | Expand Operations
flows : Query and program flow rules	Show/Hide | List Operations | Expand Operations
groups : Query and program group rules	Show/Hide | List Operations | Expand Operations
hosts : Manage inventory of end-station hosts	Show/Hide | List Operations | Expand Operations
intents : Query, submit and withdraw network intents	Show/Hide | List Operations | Expand Operations
links : Manage inventory of infrastructure links	Show/Hide | List Operations | Expand Operations
mastership : Manage the mastership of ONOS instances	Show/Hide | List Operations | Expand Operations
meters : Query and program meter rules	Show/Hide | List Operations | Expand Operations
metrics : Query metrics	Show/Hide | List Operations | Expand Operations

Figure 94 : ONOS Core REST API list of available features.

In the next steps we will use the above features to configure flows and meters with different parameters.

4.11.1 Configuration of flows for specific Network Address

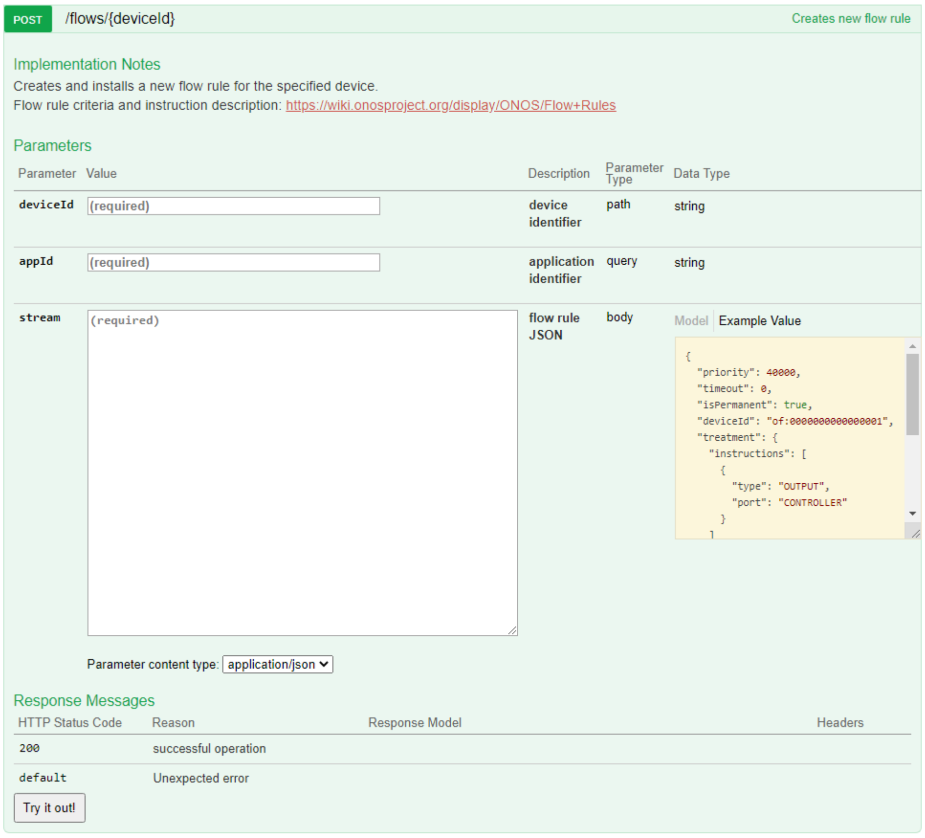
Out of the available features on the Rest API Documents we navigate to the flows by clicking on the it in the Figure 94. Then, we have several options as it can be observed in Figure 95.


flows : Query and program flow rules — Show/Hide | List Operations | Expand Operations

GET	/flows/table/{tableId}	Gets all flow entries for a table
DELETE	/flows/application/{appId}	Removes flow rules by application ID
GET	/flows/application/{appId}	Gets flow rules generated by an application
DELETE	/flows	Removes a batch of flow rules
GET	/flows	Gets all flow entries
POST	/flows	Creates new flow rules
DELETE	/flows/{deviceId}/{flowId}	Removes flow rule
GET	/flows/{deviceId}/{flowId}	Gets flow rules
GET	/flows/pending	Gets all pending flow entries
GET	/flows/{deviceId}	Gets flow entries of a device
POST	/flows/{deviceId}	Creates new flow rule

Figure 95 : All options available with the feature FLOWS

The options speak for themselves. GET lets the user get the existing flow or table data, DELETE lets the user delete them and POST lets the user create a new flow entry. We select the POST option to start creating a new flow entry, after clicking on the post the following option appears as seen in Figure 96.



POST /flows/{deviceId} Creates new flow rule

Implementation Notes

Creates and installs a new flow rule for the specified device.

Flow rule criteria and instruction description: https://wiki.onosproject.org/display/ONOS/Flow+Rules

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceId	(required)	device identifier	path	string
appId	(required)	application identifier	query	string
stream	(required)	flow rule JSON	body	Model Example Value

```
{
  "priority": 40000,
  "timeout": 0,
  "isPermanent": true,
  "deviceId": "of:0000000000000001",
  "treatment": {
    "instructions": [
      {
        "type": "OUTPUT",
        "port": "CONTROLLER"
      }
    ]
```

Parameter content type: application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	successful operation		
default	Unexpected error		

Figure 96 : Creation of flows with GUI

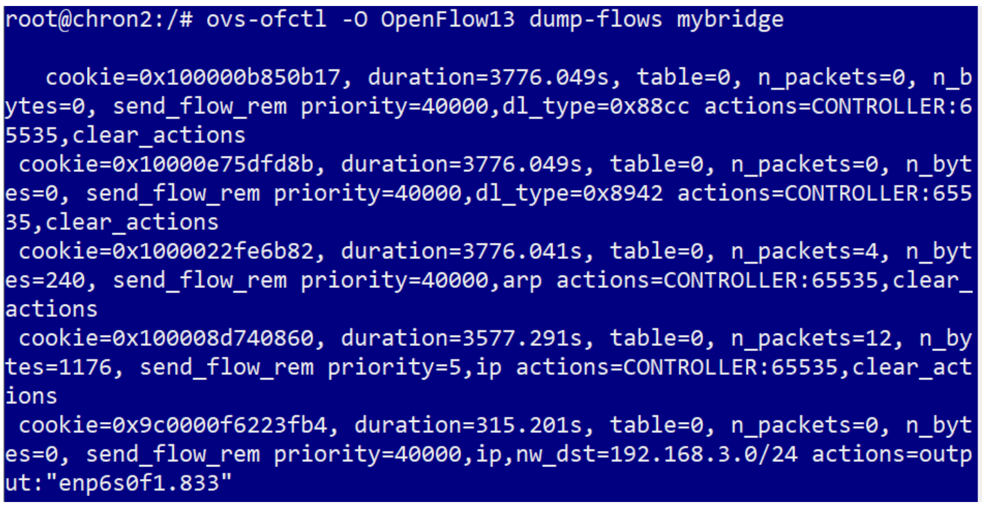
The user gets three input fields. The data for this input field are self-explanatory. The *device id* is the id of the switch that can be found on the topology table as it was shown earlier in Figure 91 (of:0000f04da23cb980). The *appID* is the name of the application that was enabled in Figure 92 (org.onosproject.fwd**)**. Finally, the *stream* field will contain the rules and instruction for the flow the user wants to create.

As it can also be seen on the right side of Figure 96, the REST API also provides with an example flow that can be used to create a new flow. We use that example flow as a template for a new flow. Then, to create a flow that is used for the packet's destination for IP's in the Network Address 192.168.3.0/24 the following stream is used:

```
{ "flows": [ {
    "priority": 50000,
    "timeout": 0,
    "isPermanent": true,
    "deviceId": "of:0000f04da23cb980",
"treatment": {
  "instructions": [ {
     "type": "OUTPUT",
     "port": "1" }]},
"selector": {
  "criteria": [  {
     "type": "ETH_TYPE",
     "ethType": "2048"},{
  "type": "IPV4_DST",
  "ip": "192.168.3.0/24"
}  ]  }   } ]}
```

With the three input fields being filled, the flow can be uploaded with the button *Try it out!* as seen in Figure 96. The flow entry is very similar to the one that was created with the ODL. For the ONOS controller, the json format is being used because the example flow that the REST API provides is also on the json format. The flow has 40000 priority and will forward all data destined for 192.168.3.0 through its port 1(enp6s0f1.833 is saved as Port 1 in the switch database, this was also mentioned during the flow configurations with ODL). This flow is very similar to the flow we created with ODL, only the formats are different, xml format with ODL and json format with ONOS.

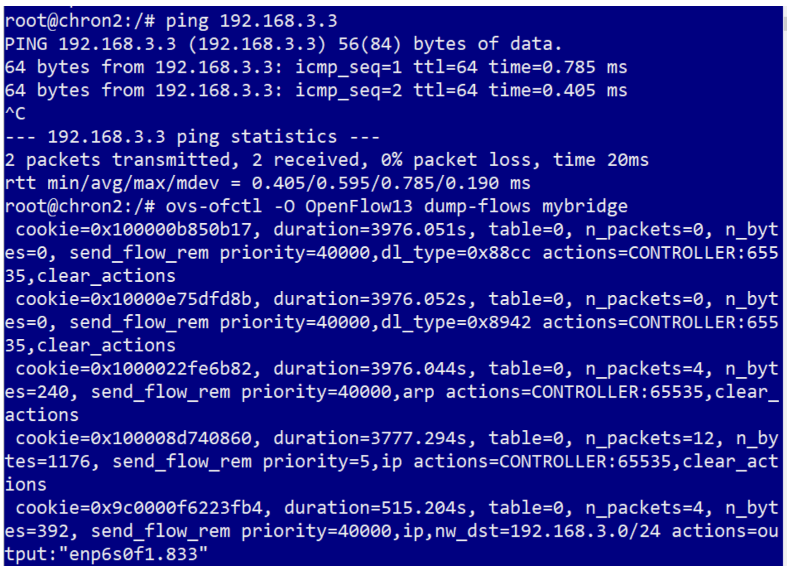
To check if the OVS has accepted the flow, similar tests can be done as it was done during the flow creations with ODL. With the command *ovs-ofctl -O OpenFlow13 dump-flows mybridge,* the switch is asked to output all its flows as seen in Figure 97.



```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge

   cookie=0x100000b850b17, duration=3776.049s, table=0, n_packets=0, n_b
ytes=0, send_flow_rem priority=40000,dl_type=0x88cc actions=CONTROLLER:6
5535,clear_actions
 cookie=0x10000e75dfd8b, duration=3776.049s, table=0, n_packets=0, n_byt
es=0, send_flow_rem priority=40000,dl_type=0x8942 actions=CONTROLLER:655
35,clear_actions
 cookie=0x1000022fe6b82, duration=3776.041s, table=0, n_packets=4, n_byt
es=240, send_flow_rem priority=40000,arp actions=CONTROLLER:65535,clear_
actions
 cookie=0x100008d740860, duration=3577.291s, table=0, n_packets=12, n_by
tes=1176, send_flow_rem priority=5,ip actions=CONTROLLER:65535,clear_act
ions
 cookie=0x9c0000f6223fb4, duration=315.201s, table=0, n_packets=0, n_byt
es=0, send_flow_rem priority=40000,ip,nw_dst=192.168.3.0/24 actions=outp
ut:"enp6s0f1.833"
```

Figure 97 : Dump-flows on OVS

In this figure, the recently created flow for the network address 192.168.3.0/24 (the last flow in Figure 97), along with four default flows created by the ONOS are available. Three of the default flows are created during the initial connection between the OVS and ONOS. They are necessary for the controller to work. The fourth flow is the flow that is enabled when we installed the fwd application earlier. The default flows are hidden on the ODL but ONOS lets the user know by just displaying them along with other flows even though the users cannot change or delete them. To test if the flow works as it is expected we ping on the network 192.168.3.0/24. On the last flow of Figure 98 the flow entry is being hit when the user pings through it, as the *n_packets* changes from 0 to 4, so the flow works.



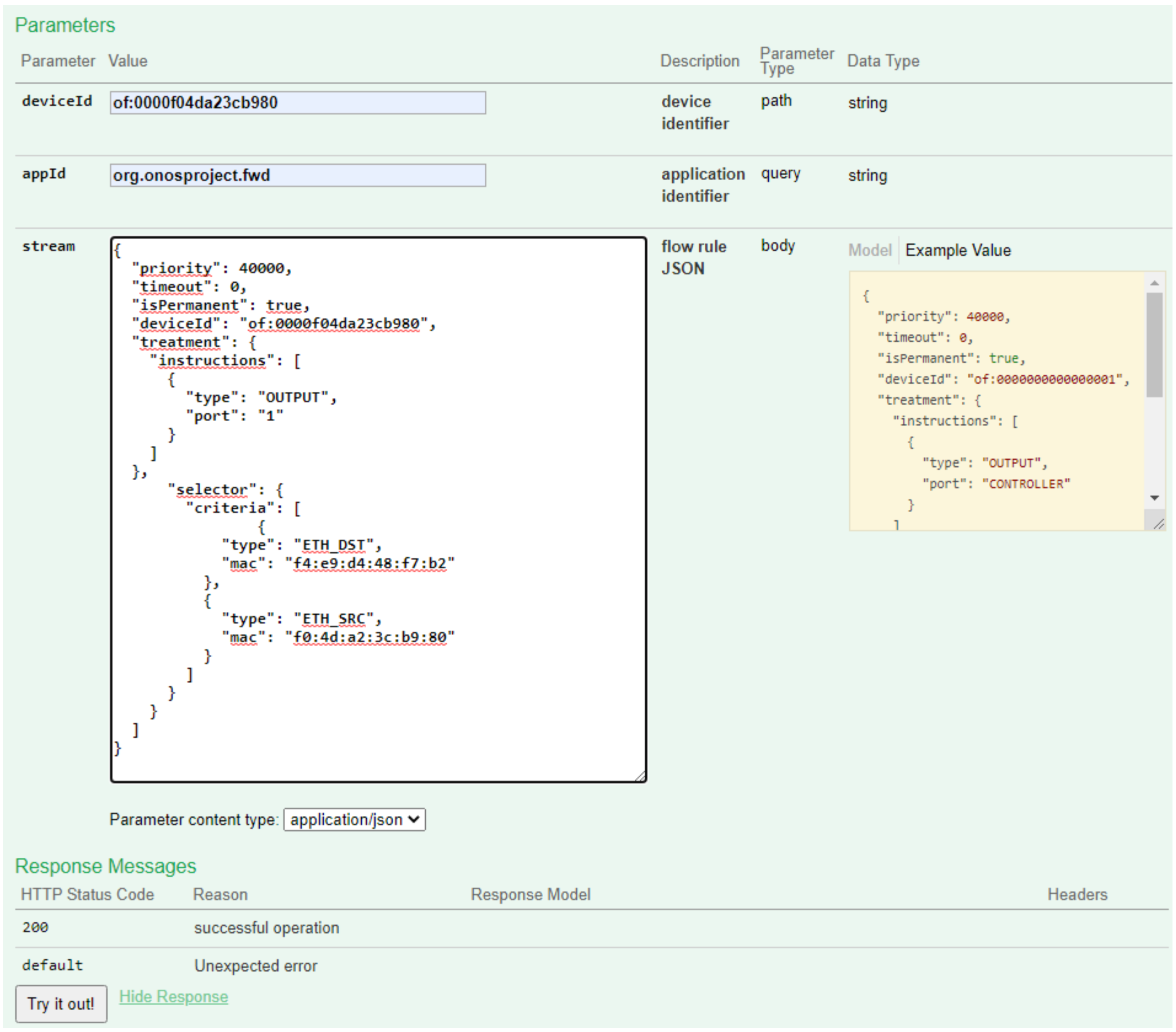
```
root@chron2:/# ping 192.168.3.3
PING 192.168.3.3 (192.168.3.3) 56(84) bytes of data.
64 bytes from 192.168.3.3: icmp_seq=1 ttl=64 time=0.785 ms
64 bytes from 192.168.3.3: icmp_seq=2 ttl=64 time=0.405 ms
^C
--- 192.168.3.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 20ms
rtt min/avg/max/mdev = 0.405/0.595/0.785/0.190 ms
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x100000b850b17, duration=3976.051s, table=0, n_packets=0, n_byt
es=0, send_flow_rem priority=40000,dl_type=0x88cc actions=CONTROLLER:655
35,clear_actions
 cookie=0x10000e75dfd8b, duration=3976.052s, table=0, n_packets=0, n_byt
es=0, send_flow_rem priority=40000,dl_type=0x8942 actions=CONTROLLER:655
35,clear_actions
 cookie=0x1000022fe6b82, duration=3976.044s, table=0, n_packets=4, n_byt
es=240, send_flow_rem priority=40000,arp actions=CONTROLLER:65535,clear_
actions
 cookie=0x100008d740860, duration=3777.294s, table=0, n_packets=12, n_by
tes=1176, send_flow_rem priority=5,ip actions=CONTROLLER:65535,clear_act
ions
 cookie=0x9c0000f6223fb4, duration=515.204s, table=0, n_packets=4, n_byt
es=392, send_flow_rem priority=40000,ip,nw_dst=192.168.3.0/24 actions=ou
tput:"enp6s0f1.833"
```

Figure 98 : Connection test on the newly created flow.

4.11.2 ***Configuration of flows for specific Mac Address***

The steps are the same as above, only a few details needs to be changed in the stream input field, as it is observed in Figure 99.

As seen in this figure 99 the rest of the code remains the same for the mac address and the source and destination is also inserted according to our device mac address. The parameters of the flow, the connection and its working tests are very similar to the tests done with ODL.



Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceId	of:0000f04da23cb980	device identifier	path	string
appId	org.onosproject.fwd	application identifier	query	string
stream	(see below)	flow rule JSON	body	Model / Example Value

```
{
  "priority": 40000,
  "timeout": 0,
  "isPermanent": true,
  "deviceId": "of:0000f04da23cb980",
  "treatment": {
    "instructions": [
      {
        "type": "OUTPUT",
        "port": "1"
      }
    ]
  },
      "selector": {
        "criteria": [
            {
          "type": "ETH_DST",
          "mac": "f4:e9:d4:48:f7:b2"
        },
        {
          "type": "ETH_SRC",
          "mac": "f0:4d:a2:3c:b9:80"
        }
      ]
    }
  }
]
}
```

Example Value:

```
{
  "priority": 40000,
  "timeout": 0,
  "isPermanent": true,
  "deviceId": "of:0000000000000001",
  "treatment": {
    "instructions": [
      {
        "type": "OUTPUT",
        "port": "CONTROLLER"
      }
    ]
```

Parameter content type: application/json

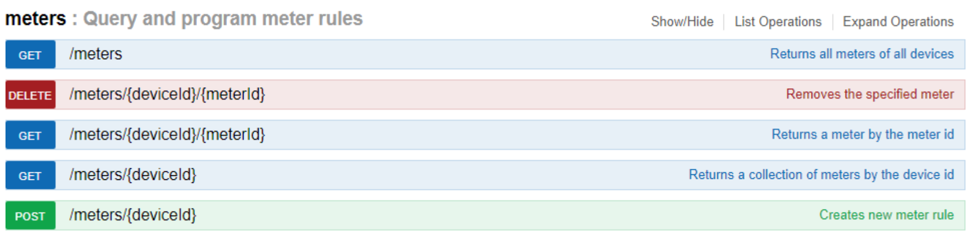
Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	successful operation		
default	Unexpected error		

Figure 99 : Flow entry for specific MAC id

4.11.3 Configurations of meters

To create a meter via the REST API of the ONOS GUI, we navigate by clicking on it, to the meters feature on the earlier Figure 94. The options available with the feature meters in this figure 100 are self-explanatory. GET lets the user view the existing meter, DELETE lets the user delete it and POST lets the user create a new meter.

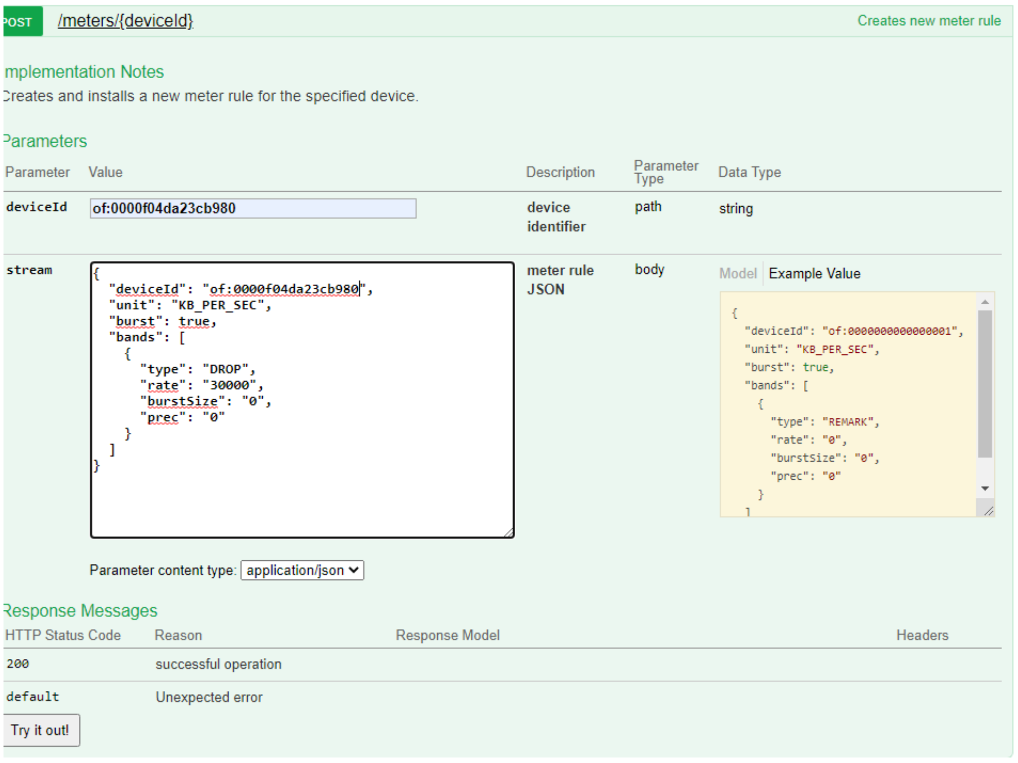


meters : Query and program meter rules

GET	/meters	Returns all meters of all devices
DELETE	/meters/{deviceId}/{meterId}	Removes the specified meter
GET	/meters/{deviceId}/{meterId}	Returns a meter by the meter id
GET	/meters/{deviceId}	Returns a collection of meters by the device id
POST	/meters/{deviceId}	Creates new meter rule

Figure 100 : Options on the meters feature.

The Post option on Figure 100 asks the user to fill information in two input fields with information. The first field asks the user for the switch id and the second one asks the user for the meter rules and instructions. The REST API docs also provides the user with a meter example as it can be observed on the right part of Figure 101. This example meter can be used as a template for our meters.



POST /meters/{deviceId} Creates new meter rule

mplementation Notes

Creates and installs a new meter rule for the specified device.

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceId	of:0000f04da23cb980	device identifier	path	string
stream	(see below)	meter rule JSON	body	Model | Example Value

```
{
  "deviceId": "of:0000f04da23cb980",
  "unit": "KB_PER_SEC",
  "burst": true,
  "bands": [
    {
      "type": "DROP",
      "rate": "30000",
      "burstSize": "0",
      "prec": "0"
    }
  ]
}
```

Example Value:

```
{
  "deviceId": "of:0000000000000001",
  "unit": "KB_PER_SEC",
  "burst": true,
  "bands": [
    {
      "type": "REMARK",
      "rate": "0",
      "burstSize": "0",
      "prec": "0"
    }
  ]
```

Parameter content type: application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
200	successful operation		
default	Unexpected error		

Figure 101 : Creation of meters in ONOS

A typical meter can be created with the following json stream:

```
{  "deviceId": "of:0000f04da23cb980",
  "unit": "KB_PER_SEC",
  "burst": true,
  "bands": [{
     "type": "DROP",
     "rate": "30000",
     "burstSize": "0",
     "prec": "0"  }  ]}
```

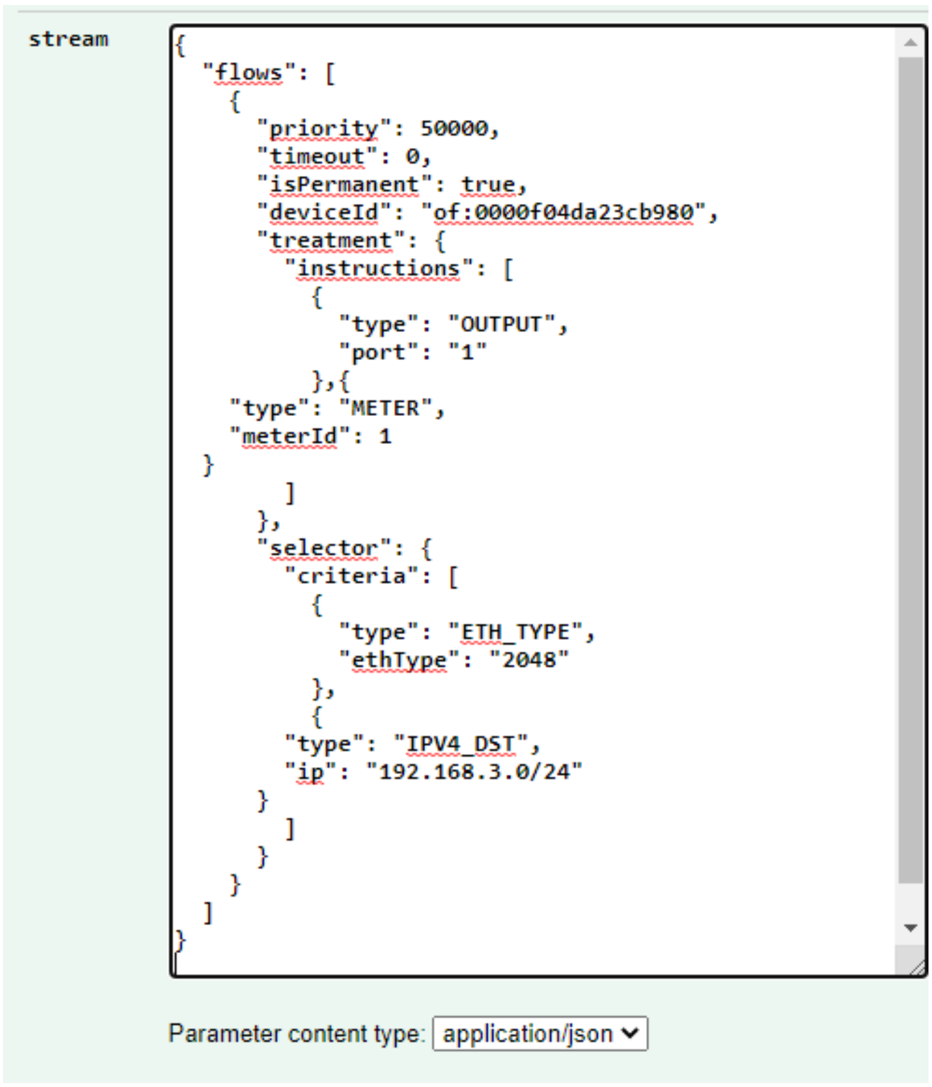
The above json stream creates a meter that will only allow a throughput of 30 Mbps on the flow entry where this meter has been added to. For test purpose we will create multiple meters from 30 Mbps up to 5 Gbps by just replacing the rate in the code. The meter value and tests will be the same as ODL tests. To check if the meters have been accepted by the switch the same steps can be followed with the command *ovs-ofctl -O OpenFlow13 dump-meters mybridge.*

4.11.4 *Configuration of flows with meters*

Now the flows and meters have been created, the meters can be added to a flow by adding 2 extra line of code on the flow's json stream, in the next way:

```
{
    "type": "METER",
    "meterId": 1
 }
```

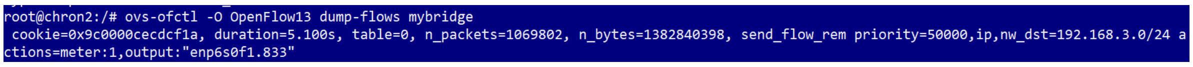
The, a flow-entry with a meter configured into it looks as Figure 102. This figure shows the json-stream used to configure the flow with the meter added into it and Figure 103 shows that the flow has been configured with meter 1. During the test phase, the same flow will be used multiple times while only changing the meter ID from 1 to 15 in order to test the meters' performance with ONOS and comparing it with ODL.



```
stream
{
  "flows": [
    {
      "priority": 50000,
      "timeout": 0,
      "isPermanent": true,
      "deviceId": "of:0000f04da23cb980",
      "treatment": {
        "instructions": [
          {
            "type": "OUTPUT",
            "port": "1"
          },{
    "type": "METER",
    "meterId": 1
  }
        ]
      },
      "selector": {
        "criteria": [
          {
            "type": "ETH_TYPE",
            "ethType": "2048"
          },
          {
        "type": "IPV4_DST",
        "ip": "192.168.3.0/24"
      }
        ]
      }
    }
  ]
}
```

Parameter content type: application/json

Figure 102 : Configuration of a flow with meter into it

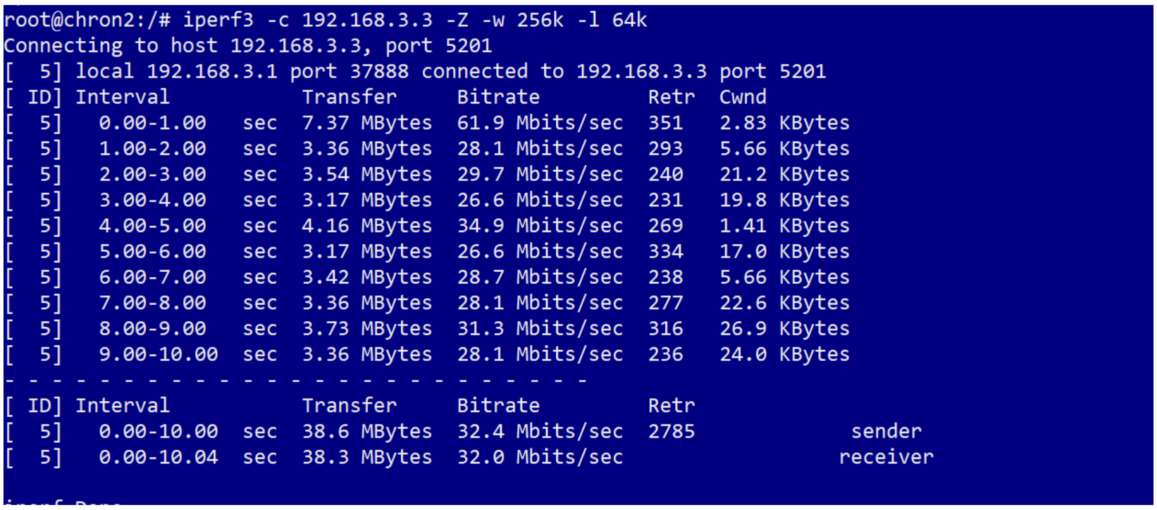


```
root@chron2:/# ovs-ofctl -O OpenFlow13 dump-flows mybridge
 cookie=0x9c0000cecdcf1a, duration=5.100s, table=0, n_packets=1069802, n_bytes=1382840398, send_flow_rem priority=50000,ip,nw_dst=192.168.3.0/24 a
ctions=meter:1,output:"enp6s0f1.833"
```

Figure 103 : Check the flow entry on the ovs

4.12 Tests with flows and meters in ONOS

The test process of the meters and flows are the same as they were with ODL and to avoid repeating the same figures and commands in the report only one example will be shown here.



```
root@chron2:/# iperf3 -c 192.168.3.3 -Z -w 256k -l 64k
Connecting to host 192.168.3.3, port 5201
[  5] local 192.168.3.1 port 37888 connected to 192.168.3.3 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  7.37 MBytes  61.9 Mbits/sec  351   2.83 KBytes
[  5]   1.00-2.00   sec  3.36 MBytes  28.1 Mbits/sec  293   5.66 KBytes
[  5]   2.00-3.00   sec  3.54 MBytes  29.7 Mbits/sec  240   21.2 KBytes
[  5]   3.00-4.00   sec  3.17 MBytes  26.6 Mbits/sec  231   19.8 KBytes
[  5]   4.00-5.00   sec  4.16 MBytes  34.9 Mbits/sec  269   1.41 KBytes
[  5]   5.00-6.00   sec  3.17 MBytes  26.6 Mbits/sec  334   17.0 KBytes
[  5]   6.00-7.00   sec  3.42 MBytes  28.7 Mbits/sec  238   5.66 KBytes
[  5]   7.00-8.00   sec  3.36 MBytes  28.1 Mbits/sec  277   22.6 KBytes
[  5]   8.00-9.00   sec  3.73 MBytes  31.3 Mbits/sec  316   26.9 KBytes
[  5]   9.00-10.00  sec  3.36 MBytes  28.1 Mbits/sec  236   24.0 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  38.6 MBytes  32.4 Mbits/sec  2785             sender
[  5]   0.00-10.04  sec  38.3 MBytes  32.0 Mbits/sec                  receiver
```

Figure 104 : Iperf test with meter 1 enabled

As it was the case with ODL earlier, the meter 1 has been enabled with drop rate of 30Mbps. Figure 104 shows the meter is working as intended on the flow.

Finally, the following table summarized the results of different meters with different drop rates and enabled on the same flow entry.

METER (Mbps)	IPERF RESULTS (Mbps)
30	32 Mbps
50	54 Mbps
100	119 Mbps
150	215 Mbps
200	342 Mbps
250	452 Mbps
300	553 Mbps

350	666 Mbps
400	789 Mbps
450	851 Mbps
500	954 Mbps
550	1.09 Gbps
600	1.07 Gbps
800	1.15 Gbps
1000	1.10 Gbps
20000	1.10 Gbps

The results are the same as ODL tests, the low bitrate meters are accurate while the bitrate gets bad as we go higher, but the bitrate starts to get inaccurate after the 150Mbps. Finally, the maximum throughput of the switch is still capped at 1.10 Gbps as it was the case with the tests done from ODL controller.

From the tests done with ODL, ONOS and without any controller, the fault seems to lie on the OVS. The OVS does not meet our bandwidth expectation of 10 Gbps. The OVS is being run in the user space. Further research needs to be done regarding installing the kernel modules of OVS inside the docker container to test if the problem persists.

4.13 Conclusions

In this chapter, we have described the implementation and configuration of a router and a DHCP server with Linux network utilities, solving the different difficulties of this installation for our use case. Subsequently, the virtual switch was installed on the container. Below, we have described the connection of the switch with the ODL controller following a special routine to avoid the instability of the Open vSwitch utility, being able to add, modify or remove flows or meters on each switch. Finally, we have described, configured and tested the Flow entries with different scenarios. The same tests were repeated with a different sdn -controller named ONOS. The ONOS controller was more user friendly than the ODL as the onos developers have updated version of onos image on their Docker hub and lets the user configure the flows and meters via the GUI, that is, a graphical interface.

5

Conclusions and Future lines

5.1 Conclusions

In this project we had to adjust our methodology, and objectives because of the circumstances created by the Pandemic. We managed to make vital changes in time so we could still keep working on the project. The research done will still be useful when the labs are accessible, and the real configurations can be implemented and tested.

To achieve the completion, the structure of the GPON access network of the L2007 laboratory of the Higher Technical School of Telecommunications Engineers of Valladolid has first been analyzed. Subsequently, the different crucial components have been implemented to convert a conventional GPON access network into an SDN network, implementing a router to be able to control the network through the routing skills of a Linux system and using other utilities such as iptables (famous firewall and in general, packet interceptor) and vconfig (to control VLANs). In addition, we installed virtual switch in order to emulate SDN functionalities in the OLT. Finally, OpenFlow communication was implemented between the virtual switch and a central controller (using OpenDayLight and ONOS) located inside a docker container.

During the development of this work, we had to be aware of the difficulty of the research work and the challenges that must be faced in order to carry out a project to success, along with the difficulty of proposing an open source based project still in development, since many of the functions do not work exactly as detailed in the documentation or are not yet implemented.

5.2 Future lines

The biggest part of the project was completed as further as the docker Technology allowed. The lack of time still left us with many holes to be filled that could be carried onto the next project.

First, starting from the top, after thorough research, we found out that Docker utilizes application level virtualization, and it is used to virtualize an application or a software, whereas VM's use hardware level virtualization. However, it gets complex when we want to virtualize the functions like switching and routing. For these functions to work properly we have to force the containers into host networking mode, which simply uses the network configurations of the host machine, so they have access to the Network interfaces of the host. It is as if there is no virtualization happening here. This subject needs to be researched more, if the virtualization of a router and ovs are worth it at all.

Secondly, more research needs to be done regarding DHCP server inside a container. During this project a DHCP server was installed and configured using the host networking mode but according to documents on the website PI-HOLE [19], it is possible to deploy a DHCP server inside a container using Docker-bridge when a dhcp-relay agent is installed inside a router. We did not succeed on this aspect after multiple tries and decided to move on with the project with the host mode networking to save time. When we get the DHCP server to work on a docker-bridge, we could also investigate to create a complete version of the container, that starts its DHCP service automatically on boot. This should be possible if the docker image for dhcp-server is self-built and accompanied with a script that creates the VLANS, configures the IP for the VLANS and starts the service when the command *docker run* is executed. Creating a script inside a container is not possible so the script needs to be created along with the image.

Regarding the OVS, we should research on how to install a kernel module of the OVS inside a docker container. As we know docker containers do not have a kernel of their own and uses the host's kernel. We need to look into building an ovs-image that can utilize the host's Kernel and run in the background smoothly.

For the SDN controllers inside a docker container, it is suggested to stick with the ONOS controller, it is user friendlier than ODL and the open community updates the docker images with the latest version frequently on the Docker Hub. The ODL image on the Docker Hub was updated 3 years ago for the last time and the GUI to configure the flows are not supported by the newer releases anymore. It seems like the project (docker image for ODL) has been abandoned and no one in the open community seems to be interested in taking it further.

Finally, we could look into implementing the configurations as it is, on the real devices in the laboratory with the GPON network. We could start by testing the 10 Gbps output of the GPON and if the SDN technology that we have implementd on the docker containers run smoothly on the real interfaces. If the router works perfectly and the DHCP server is able to give IP address to every connected ONT(residental users) and if the GPON works network works in both 1Gbps and 10 Gbps configurations using the developed SDN implementation the major part of the project goal could be deemed complete and the project could be further developed.

When the testing process succeeds future implementations related to OpenFlow can be carried out, one of them being able to move certain global policies related to Dynamic Bandwidth Allocation and GPON network resources to the central OpenFlow controller. In this sense, DBA algorithms dynamically distribute the available bandwidth in a PON network cycle after cycle based on the real-time needs of each user (connected to an ONU / ONT) and the priority of their contracted services. Therefore, these types of algorithms provide a more realistic, flexible and efficient bandwidth distribution in PON networks.

6 Docker commands Cheat sheet

The basic information regarding docker commands is available on Chapter 2 Section 3.3. This section will however show few of the important Docker commands used during the creation and configurations of all containers. This will list the commands used during this project. For more general commands guide refer to the Chapter 2.

docker pull Debian *docker pull ubuntu*	clones the latest debian/ubuntu image from the Docker hub on the machine.
docker tag <image-name> <new-image-name>	Renames the docker image with the name we want.
docker rmi <image-name>	removes a docker image
docker run -it –name<container-name> <image-name>	Start a docker container from an image, in interactive mode with your own container name
docker kill <container-name>	To stop a running container
docker commit <container-name> <image-name>	Save all made changes inside docker container to a new docker image.

docker network create --subnet 10.1.0.0/16 --gateway 10.1.0.1 --ip-range 10.1.4.0/24 bridge_test	Create your own docker-bridge network named bridge_test with NA 10.1.0.0/16
docker run -t -i -d --cap-add SYS_ADMIN --network host --name=ovs_router_dhcp --privileged <image-name:tag>	Start a docker container from an image in privileged most with host networking and admin permissions.
service isc-dhcp-server start	Once inside the container start the dhcp-server with this command
ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock \ --remote=db:Open_vSwitch,Open_vSwitch,manager_options \ --private-key=db:Open_vSwitch,SSL,private_key \ --certificate=db:Open_vSwitch,SSL,certificate \ --bootstrap-ca-cert=db:Open_vSwitch,SSL,ca_cert \ --pidfile --detach --log-file	To configure ovsdb-server to use database

docker run -t -i -d --cap-add SYS_ADMIN --network bridge_test --name=onos -p 8101:8101 -p 5005:5005 -p 830:830 -p 10.0.103.73:8181:8181 --privileged <image-name:tag>	To run the onos image and start a container using bridge_test network and publish port 8181 so the user can access the controller on web browser. Image name in our case was onos:1.1 but it's a subject for change when the images are updated after enabling new applications into them.
docker run -t -i --cap-add SYS_ADMIN --network bridge_test --name=odl -p 10.0.103.73:8181:8181 --privileged <image-name:tag>	To run the odl image and start a container
cd opendaylight-0.11.2 *./bin/karaf*	Once inside the ODL container navigate to the directory and run the script to start the ODL

For more information regarding docker commands we refer to the cheat sheet on the website dockerlabs. [35]

7 Reference

[1] M. Cooney, „What is SDN?," 2019.

[2] A. Amokrane, „Software defined enterprise passive optical network," in *10th International Conference on Network and Service Management (CSNM)*, Rio de Janeiro, 2014.

[3] Webpage for controller OpenDayLight," [Online]. Available: https://www.opendaylight.org/.

[4] Linux Foundation, „Open vSwitch Documentation," [Online]. Available: http://docs.openvswitch.org/en/latest/. [Geopend 11 Mayo 2018].

[5] D. Foundation, „Get started with Docker," [Online]. Available: www.docker.com.

[6] OpenFlow Standard 1.3," [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[7] O. N. Foundation, „ONOS," 2020.

[8] J. K. S. v. d. M. S. W. Andrei Bondkovskii, „Qualitative Comparison of Open-Source," Trinity College Dublin, 2016.

[9] The Linux Foundation, „Open vSwitch Documentation," 2018. [Online]. Available: http://docs.openvswitch.org/en/latest/. [Geopend 2018].

[10] P. Göransson, Software defined networks: a comprehensive approach, 2014.

[11] Creative Commons BY-SA, „Learning Postman”.

[12] S. Pillai, „IPERF: How to test network Speed,Performance,Bandwidth,” 2013.

[13] The Linux Foundation, „Open vSwitch on Linux, FreeBSD and NetBSD,” 2018. [Online]. Available: http://docs.openvswitch.org/en/latest/intro/install/general/. [Geopend 26 Abril 2018].

[14] S. E. B. A. M. J. P. K. P. Jon Dugan.

[15] Z. Kaleem, „iPerf vs iPerf3,” 2017.

[16] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer en K. Prabhu, „iPerf - The ultimate speed test tool for TCP, UDP and SCTP,” [Online]. Available: https://iperf.fr. [Geopend 23 Mayo 2018].

[17] Docker, „Container-networking,” [Online]. Available: https://docs.docker.com/config/containers/container-networking/. [Geopend May 2020].

[18] Debian Project, „Debian -- Packages,” 19 07 2019. [Online]. Available: https://www.debian.org/distrib/packages. [Geopend 20 02 2020].

[19] Pi-Hole, „Docker DHCP and Network Modes,” 2020.

[20] O. Project, „OpenDaylight Documentation,” 2020.

[21] OnosProject, „onosproject/onos,” DockerHub.

[22] E. Nemeth, G. Snyder, T. R. Hein, B. Whaley en D. Mackin, Unix and Linux system administration handbook, 5th edition red., Boston: Addison-Wesley, 2018.

[23] S. Biradar, „The Ultimate Docker cheatsheet,” [Online]. Available: http://dockerlabs.collabnix.com/docker/cheatsheet/.

[24] Tianon, Paultag, "Dockerhub," Debian, 1 5 2020. [Online]. Available: https://hub.docker.com/_/debian. [Accessed 02 5 2020].

[25] S. Seth en M. A. Venkatesulu, TCP/IP Architecture, Design and Implementation in Linux, Wiley-IEEE Press eBook Chapters, 2008.

[26] „vconfig(8) - Linux man page,” [Online]. Available: https://linux.die.net/man/8/vconfig. [Geopend 7 Mayo 2018].

[27] Internet Systems Consortium, „ISC's open source DHCP software system,” 2003. [Online]. Available: https://www.isc.org/downloads/dhcp/. [Geopend 21 Mayo 2018].

[28] The Linux Foundation, „Installing Open vSwitch,” 2018. [Online]. Available: http://docs.openvswitch.org/en/latest/intro/install/. [Geopend 26 April 2020].

[29] D. C, „Open vSwitch Cheat Sheet”.

[30] PICA8, „PICA8-Documentation,” [Online]. Available: https://docs.pica8.com/pages/viewpage.action?pageId=5112035. [Geopend April 2020].

[31] O. N. Foundation, „OpenFlow Switch Specification,” 2012.

[32] O. Project, „OpenFlow Plugin Project User Guide”.

[33] P. inc.

[34] O. N. Foundation, „OpenFlow Switch Specification,” 2012.

[35] S. Biradar, „The Ultimate Docker Cheat Sheet”.

[36] D. Molloy, Raspberry Pi® a fondo para desarrolladores, Madrid: Marcombo, D.L., 2017.

[37] Z. Hu, „A Comprehensive Security Architecture for SDN,” *18th International Conference on Intelligence in Next Generation Networks.*

[38] S. W. e. a. Lee, Design and implementation of a GPON-based virtual OpenFlow-enabled SDN switch, vol. 34.

[39] R. e. a. Gu, „Software defined flexible and efficient passive optical networks for intra-datacenter communications,” *Optical Switching Networking,* vol. 14, p. 289, 2014.

[40] H. Kahlili, D. Rincón en S. Sallent, „Towards and integrated SDN NFV architecture for EPON networks," in *Advances in Communication Networking (LNCS 8846)*, Cham, 2014.

[41] L. Youngsuk, „A design of 10 Gigabit Capable Passive Optical Network(XG-PON1) architecture based on Software Defined Network (SDN)," in *International Conference on Information Networking (ICOIN)*, 2015.

[42] P. Parol en M. Pawlowski, „Towards networks of the future: SDN paradigm introduction to PON networking for business applications," in *Federated Conference on Computer Science and Information Systems*, Krakow, 2013.

[43] IEEE, „802.1Q-2011 - IEEE Standard for Local and metropolitan area networks--Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks". 31 Agosto 2011.

[44] „vconfig(8) - Linux man page," [Online]. Available: https://linux.die.net/man/8/vconfig. [Geopend 9 Mayo 2018].

[45] The Apache Software Foundation, „Versión 2.4 de la documentación del Servidor de HTTP Apache," [Online]. Available: http://httpd.apache.org/docs/2.4/. [Geopend 23 Mayo 2018].

[46] Raspberry Pi Foundation, „Raspberry Pi 3 Model B Specifications," [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. [Geopend 23 Mayo 2018].

[47] „iperf(1) - Linux man page," [Online]. Available: https://linux.die.net/man/1/iperf. [Geopend 23 Mayo 2018].

[48] G. Combs, „Wireshark," 1998. [Online]. Available: https://www.wireshark.org. [Geopend 23 Mayo 2018].

[49] T. B. O. M. F. T. a. TTPs, „Open Networking Foundation,” 2 Febrero 2015. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/TR_Multiple_Flow_Tables_and_TTPs.pdf. [Geopend 29 Mayo 2018].

[50] Y. U. I. i. OpenDayLight, „OpenDayLight Summit,” 29 Julio 2015. [Online]. Available: https://events.static.linuxfound.org/sites/events/files/slides/YANGUI-metz-malachovsky-sebin-ODL-Summit-final-July29.pdf. [Geopend 30 Mayo 2018].

[51] R. 6. YANG, „IETF,” Octubre 2010. [Online]. Available: https://tools.ietf.org/html/rfc6020. [Geopend 30 Mayo 2018].

[52] E. Dai en W. Dai, „Towards SDN For Optical Access Networks,” *Spring Technical Forum Proceedings,* 2016.

[53] O. Project, „OpenStack,” [Online]. Available: https://www.openstack.org. [Geopend 8 Junio 2018].

[54] I. S. C. i. M. Networks, „Department of Computer Science,” [Online]. Available: http://yuba.stanford.edu/~sd2/Ch_5.pdf. [Geopend 12 Junio 2018].

[55] Tianon ; Paultag, "Debian Official Images," 10 05 2020. [Online]. Available: https://hub.docker.com/_/debian.

[56] C. R. o. openflowplugin.git, „OpenDayLight,” 18 Septiembre 2013. [Online]. Available: https://git.opendaylight.org/gerrit/gitweb?p=openflowplugin.git;f=model/model-flow-base/src/main/yang/opendaylight-meter-types.yang;a=blob;hb=refs/heads/stable/boron. [Geopend 2018].

Pehrs, 2012.