Università degli Studi dell'Aquila

Ingegneria

Corso di laurea Magistrale in Ingegneria Informatica e Automatica

Experimental validation of priority queueing in SDN using Regression Trees and Model Predictive Control

Relatore

Prof. Alessandro D'Innocenzo

Studente

Marta Benedetti

Correlatore

Dott. Enrico Reticcioli

Matricola

254139

A.A. 2018/2019

Alla mia famiglia

Contents

List of Figures

List of Tables

LIST OF ABBREVIATIONS

SDN Software Defined Networking

OVS OpenVSwitch

OF OpenFlow

MPC Model Predictive Control

ML Machine Learning

RF Random Forest

RT Regression Tree

NN Neural Network

QP Quadratic Programming

CART Classification and Regression Tree

ToS Type of Service

DSCP Differentiated Service Code Point

NFV Network Function Virtualization

IT Information Technology

API Application Programming Interfaces

NBI Northbound Interfaces

SBI Southbound Interfaces

ONF Open Networking Foundation

JVM Java Virtual Machine

ONOS Open Network Operating System

TLS Transport Layer Security

IETF	Internet Engineering Task Force
D-ITG	Distributed Internet Traffic Generator

1 Introduction

In the last decade there was a radical change in the type of traffic passing through the network; mobile data traffic is growing continuosly year after year. However, in addition to the type of data, the type of users is changed significantly too. At the beginning, the network was developed in the University for academic, industrial and military purposes. Nowadays, the most part of people wishes to have internet access. Furthermore, there have been a sudden increase in the number of mobile devices with the expectation to perform whatever task, wherever, with any device. The arrival of new technologies as mobile devices, server and content virtualization, cloud services, are the key forces driving the networking industry today [1]. These new technologies force the networking industry to take a fresh look at the traditional network architectures currently in use.
These old architectures have a lot of limits such as:

- Complexity of networks which must fit with data, voice, and video on many different devices.

- Inconsistent network-wide policies required to configure thousands of devices and mechanisms.

- Inability to scale when the network becomes too much complex.

- Lack of industry-wide standards and protocols that create vendor-dependence.
- Non-heterogeneity between the devices (devices of different ages, devices from different manufacturers, devices of different cost ranges) which lead to difficult interoperability between them.

Some of the limits described above, can be ascribed to the fact that the implementation of traditional network devices couples two conceptually separated planes:

- control plane ("brain");
- forwarding plane ("muscles").

The need of a new type of network that is dynamic, agile, fast, easily controllable and reprogrammable arises: the Software Defined Networking (SDN) [2], section 2.1. In this way, a SDN controller has a global view of the network topology and it can configure the forwarding state of each SDN device (e.g. switch) by using a standard protocol called OpenFlow [3], section 2.2. Thanks to the OF counter variables (e.g. flow statistics, queue statistics, port statistics), the controller can retrieve information from the network devices and process them for Machine Learning (ML) and optimization purposes [4]. One of the many aspects that this controller can supervise in a SDN architecture is queue management on switch ports, using such measured data.
In this work, a SDN architecture has been created using real devices, the queues on the switch ports have been set and the statistics on the ports have been retrieved through OF counter variables. After collecting the data, it has been used to predict the bandwidth to allocate to the queues using Machine learning techniques, like Regression Trees and Random Forests [5]-[6], section 2.3. A predictive model of the switch behaviour is needed in order to apply advanced control techniques like Model Predictive Control (MPC). In MPC, control inputs, that minimize a certain objective function subjected to constraints, are computed by solving a corresponding optimization problem at each sampling instant [7]. Model Predictive Control is a closed loop system where at each sampling instant different optimization problems,

that are based on different available data, are solved. The goal is to apply optimal inputs to the process, to be controlled, in order to improve outputs. The MPC with Machine learning techniques, like Regression Trees and Random Forests, can be solved using standard Quadratic Programming (QP) solvers, instead of other ML techniques, like Neural Networks (NNs), based on nonlinear functions.
This thesis is organized as follows:

- Chapter 2 describes the MPC and how it is used in a SDN network with Machine Learning techniques. Then, it deals with the SDN architecture and its benefits, the most famous southbound protocol OpenFlow and two techniques of ML, Regression Trees and Random Forests.

- Chapter 3 explains the SDN architecture realized using real devices. For this reason it describes the controller (Ryu Controller), the OpenFlow switch realized with OpenVswitch (OVS), the two hosts that generate traffic through the D-ITG generator and the queues set on the OpenFlow switch. Finally, it treats the REST APIs provided by the Ryu controller, that are useful for retrieving the switch stats and updating the switch stats.

2 Problem Formulation

Advanced control techniques have been applied to predict the behavior of a real Software Defined Networking (SDN) starting from the avaible historical data. SDN, explained in section 2.1, aims to decuple network control and forwarding functions, enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services.
An advanced control technique is Model Predictive Control (MPC), [8]-[9], which is applied to optimize the performance of the SDN network. Model Predictive Control (MPC) is a consolidated technique to design optimal control strategies, leveraging the capability of a mathematical model to predict a system's behavior over a time horizon [10]. The basic idea is to substitute the classical control, figure 2.1 , with an optimal control, figure 2.2. In the figure 2.1 is rapresented the classical control, where there is a plant to be controlled according a control law. In the figure 2.2 the controller is substituted by an optimizer, in this case there

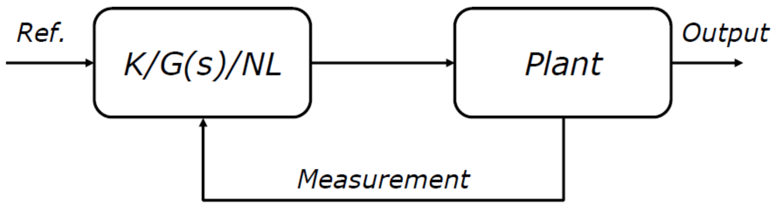


Figure 2.1: Classical control.

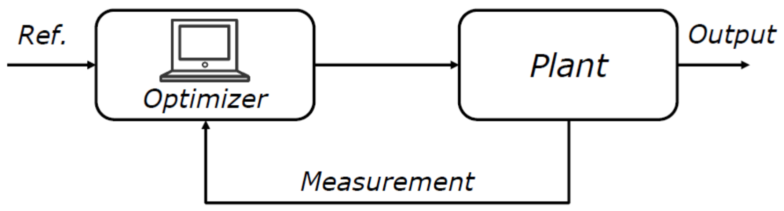


Figure 2.2: Optimal control.

isn't a control law but a board, a PC that will execute an optimization problem at every step. The plant model, which usually is used to calculate the control, in the Model Predective Control (MPC) is used to predict the behavior of the system over a finite horizon. An MPC formalization can be the following:

$$
\begin{aligned}
& u_k^*(x(k)) = \underset{u_k}{\operatorname{argmin}} \sum_{j=0}^{N-1} J(x_{k+j}, u_{k+j}) \\
& \text{subject to} \qquad x_{k+j+1} = f(x_{k+j}, u_{k+j}) \\
& \qquad x_{k+j} \in \mathcal{X} \\
& \qquad u_{k+j} \in \mathcal{U} \\
& \qquad x_k = x(k) \\
& \qquad u_k = \{u_k, u_{k+1}, \dots, u_{k+N-1}\}
\end{aligned}
\tag{2.1}
$$

At time k the current plant state is sampled and a cost minimizing control strategy is computed (via a numerical minimization algorithm) for a relatively short time horizon in the future: $[k, k+N]$. Only the first step of the control strategy is implemented, then the plant state is sampled again and the calculations are repeated starting from the new current state, yielding a new control and new predicted state path. The prediction horizon keeps being shifted forward and for this reason MPC is also called **receding horizon control**. To better understand this iterative process see figure 2.3. In this figure there is, in red, a reference trajectory $r(t)$, the only one to be followed, so that predicted outputs, brown line, are as close as possible to the red line. To make this, an optimal control sequence is computed at each instant of sampling. In the figure, the sky blue line represents the optimal control sequence at time k and so the first element of the sequence is applied in order to minimize the distance between the red line and brown line. This process is iterative

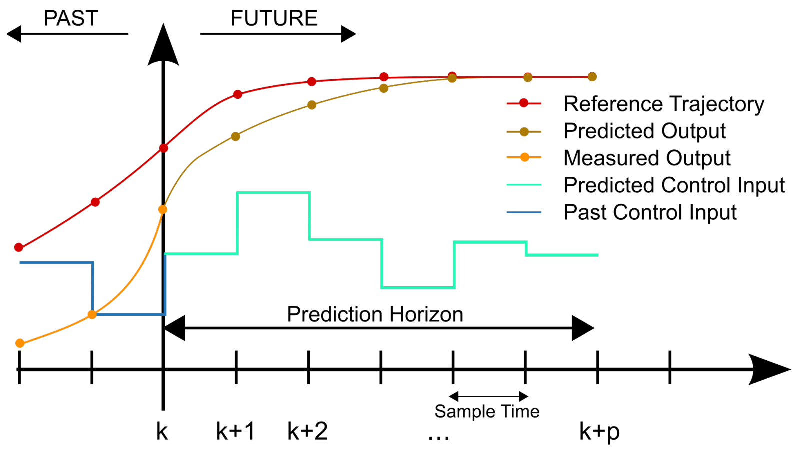


Figure 2.3: MPC scheme.
Fonte: `https://en.wikipedia.org/wiki/Model_predictive_control`

so to calculate the optimal input at the instant $k+1$, a different control sequence is re-computed which will probably be different from that at the instant k. Model Predictive Control (MPC) has been applied with data-driven models obtained using Regression Tree (RT) and Random Forests (RF); some basic informations of these techniques are shown in the section 2.3. These Machine Learning techniques are important to make a predition in order to apply optimal inputs to the process to improve outputs. The variables considered of the dynamical model are:

- $d(k) \in \mathbb{R}^8, \forall k$; these variables are considered disturbance variables (i.e. variables that cannot be modified by the control action) and are associated with the traffic generation entrusted to the D-ITG generator in section 3.3.

- $x(k) \in \mathbb{R}^3, \forall k$; these variables correspond to the packets sent by the queues differentiated for queues x_1 the "Default Queue" (DSCP 0,1,3), x_2 the "Premium Queue" (DSCP 2,4,6,7), x_3 the "Gold Queue" (DSCP 5).

- $u(k) \in \mathbb{R}^3, \forall k$; these variables correspond to bandwidth associated to the queues.

2.1 Sofware Defined Networking (SDN)

Software-Defined Networking (SDN) is a new networking paradigm that aims to make networks agile and flexible. SDN is meant to address the fact that the static

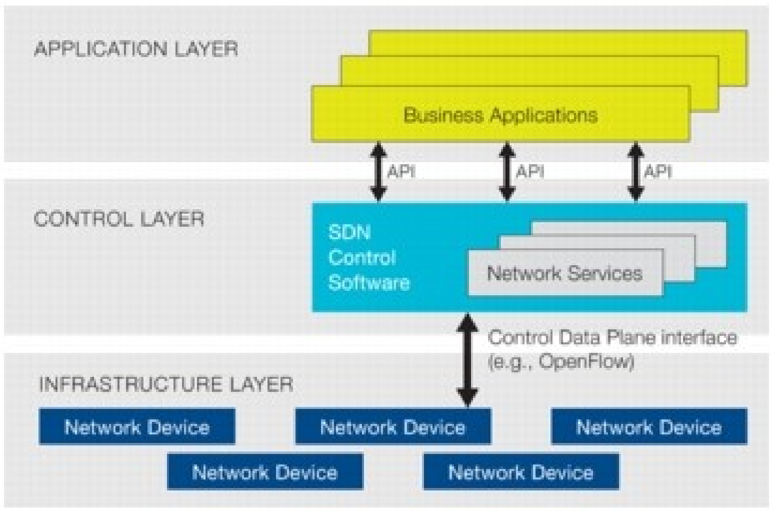


Figure 2.4: SDN architecture. Fonte: [1]

architecture of traditional networks is decentralized and complex while current networks require more flexibility and easy troubleshooting [11], [2]. SDN attempts to centralize network intelligence in one network component by disassociating the forwarding process of network packets from the routing process, so it wants to deacuple the data plane from the control plane, the "muscles" from the "brain". A typical representation of a SDN architecture comprises three layers, figure 2.4.

- **Application layer**: are programs that explicitly, directly, and programmatically communicate their network requirements and desired network behavior to the SDN Controller.
- **Control layer**: represents the centralized SDN controller software that acts as the brain of the software defined network. This controller resides on a server and manages policies and the flow of traffic throughout the network.
- **Infrastructure layer**: is composed of various networking equipment which forms underlying network to forward network traffic. It could be a set of network switches and routers in the data centre.

These three layers communicate using respective **northbound** and **southbound** application programming interfaces (APIs). For example, applications talk to the controller through its northbound interface (NPI), while the controller and switches communicate using southbound interfaces (SPI), such as with the **OpenFlow** protocol, explained on the section 2.2, although other protocols exist.

2.1.1 Benefits of the SDN architecture

The advantages generated by the transition from a traditional network to the SDN paradigm are a lot, however opinions and attitudes of the various bodies, organizations and companies are sometimes discordant; these are often influenzed by their role and their economic interests rather than the real technical value that the introduction of this technology would bring. Now, a lot of research bodies, like University, are switching to hardware that supports SDN and OpenFlow. Adopting this technology offers the possibility of reducing costs, as it is mostly released under an Open Source license. While some institutions are switching to SDN and OpenFlow, as seen before, others interpret the introduction of SDN as the ruin of their market. Indeed, the strength of these companies is exactly to produce hardware with proprietary sofware but with the advent of SDN, network elements, like switch and router, would become simple low-cost hardware devices whose trademark makes no difference.
However some benefits of the SDN architecture are the following:

- **directly programmable**: network control is directly programmable because it is decoupled from forwarding functions;
- **centrally managed**: network intelligence is centralized in software-based SDN controllers that maintain a global view of the network;
- **flexible and agile**: to allow to rapidly reconfigure the network, adapting it to variable needs;
- **programmatically configured**: SDN lets network managers manage, secure, configure, and optimize network resources very quickly through dynamic SDN programs, which they can write themselves because the programs don't depend on proprietary software;

- **Independence from the vendor**: the network administrator defines more abstract rules, that subsequently are automatically mapped on configurations that are provided in different "languages" for different manufacturers.

2.2 OpenFlow

OpenFlow, the most famous southbound protocol, is an open source protocol and it is used for communication between a controller and a switch in a SDN architecture. OpenFlow enables network controllers to determine the path of network packets across a network of switches, [12]. In a SDN architecture the controllers are distinct from the switches and thanks to this protocol these levels are able to communicate. In the figure 2.5 there is the symbol of the OF protocol.

Figure 2.5: OpenFlow symbol. Fonte:[12]

2.2.1 Birth and development

The Openflow protocol was born in 2006 to a funded project at Stanford University. A student, M. Casado, developed Ethane, a new network architecture [13]. Ethane sought a better way to deal with security by having fine-grain flow-based security policies. A centralised controller managed these policies. OpenFlow was proposed in a paper published in 2008 [3]. The idea presented was a way for network researchers to experiment at line-rate on heterogeneous switches and in real-world traffic settings. In 2011 the Open Networking Foundation (ONF), a non-profit association, was founded whose purpose is to innovate the network through SDN technologies and standardize the OpenFlow protocol with related technologies. The first version that was standardized by the ONF is v1.0.0, relased in in December 2009. The current version of OpenFlow is 1.5.1. However, version 1.6 has been available since September 2016, but accessible only to ONF's members.

2.2.2 OpenFlow Switch

An OpenFlow Switch consists of one or more *flow tables* 2.2.2, a *group table* 2.2.2, which perform packet forwarding and lookups, *OpenFlow ports* 2.2.2 and an *OpenFlow channel* 2.2.2 to an external controller, like figure 2.6. Using the

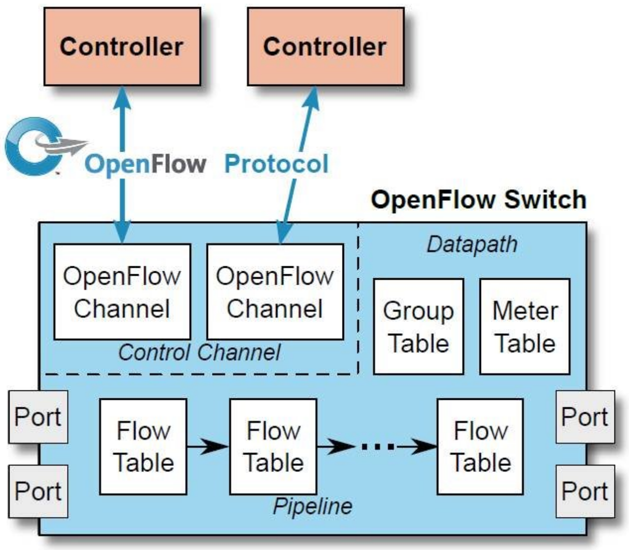


Figure 2.6: Main components of an OpenFlow switch. Fonte: [12]

OpenFlow protocol, the controller can add, update, and delete *flow entries* in flow tables, both reactively (in response to packets) and proactively. Each flow table in the switch contains a set of flow entries; each flow entry consists of other fields that will be described later in 2.2.2.

OpenFlow ports

There are a lot of types of OpenFlow ports. OpenFlow ports are the network interfaces for passing packets between OpenFlow processing and the rest of the network, [12]. OpenFlow packets are received on an ingress port and processed by the OpenFlow pipeline to an output port. An OpenFlow switch must support three types of OpenFlow ports.

- **Physical ports**. The OpenFlow physical ports are switch defined ports that correspond to a hardware interface of the switch. In some deployments, the OpenFlow switch may be virtualised over the switch hardware.

- **Logical ports**. The OpenFlow logical ports are switch defined ports that don't correspond directly to a hardware interface of the switch, but are an higher level abstractions independent from the OF protocol.

- **Reserved Ports**. The OpenFlow reserved ports define generic forwarding actions like sending to the controller, flooding, or forwarding method using non-methods OpenFlow. A switch is not required to support all reserved ports but the necessary ones are:

  - **ALL**: represents all ports the switch can use for forwarding a specific packet, a copy of the packet is sent to all standard ports[1], excluding the packet ingress port and ports that are configured OFPPC_NO_FWD.
  - **CONTROLLER**: represents the control channel with the OpenFlow controller that can be used as an ingress port or as an output port. In the first case, the packages are marked as received by the controller, while in the second case it is necessary to encapsulate the messages sent via OF protocol.
  - **TABLE**: this port can only be used as outputs and allows you to direct a specific message towards comparison with the first table in order to have it processed correctly.
  - **IN_PORT**: represents the packet ingress port that can be used only as an output port, send the packet out through its ingress port.
  - **ANY**: special value, it is used when an operation can be performed on any port.
  - **UNSET**: special value used when the output port isn't set in the Action-Set.

Other ports like LOCAL, FLOOD and NORMAL are optional.

[1]The OpenFlow standard ports are defined as physical ports, logical ports, and the LOCAL reserved port if supported.

Flow tables

A flow table in the switch contains a set of flow entries, each flow entry consists of other fields described in the table 2.1.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie	Flags

Table 2.1: Main components of a flow entry in a flow table.

- **Match Fields**. To match against packets. These consist of the ingress port, packet headers and optional other fields.
- **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 flow is expired by the switch.
- **Cookie**. Data value chosen by the controller, may be used by it to filter flow entries affected by flow statistics, flow modification and flow deletion requests. Not used when processing packets.
- **Flags**. Used to differentiate voice management.

Each flow entry in the table is uniquely determined by the fields of *match fields* and *priority.* If two flow entries have the same *match fields* then they must have different priority.

Group table

A group table consists of group entries, [12]. A group entry may consist of zero or more buckets. A bucket typically contains actions that modify the packet and an output action that forwards it to a port. Each group entry is identified by its group identifier, like shown in the table 2.2.

Group Identifier	Group Type	Counters	Action Buckets

Table 2.2: Main components of a group entry in the group table.

- **Group Identifier**: a 32 bit unsigned integer uniquely identifying the group on the OF switch.
- **Group Type**: to determine group type. A switch is not required to support all group types, the necessary ones are:
  - **indirect**: this group supports only a single bucket and so execute the one in this group.
  - **all**: execute all buckets in the group.
- **Counters**: updated when packets are processed by a group (like Counters in flow table 2.2.2).
- **Action Buckets**: an ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters.

OpenFlow channel and messages used

The OpenFlow channel is the interface that connects each OpenFlow Logical Switch to an OpenFlow controller. Through this interface, the controller configures and manages the switch in order to forward and receive packets between the two devices. The control channel of a switch can support one or more OF communication channels with one or more controllers. The OpenFlow channel is usually encrypted using TLS[2] or it can be used the TCP protocol. The OpenFlow protocol supports three types of exchanged messages.

- **Controller-to-switch**. The communication is always initiated by the controller, and is used to configure or query the status of the switch. Usually

[2]Transport Layer Security (TLS), and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols designed to provide communications security over a computer network. TLS is a Internet Engineering Task Force (IETF) standard, first defined in 1999, and the current version is TLS 1.3 defined in RFC 8446 (August 2018).

the controller query the switch about its basic functionality, its specications techniques and modification of the entries in the table.

- **Asynchronous**. These messages are sent by the switch without prior request from the controller. There are: `Packet − in`, `Packet − out` o `Flow − mod`, `Flow − Removed`, `Port − status`, `Role − status`, `Controller − status` and `Flow − monitor`.

- **Symmetric**. Symmetric messages are sent without solicitation, in either direction. They include messages to establish a new connection like `Hello` or messages to check the connection status like `Echo`.

2.3 Machine Learning

Machine Learning is a sub-field of artificial intelligence that provides models with the ability to automatically learn and improve from experience without being explicitly programmed, see [5]. There exist a wide range of models that make uses ML approaches. This work focus on Regression Tree, on subsection 2.3.1, and Random Forests, on subsection 2.3.2.

2.3.1 Regression Trees

The RT is a type of decision tree. There are two type of decision trees: the classification trees and the regression trees. Both types of decision trees fall under the **Classification and Regression Tree (CART)** designation. The difference is that the first type has a fixed or categorical target variable while the second one has a continuous target variable. The figure 2.7 rapresents on the right an example of the Regression Tree while on the left an example of the Classification Tree. A regression tree is built through a process known as binary recursive partitioning, which is an iterative process that splits the data into partitions or branches, and then continues splitting each partition into smaller groups as the method moves up each branch. An example of this binary recursive partitioning is in figure 2.8. On the right the recursive partition is rapresented by a tree while on the left by a

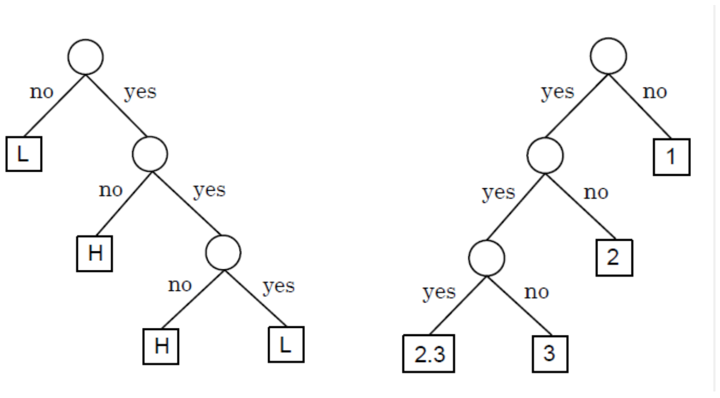


Figure 2.7: Example of Regression Tree and Classification Tree.

set of rectangles. It is interesting to note how the terminal nodes (or leaves), on the right figure, exactly correspond to the regions $\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_5$. In general, each

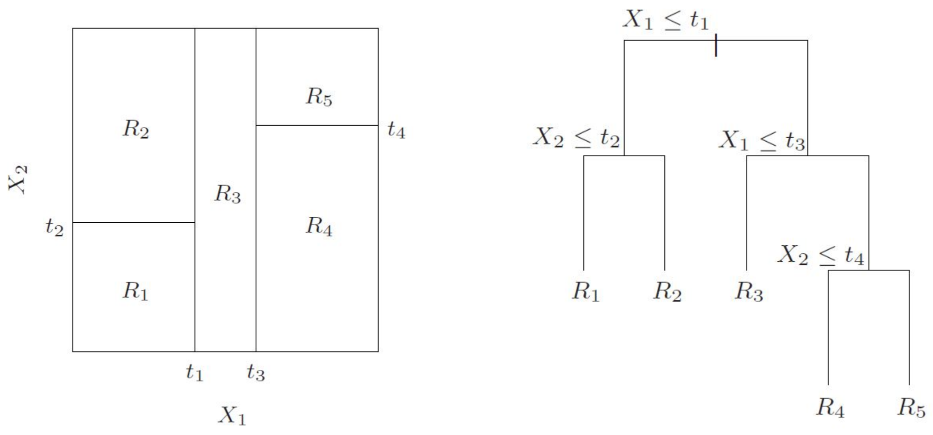


Figure 2.8: Partitioning via RT.

root node rapresents a single input variable ($\mathtt{X}$) and a split point on that variable, for example in the figure 2.8 the input variable is $\mathtt{X}_1$ and the first split point is $\mathtt{t}_1$. The leaf nodes of the tree contain an output variable ($\mathtt{Y}$) which is used to make prediction, in the figure are $\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_5$. The selection of which input variable to use and the specific split or cut-point is chosen using a greedy algorithm to minimize a cost function. The split with the best cost (lowest cost) is selected. Tree construction ends using a predefined stopping criterion. The most common stopping procedure is to use a minimum count on the number of training instances assigned to each leaf node. If the count is less than some minimum then the split is not accepted and the node is taken as a final leaf node.

2.3.2 Random Forests

A Random Forest consists of a collection of simple tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest, see [6]. The prediction is given by avaraging the response of all the trees in the forest. The error for forests converges as number of trees in the forest becomes large and also depends on the strength of the individual trees in the forest and the correlation between them. Some advantages of RF are the following.

- They are considered as a highly accurate and robust method because of the number of decision trees participating in the process.
- They do not suffer from the overfitting problem. The main reason is that it takes the average of all the predictions.
- They can be parallelizable, meaning that we can split the process to multiple machines to run. This results in faster computation time.
- They also maintains accuracy even when a large proportion of the data are missing.
- They have less variance than a single decision tree. It means that it works correctly for a large range of data items than single decision trees.

On the other hand, random forests also have a few disadvantages.

- Training a large number of deep trees can have high computational costs and use a lot of memory.
- The model is difficult to interpret compared to a decision tree, where you can easily make a decision by following the path in the tree.

3 Experimental setup

In this chapter the SDN architecture realized using real devices will be explained, in order to apply MPC to do the predictive control on the queue bandwidth.

3.1 SDN architecture

The architecture used is shown schematized in figure 3.1 and in reality in figure 3.2. For simplicity, the figure 3.1 will be described because it is easier to understand. The architecture shown in the figure 3.1 is a SDN architecture and, as explained in chapter 2, there is a separation between the control plane and the data plane. The controller, the intelligent part of the network, that has been used, is Ryu controller; some of its features are explained on 3.1.1. The Ryu controller has been run on a UDOO x86 Advanced with an Intel Braswell N3160 processor up to 2.24 GHz and 4 GB of RAM [14].The data plane in the figure is represented by a switch, this is dumb and fast, and its task is to forward packets. For the switch a Raspberry Pi 3[1] with OpenVSwitch (OVS) has been used. To convert a Raspberry into an Openflow switch a lot of cmd commands has been executed, that are shown in the section 3.1.2. In the architecture there are two host (`HOST_DX` and `HOST_SX`)

[1]The Raspberry Pi is a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation to promote teaching of basic computer science in schools and in developing countries.

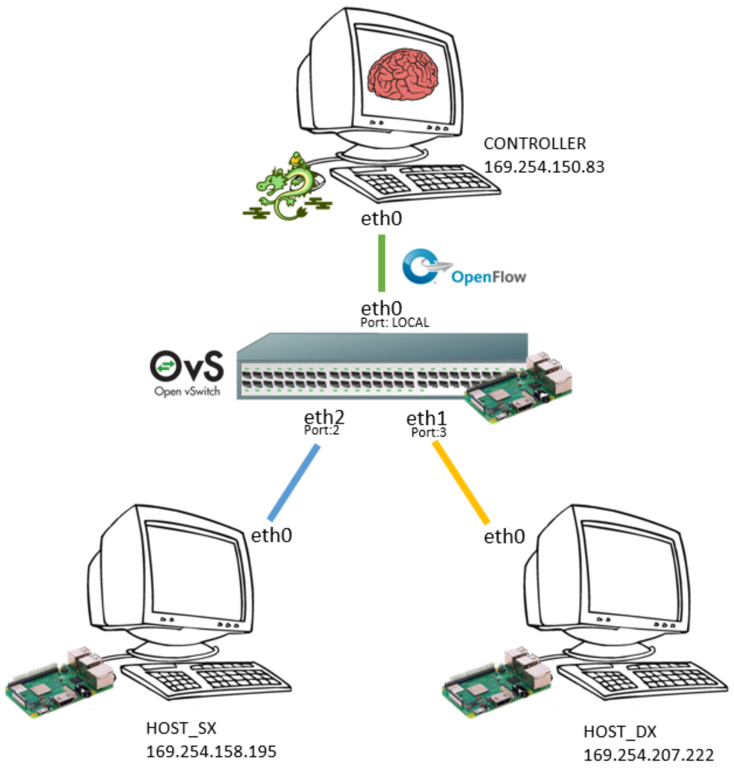


Figure 3.1: Schema of the architecture.

Figure 3.2: The architecture in reality.

and, also for these, two raspberries have been used. The task of the hosts is to generate traffic. Traffic generation is entrusted to the D-ITG generator, explained in the section 3.3, which uses informations inside a database. It is created from real measures that contains the number of packets trasmitted and received by the switch. The database fields have been divided for each type of Differentiated Service Code Point (DSCP) with a sampling time of 5 minutes. DSCP is the modern definition of the Type of Service (ToS) field in which the first 6 bits are the Differentiated Services field that are in common with ToS field, and the last 2 bits are the explicit congestion notification [15]- [16]. All these devices communicate with ethernet cables. In particular the raspberry-switch has only one ethernet port and so, to create other ethernet connections, usb-ethernet adapters have been used. The raspberry-switch communicates with the controller through the eth0 port (`Port : LOCAL`), this is internal to the device, while it communicates with the hosts through eth1 (`Port : 3`) and eth2 (`Port : 2`) ports. In figure 3.1 these connections have been applied with different colours because they correspond to the colors of the cables in reality, figure 3.2.
In this thesis the behavior of the switch has been considered according to the rules used by the provider *Telecom Italia* [17]. For this reason, the eth2 port of the switch has been configured with a maximum packet rate of 100 *MB/s* and three queues have been defined for it, while the eth1 port has been configured with a maximum packet rate of 100 *MB/s* and with an only queue. The prioritization is based on the Differentiated Service Code Point (DSCP) number. According to this number, the queues have been configured as follows: packets with the DSCP values 0, 1 and 3 are routed on queue 0, "**Default Queue**", with maximum rate of 20 *MB/s*; the packets with values 2, 4, 6 and 7 are routed on queue 1, "**Premium Queue**", with maximum rate of 80 *MB/s*; the packets with value 5 are routed on queue 2, "**Gold Queue**", with minimum 100 *MB/s*. With this configuration, the Gold Queue uses the maximum capacity of the port to send packets, indeed, its minimum bit rate that it can use to send packets corresponds to the maximum packet rate of the switch. The Default Queue, instead, can use the packet rate from 0 *MB/s* to 20

MB/s while the Premium Queue from 0 MB/s to 80 MB/s. Indeed, for the port number 3 (eth1), I configure only a queue with can use the packet rate from 0 MB/s to 100 MB/s. In figure 3.3 there is a schema of the eth2 port, explained before. The approach introduced by *Telecom Italia* is a static approach and is in contrast to the dynamic one that has been used in order to have improvements in terms of reducing Packet Losses, and thus improvement of the Quality of Service. This dynamic approach has been used with advanced control techniques such as ML and MPC, explained in chapter 2, where the packets sent by the queues will be collected in the state vector to be controlled $x(k) \in \mathbb{R}^3, \forall k$, while the bandwidth associated to the queues will compose the control variable $u(k) \in \mathbb{R}^3, \forall k$. A schema of the control architecture can be visualized in the figure 3.4.

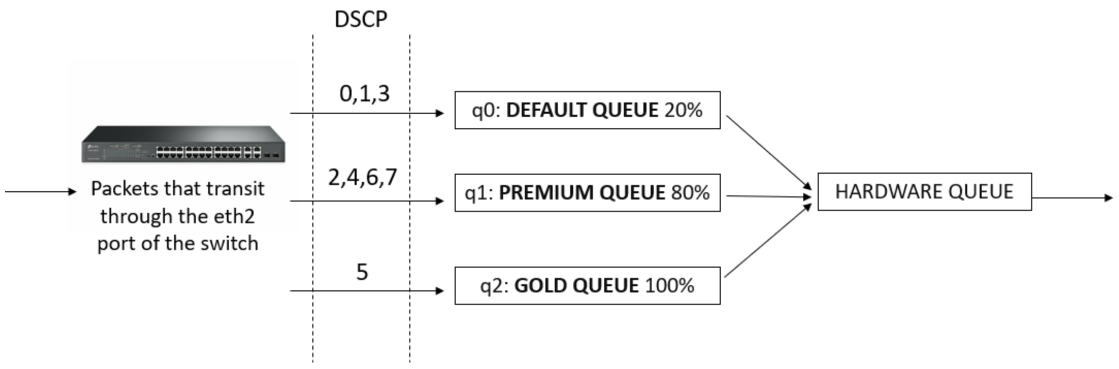


Figure 3.3: Static queues rate with routed packets relative to DSCP for eth2 port.

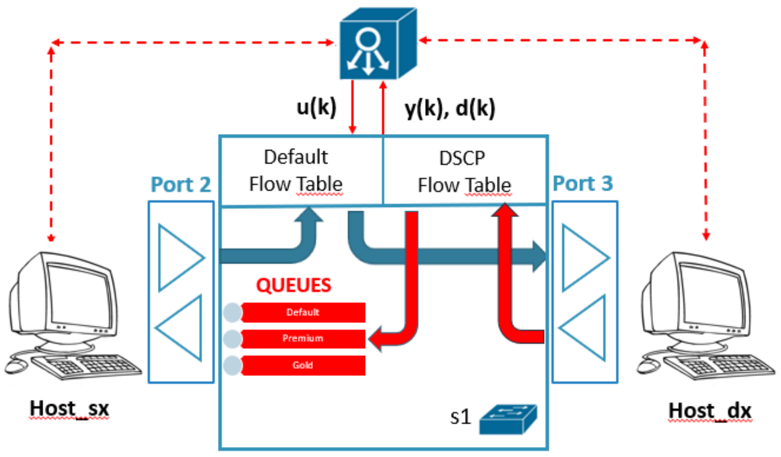


Figure 3.4: Control architecture.

3.1.1 Ryu Controller

Ryu is a component-based software defined networking framework. Ryu provides software components with well defined API that make it easy for developers to create new network management and control applications [18]. Ryu supports various protocols for managing network devices, such as OpenFlow, Netconf, OF-config, etc. About OpenFlow, Ryu supports fully 1.0, 1.2, 1.3, 1.4, 1.5 and Nicira Extensions. All of the code is freely available under the Apache 2.0 license. Ryu is fully written in Python. To install Ryu there are two options:

1. Using **pip** command.
   ```
           pip install ryu
   ```

2. From the **source code**.
   ```
       git clone git://github.com/osrg/ryu.git
       cd ryu; python ./setup.py install
   ```

3.1.2 Raspberry Pi 3 with OpenvSwitch

Open vSwitch (OVS) is a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license [19]- [20]. The main purpose of Open vSwitch is to provide a switching stack for hardware virtualization environments, while supporting multiple protocols and standards used in computer networks. In this section the conversion of the raspberry into an OpenFlow switch through OVS will be explained, following the tutorial in [21]. This tutorial starts by downloading OVS typing this command.

```
wget https://www.openvswitch.org/releases/openvswitch-2.12.0.tar
```

The url has been obtained from the website of OpenVSwitch, in the section of download, in this case the most recent release has been taken. Since it is a zipped archive (`.tar`) it has been unzipped.

```
tar -xvzf openvswitch-2.12.0.tar.gz
```

Entering in the directory that has been unzipped, with the common command (`cd`), the following commands have been typed.

```
apt-get install python-simplejson python-qt4 libssl-dev python-twisted-conch
    automake autoconf gcc uml-utilities libtool build-essential pkg-config
apt-get install linux-headers-4.9.0-6-rpi
```

Always in the openvswitch folder, these three commands have been typed, shown below. The installation may take some time.

```
./configure --with-linux=/lib/modules/4.9.0-4-rpi/build
make
make install
```

The openvswitch module in the correct folder has been turned on.

```
cd openvswitch-2.12.0/datapath/linux
modprobe openvswitch
```

A script `script.sh` has been created.

```
nano script
```

This code has been copied into the previous script.

```
#!/bin/bash
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
ovs-vsctl --no-wait init
ovs-vswitchd --pidfile --detach
ovs-vsctl show
```

The $\texttt{ovs} - \texttt{vswitchd.conf}$ file for the database, containing the details of the switch, and the directory `openvswitch` have been created with these two commands.

```
touch /usr/local/etc/ovs-vswitchd.conf
mkdir -p /usr/local/etc/openvswitch
```

The database has been created, which will be used by the OVS.

```
./openvswitch-2.5.2/ovsdb/ovsdb-tool create /usr/local/etc/openvswitch/conf.db
    /root/openvswitch-2.5.2/vswitchd/vswitch.ovsschema
```

The `script.sh` has been run.

```
-chmod +x script
./script
```

The switch has now been created. The next steps are to configure and start the switch. First of all, a new bridge `s1` has been added.

```
ovs-vsctl add-bridge s1
```

Then, the raspberry-switch has been connected to the controller and to the two hosts through the ethernet cables. The ethernet port internal to the raspberry (eth0) has been used for the first, while the ports eth1 and eth2 for the seconds, as shown in figure 3.1. The port to the bridge has been added with these commands.

```
ovs-vsctl add-port s1 eth1
ovs-vsctl add-port s1 eth2
```

The eth0 port has not been added because, being internal to the device, if it is added, the connectivity with the controller is lost, [22]. To establish a communication with the Ryu controller this command has been entered.

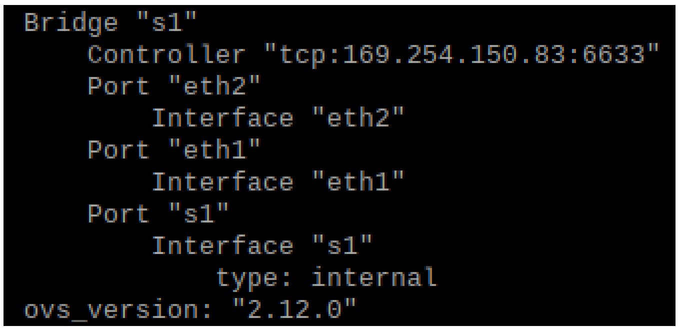
```
ovs-vsctl set-controller s1 tcp:169.254.150.83:6633
```

For simplicity, the Datapath ID and the port numbers eth1 and eht2 has been set with commands shown below.

```
ovs-vsctl set bridge s1 other-config:datapath-id=0000000000000002
ovs-vsctl add-port s1 eth1 -- set interface eth1 ofport_request=3
ovs-vsctl add-port s1 eth2 -- set interface eth2 ofport_request=2
```

A useful command that shows a brief overview of the database contents is the following while the result of it is illustrated in the figure 3.5.

```
ovs-vsctl show
```



```
Bridge "s1"
    Controller "tcp:169.254.150.83:6633"
    Port "eth2"
        Interface "eth2"
    Port "eth1"
        Interface "eth1"
    Port "s1"
        Interface "s1"
            type: internal
ovs_version: "2.12.0"
```

Figure 3.5: Overview of the database contents

3.1.3 Setting Telecom queue

As explained in 3.1, three queues have been configured on eth2 port and one on the eth1 port, following the tutorial on [23].

The tutorial starts by typing these commands on the raspberry-switch:

```
ovs-vsctl set Bridge s1 protocols=OpenFlow13
ovs-vsctl set-manager ptcp:6632
```

With the first command, the version 1.3 of OpenFlow has been set in the switch `s1`. With the second one, listening to port 6632 has been set to access OVSDB.
After typing these two simple commands on the raspberry-switch, all the other ones have been set on the controller; starting by setting `ovsdb_addr` in order to access OVSDB.

```
curl -X PUT -d '"tcp:169.254.153.160:6632"'
    http://localhost:8080/v1.0/conf/switches/0000000000000002/ovsdb_addr
```

The address `169.254.153.160` is the switch address and `0000000000000002` is the Datapath ID.
As explained in 3.1, the eth2 and eht1 ports behave differently, because the first has three queues and the second only one. For this reason, the bandwidth and the rules on the queues have been set in different ways.

- **Eth2 port (port number 2)**.

  The bandwidth has been set according to the 3.1 table and with the command below.

Queue ID	Queue Name	Max Rate	Min Rate
0	Default	20Mbit/s	0Mbit/s
1	Premium	80Mbit/s	0Mbit/s
2	Gold	100Mbit/s	100Mbit/s

Table 3.1: Set different bandwidth for different queues of the eth2 port

```
curl -X POST -d '{"port_name":"eth2",
    "type":"linux-htb", "max_rate":"100000000",
    "queues":[{"max_rate":"20000000"},
    {"max_rate":"8000000"}, {"min_rate": "100000000"}]}'
    http://localhost:8080/qos/queue/0000000000000002
```

  Later, the rules on the eth2 port of the switch have been set up to forward packets according to the DSCP numbers:

  – DSCP equal to 0,1,3 on queue 0;

  - DSCP equal to 2,4,6,7 on queue 1;
  - DSCP equal to 5 on queue 2.

  Really these values do not correspond to the DSCP number but to `IP Precedence Values` (`IP_Prec`) [2]. The mapping between DSCP and `IP_Prec` is shown in the table 3.2, [24]-[25].

  In this table 3.2 are shown the rules for sending packets on different queues, according to the DSCP field. The Destination Address, in the table 3.2, has been set to `169.254.207.222`. This address corresponds to the host address on the right (`HOST_DX`) in the figure 3.1, and so the table represents the packets that are sent from the host on the left (`HOST_SX`) to the host on the right (`HOST_DX`). Indeed, on the port 2 of the `HOST_SX` has been set the queues.

  For each rule different commands have been typed, modifying the parameters `QueueID`, `DestinationAddress` and `DSCP` according to table 3.2. For example the first row of the table is represented by this command:

```
curl -X POST -d '{"match": {"nw_dst":
    "169.254.207.222","ip_dscp": "0"},
    "actions":{"queue": "1"}}'
    http://localhost:8080/qos/rules/0000000000000002
```

- **Eth1 port (port number 3)**.

  The bandwidth has been set according to the 3.3 table and with the command below.

```
curl -X POST -d '{"port_name":"eth1",
    "type":"linux-htb", "max_rate":"100000000",
    "queues":[{"max_rate":"100000000"}]}'
    http://localhost:8080/qos/queue/0000000000000002
```

  In this case the rules to forward packets, according to the DSCP numbers, are irrelevant because all the packets are sent on the queue 0, the only existing one.

[2] The old definition of the ToS field reserved the first 3 bits for IP precedence and the others five bits for ToS. While, the modern definition of it reserved the first 6 bits for the Differentiated Services (DSCP) field, that are in common with ToS field, and the last 2 bits for the explicit congestion notification.

Queue ID	Destination Address	DSCP	IP_Prec
0	169.254.207.222	0	0
0	169.254.207.222	8	1
0	169.254.207.222	10	1
0	169.254.207.222	12	1
0	169.254.207.222	14	1
0	169.254.207.222	24	3
0	169.254.207.222	26	3
0	169.254.207.222	28	3
0	169.254.207.222	30	3
1	169.254.207.222	16	2
1	169.254.207.222	18	2
1	169.254.207.222	20	2
1	169.254.207.222	22	2
1	169.254.207.222	32	4
1	169.254.207.222	34	4
1	169.254.207.222	36	4
1	169.254.207.222	38	4
1	169.254.207.222	48	6
1	169.254.207.222	56	7
2	169.254.207.222	40	5
2	169.254.207.222	46	5

Table 3.2: Set queue rules according to the DSCP number

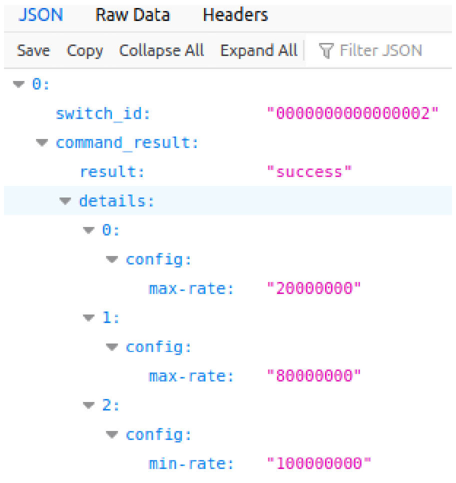
Queue ID	Max Rate	Min Rate
0	100Mbit/s	0Mbit/s

Table 3.3: Set bandwidth for the only queue of eth1 port

So, looking at the table 3.2 of eth2, the `QueueID` values will always be equal to 0 and the `DestinationAddress` obviously changes to `169.254.158.195`.

All these commands, described above, have not been typed using the command line but a python script (`real.py`) has been created in order to retrieve all the switch information (Datapath ID, Port number) from the `REST_APIs`, section 3.2, using an already existing python code `Controller_commands.py`. In the appendice A there is the python script `real.py`.

To verify the correct creation of the queue, the calls `http://localhost:8080/qos/queue/0000000000000002` and `http://localhost:8080/qos/rules/0000000000000002` have been used and also some `REST APIs`, in particular the call `http://localhost:8080/stats/queue/<dpid>`. The result of this last call is visible in the figure 3.10, while the results of previous calls are visible in 3.6 and 3.7. The call `http://localhost:8080/`



```
▼ 0:
    switch_id:          "0000000000000002"
  ▼ command_result:
      result:           "success"
    ▼ details:
      ▼ 0:
        ▼ config:
            max-rate:   "20000000"
      ▼ 1:
        ▼ config:
            max-rate:   "80000000"
      ▼ 2:
        ▼ config:
            min-rate:   "100000000"
```

Figure 3.6: Queue set on the switch.

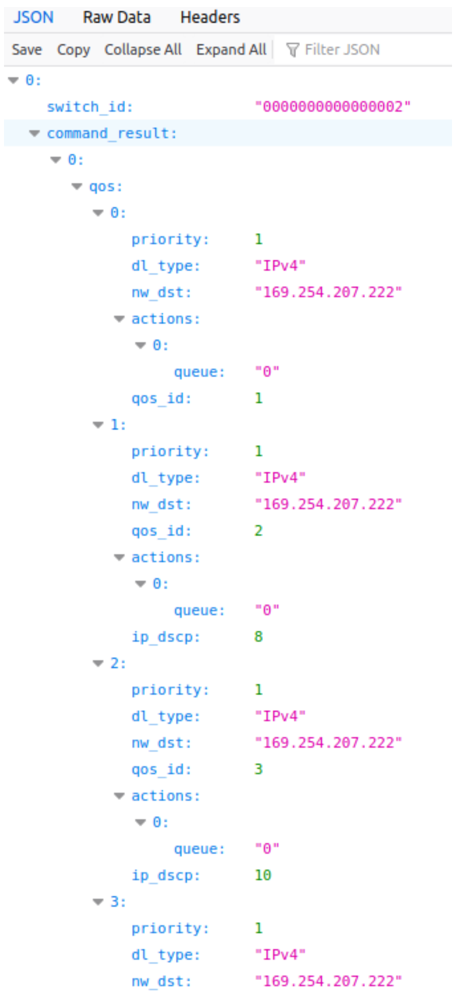
`qos/queue/0000000000000002`, in the figure 3.6, save only the last queues set and so, if the last port analyzed is eth1, this call save the queues according to the table 3.3, otherwise to the table 3.1.

Furthermore, it is possible to see the creation of queues on the raspberry-switch, typing on it, this command:

```
ovs-vsctl list queue
```

3.2 REST APIs

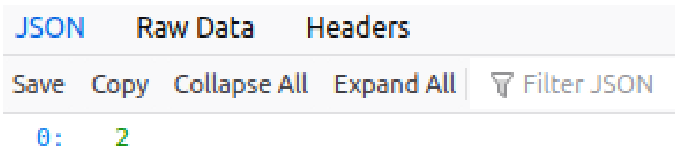
One of the best ways to get a look at the current state of OpenFlow switches connected to Ryu is through the use of its REST API. With this Application Programming Interfacing (API) it is possible to get all sorts of information about the switches while also manually installing new flows, groups, and meters. APIs are used for queue control and counter recovery from the switch. Thanks to REST APIs, it is easier to debug this application and obtain statistics. In this section, some calls will be explained, which are essential to control communication between devices,



```
0:
  switch_id:              "0000000000000002"
  command_result:
    0:
      qos:
        0:
          priority:       1
          dl_type:        "IPv4"
          nw_dst:         "169.254.207.222"
          actions:
            0:
              queue:      "0"
          qos_id:         1
        1:
          priority:       1
          dl_type:        "IPv4"
          nw_dst:         "169.254.207.222"
          qos_id:         2
          actions:
            0:
              queue:      "0"
          ip_dscp:        8
        2:
          priority:       1
          dl_type:        "IPv4"
          nw_dst:         "169.254.207.222"
          qos_id:         3
          actions:
            0:
              queue:      "0"
          ip_dscp:        10
        3:
          priority:       1
          dl_type:        "IPv4"
          nw_dst:         "169.254.207.222"
```

Figure 3.7: Some rules set on the switch to send packages.

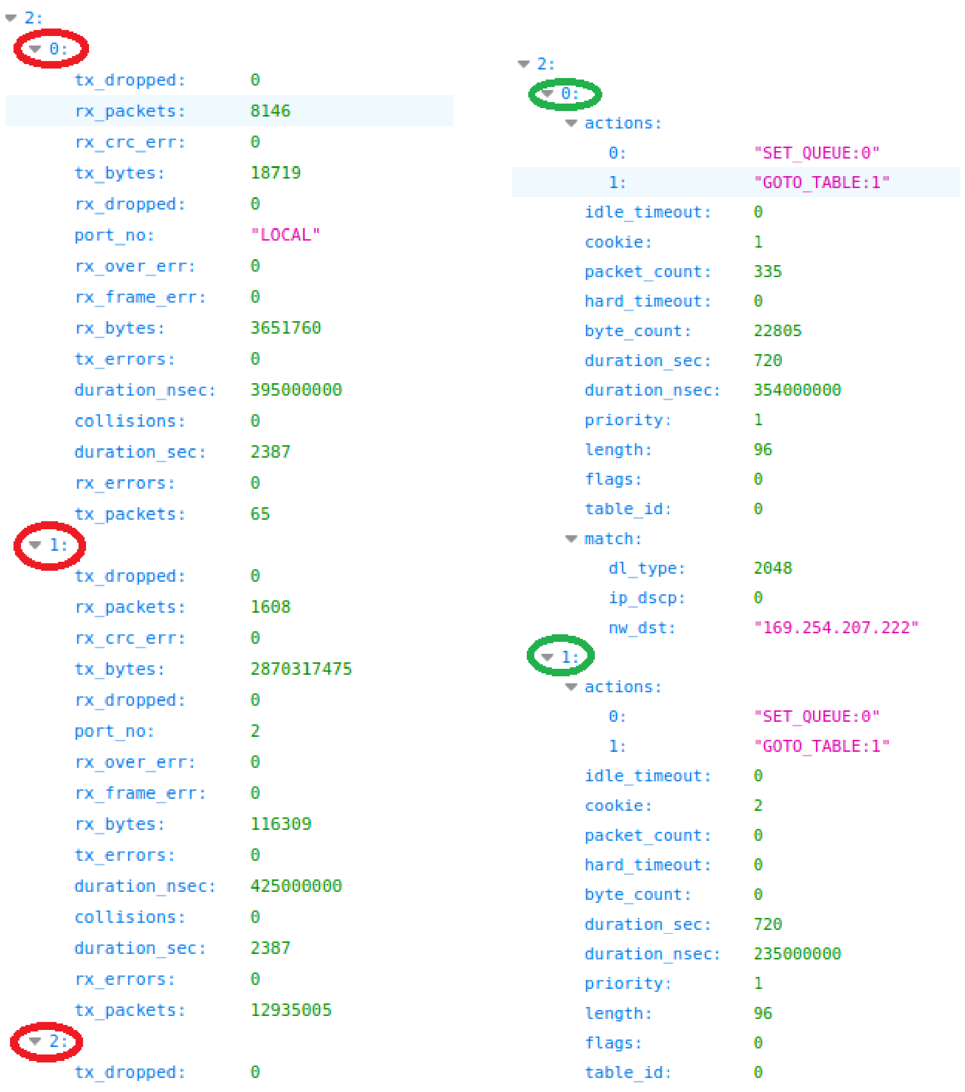
port statistics and queue settings. These calls are in `ryu.app.ofctl_rest` [26]. The first call that has been made is: `http://localhost:8080/stats/switches`. From this, the list of all switches connected to the controller have been obtained, in this case is only one and it is shown in figure 3.8. The number shown



```
0:   2
```

Figure 3.8: List of all switches.

in the figure 3.8 is the datapath ID, this is usually chosen randomly but for simplicity it has been permanently assigned with the command shown previously in 3.1.2. Other important calls are `http://localhost:8080/stats/port/<dpid>`



```
▾ 2:
  ▾ 0:
      tx_dropped:      0
      rx_packets:      8146
      rx_crc_err:      0
      tx_bytes:        18719
      rx_dropped:      0
      port_no:         "LOCAL"
      rx_over_err:     0
      rx_frame_err:    0
      rx_bytes:        3651760
      tx_errors:       0
      duration_nsec:   395000000
      collisions:      0
      duration_sec:    2387
      rx_errors:       0
      tx_packets:      65
  ▾ 1:
      tx_dropped:      0
      rx_packets:      1608
      rx_crc_err:      0
      tx_bytes:        2870317475
      rx_dropped:      0
      port_no:         2
      rx_over_err:     0
      rx_frame_err:    0
      rx_bytes:        116309
      tx_errors:       0
      duration_nsec:   425000000
      collisions:      0
      duration_sec:    2387
      rx_errors:       0
      tx_packets:      12935005
  ▾ 2:
      tx_dropped:      0
```

```
▾ 2:
  ▾ 0:
    ▾ actions:
        0:             "SET_QUEUE:0"
        1:             "GOTO_TABLE:1"
      idle_timeout:    0
      cookie:          1
      packet_count:    335
      hard_timeout:    0
      byte_count:      22805
      duration_sec:    720
      duration_nsec:   354000000
      priority:        1
      length:          96
      flags:           0
      table_id:        0
    ▾ match:
        dl_type:       2048
        ip_dscp:       0
        nw_dst:        "169.254.207.222"
  ▾ 1:
    ▾ actions:
        0:             "SET_QUEUE:0"
        1:             "GOTO_TABLE:1"
      idle_timeout:    0
      cookie:          2
      packet_count:    0
      hard_timeout:    0
      byte_count:      0
      duration_sec:    720
      duration_nsec:   235000000
      priority:        1
      length:          96
      flags:           0
      table_id:        0
```

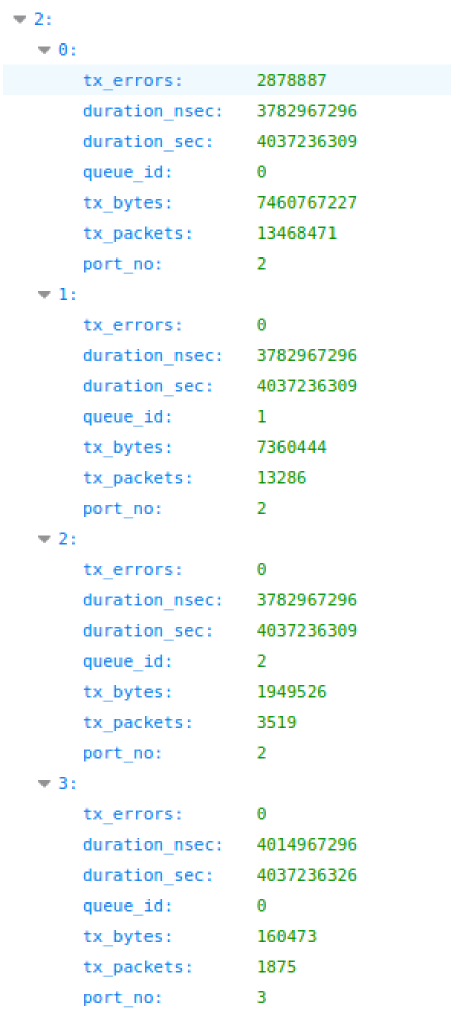
Figure 3.9: On the right some flow stats and on the left port stats.

and `http://localhost:8080/stats/flow/<dpid>`; the value of < `dpid` > in this case is equal to 2 because it represents the Datapath ID. The first gives ports stats, figure 3.9 on the left, and the second gives flows stats, figure 3.9 on the right.

In the figure 3.9 (left) all attributes of each port (red circle) represent statistics of that port; these are three and corresponds to eth0, eth1 and eth2. Some attributes are: `port_no` (port number), `rx_packets` (number of received packets), `tx_packets` (number of transmitted packets), `collisions` (number of collisions), ...

In the figure 3.9 (right) all attributes of each flow (green circle) represent features of that flow, in the figure are represented only two flows but there are many other of them. For example some attributes are: `actions` (instruction set), `lenght` (length of this entry), `priority` (priority of the entry), `match` (fields to match), ...

Another important call is `http://localhost:8080/stats/queue/<dpid>` where



```
▾ 2:
  ▾ 0:
      tx_errors:        2878887
      duration_nsec:    3782967296
      duration_sec:     4037236309
      queue_id:         0
      tx_bytes:         7460767227
      tx_packets:       13468471
      port_no:          2
  ▾ 1:
      tx_errors:        0
      duration_nsec:    3782967296
      duration_sec:     4037236309
      queue_id:         1
      tx_bytes:         7360444
      tx_packets:       13286
      port_no:          2
  ▾ 2:
      tx_errors:        0
      duration_nsec:    3782967296
      duration_sec:     4037236309
      queue_id:         2
      tx_bytes:         1949526
      tx_packets:       3519
      port_no:          2
  ▾ 3:
      tx_errors:        0
      duration_nsec:    4014967296
      duration_sec:     4037236326
      queue_id:         0
      tx_bytes:         160473
      tx_packets:       1875
      port_no:          3
```

Figure 3.10: Queues stats.

queues stats have been obtained. In the figure 3.10 there are four queues, indeed, three queues have been set for the eth2 port and one for the eth1 one.

3.3 Distributed Internet Traffic Generator

D-ITG (Distributed Internet Traffic Generator) is a platform capable to produce traffic by accurately replicating the workload of current Internet applications, see manual [27]. D-ITG supports both IPv4 and IPv6 traffic generation and it is capable to generate traffic at network, transport, and application layer. The traffic generator is composed by five main modules connected to each other, as reported in figure 3.11. The main modules of D-ITG are ITGSend and ITGRecv. ITGSend

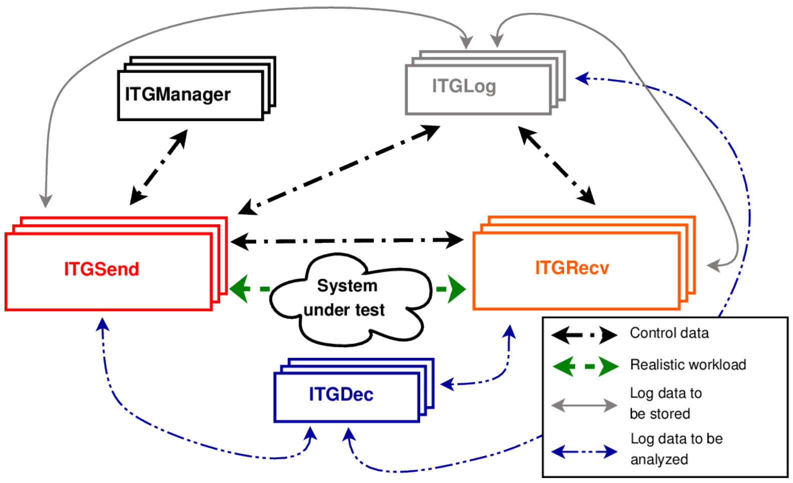


Figure 3.11: Architecture of D-ITG. Fonte: [27]

is the component responsible for generating traffic toward ITGRecv, it can send multiple parallel traffic flows toward multiple ITGRecv instances. The ITGRecv component is responsible for receiving multiple parallel traffic flows generated by one or more ITGSend instances. Between each couple of ITGSend and ITGRecv components there is a signaling channel that control the generation of all the traffic flows between them. ITGSend and ITGRecv can optionally produce log files containing information about every sent and received packet. These logs can be saved locally or sent, through the network, to the ITGLog component. The ITGManager component is responsible for controlling communications between sender and receiver while ITGDec component analyzes the log files in order to extract performance metrics related to the traffic flows.

3.3.1 ITGSend: Sender Component of the D-ITG Platform

The ITGSend component is responsible for generating traffic flows and can work in three different modes:

- **Single flow Mode**: read the configuration of the single traffic flow to generate toward a single ITGRecv instance from the command line.

- **Multiple flow Mode**: read the configuration of multiple traffic flows to generate toward one or more ITGRecv instances from a script file. Each line in the script represents a traffic flow, which includes a set of command line, like in the Single flow Mode.

- **Daemon**: act as daemons listening on a UDP socket and can be remotely controlled through the D-ITG API.

Syntax

The syntax of D-ITGSend depends on the mode used:

- **Single flow Mode**: reads the single traffic flow to generate from the command line.

```
$ ./ITGSend [log_opts] [sig_opts] [flow_opts] [misc_opts]
[ [idt_opts] [ps_opts] | [app_opts]
```

- **Multiple flow Mode**: reads the traffic flows to generate from a script file

```
$ ./ITGSend <script_file> [log_opts]
```

- **Daemon**: runs as a daemon to be remotely controlled using the ITGapi

```
$ ./ITGSend -Q [log_opts]
```

Options

In the syntax of the Single flow Mode there are multiple types of options: log options, signaling options, flow options, misc options, inter-departure time options, packet size options and application layer options. The options used, 3.3.3, are in flow options, some of these are in the table 3.4.

-a	set the destination address
-t	set the generation duration in ms (default: 10000 ms)
-z	set the number of packets to generate
-k	set the number of KBytes to generate
-d	start the generation after the specified delay in ms
-b	set the Differentiated Services field for tests with different QoS (default: 0)
-f	set the IP Time To Live
-rp	set the destionation port
-sp	set the source port

Table 3.4: Flow options of ITGSend.

3.3.2 ITGRecv: Receiver Component of the D-ITG Platform

The ITGRecv component is responsible for receiving multiple parallel traffic flows generated by one or more ITGSend instances. The ITGRecv module works always in daemon mode. It creates a new thread every time a request is received from the network. Between each couple of ITGSend and ITGRecv components there is a signaling channel that control the generation of all the traffic flows between them: the TSP (Traffic Specification Protocol) protocol. In figure 3.12 there are both data channel and the signaling channel.

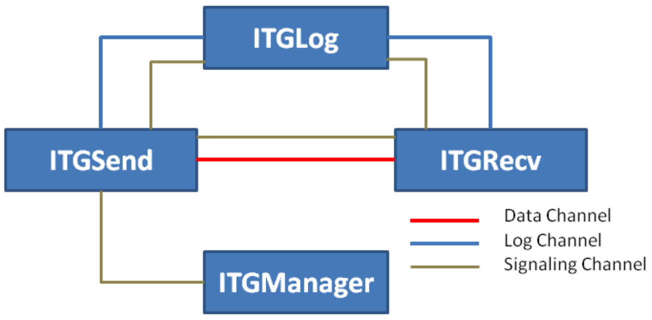


Figure 3.12: Architecture of D-ITG 2

Syntax and Options

The syntax of D-ITGRecv is easier than D-ITGSend:

```
./ITGRecv [options]
```

Some options are in the table 3.5, even if they have not been used in 3.3.3.

-P	enable thread high priority
-a	bind data channels to a specific address
-Sp	signaling channel port number
-l	enable logging to file
-Si	bind signaling channels to a specific network interface

Table 3.5: Options of ITGRecv.

3.3.3 Implementation of D-ITG

Each host has a D-ITGSend and a D-ITGRecv. The D-ITGSend of an host comunicate with the D-ITGRecv of the other host. For the implementation of the D-ITG a python script (`Traffico_ HdwReale.py`) has been used which is the same for each host. This script has been run simultaneously with the following command.

```
sudo python Traffico_HdwReale.py
```

This consists of two functions, one for the D-ITGSend (`f1`) and one for the D-ITGRecv (`f2`), that run as two threads and so they are executed concurrently. The idea, however, is to start the D-ITGRecv before the D-ITGSend, in this way the receiver is already ready to receive the data from the sender. To make this, in the function `f1`, after importing the CSV data (using Pandas[3] library), the D-ITGSend waits some minutes before generating traffic. In this way, the function `f2` certainly start before `f1`, because the latter one, as it has been explained before, have to wait some minutes. The D-ITGRecv starts in this simple way (in the function `f2`):

```
ITGRecv
```

After waiting a few minutes starts D-ITGSend (in the function `f1`) calling the function `createCmd_2` in figure 3.13. The syntax of this function is similar to

```
def createCmd_2(dst, port, tos, nPkts, avg, protocol=DEFAULT_P, ps_dim=DEFAULT_PS):
    com = 'ITGSend -a ' + dst + ' -rp ' + str(port) + ' -b ' + tos + ' ' + choice_i +
    ' ' + str(avg) + ' ' + choice_s + ' ' + ps_dim + ' -t ' + str(TIME_MS-10000)+ ' &'
    return com
```

Figure 3.13: Definition of the function for the D-ITGSend.

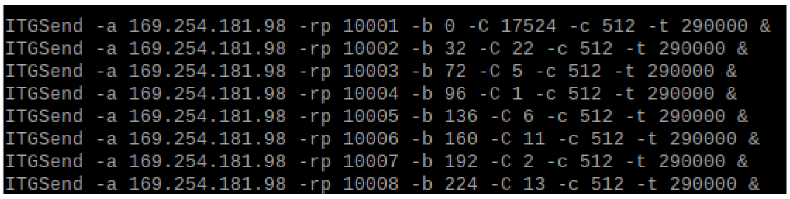
that explained in the subsection 3.3.1 and the options used are in table 3.6. Other

[3]Pandas is the most popular python library that is used for data analysis. Pandas stands for "Python Data Analysis Library".

-a	set the destination address
-rp	set the destionation port
-b	set the Differentiated Services
-t	set the generation duration in ms

Table 3.6: Options of ITGSend used

parameters, such as `ps_dim` represent the packet size (512 bytes as default setting). In figure 3.14 there are the values of the figure 3.13 in an explicit form.



```
ITGSend -a 169.254.181.98 -rp 10001 -b 0 -C 17524 -c 512 -t 290000 &
ITGSend -a 169.254.181.98 -rp 10002 -b 32 -C 22 -c 512 -t 290000 &
ITGSend -a 169.254.181.98 -rp 10003 -b 72 -C 5 -c 512 -t 290000 &
ITGSend -a 169.254.181.98 -rp 10004 -b 96 -C 1 -c 512 -t 290000 &
ITGSend -a 169.254.181.98 -rp 10005 -b 136 -C 6 -c 512 -t 290000 &
ITGSend -a 169.254.181.98 -rp 10006 -b 160 -C 11 -c 512 -t 290000 &
ITGSend -a 169.254.181.98 -rp 10007 -b 192 -C 2 -c 512 -t 290000 &
ITGSend -a 169.254.181.98 -rp 10008 -b 224 -C 13 -c 512 -t 290000 &
```

Figure 3.14: Start ITGSend.

Really the traffic is generated only by the `HOST_DX` to the `HOST_SX` because, in the peak periods, the raspberry-switch is not able to process all the amount of data. In this way, the `HOST_SX` starts only the D-ITGRecv while the `HOST_DX` starts both D-ITGRecv and D-ITGSend.

The variables $d(k) \in \mathbb{R}^8$ of the dynamic model are associated with the D-ITG generator, as seen in the chapter 2, and correspond to the disturbance variables i.e. those that cannot be modified by the control action. A schema of the D-ITG used is in the figure 3.15, where the `HOST_DX` sends traffic to the `HOST_SX` and the controller retrieves the statistics of the packets.

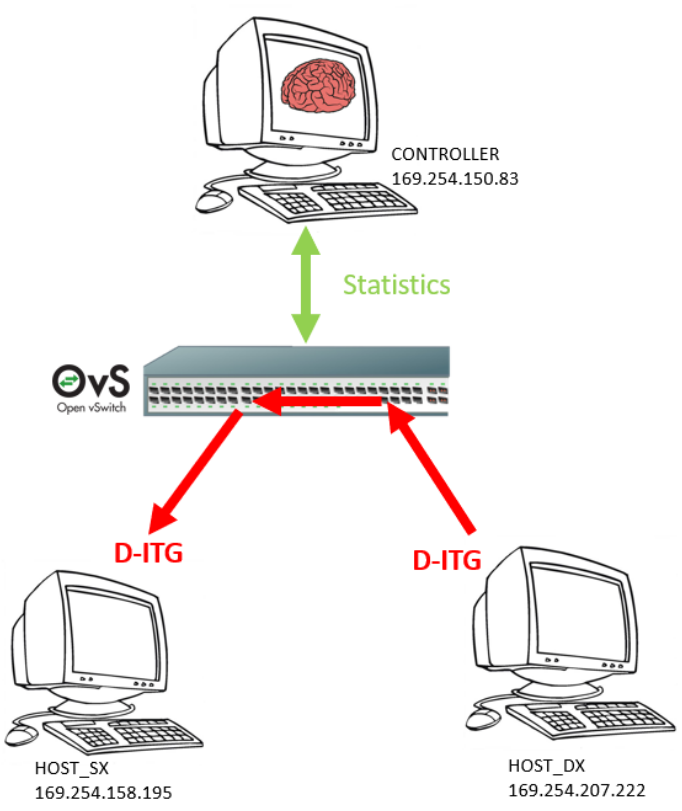


Figure 3.15: Schema D-ITG.

4 Conclusions

In this thesis a Software-Defined Networking (SDN) network using real devices has been created. A SDN network has been used because it is more dynamic, manageable, flexible, and adaptable than the traditional networks. The last ones are complex, decentralized, inable to scale and vendor-dependent. In this way, a SDN controller has a global view of the network topology and so it can configure, supervise and retrieve informations of each SDN device (e.g. switch). The SDN controller communicates with other devices through Application Programming Interfaces (APIs). The most famous communication protocol between the controller and the switch is OpenFlow (OF).
The built SDN architecture consists of an OpenFlow switch, a SDN controller (RYU Controller) and two hosts. The task of the hosts is to generate traffic, for this reason the D-ITG generator has been studied which is a platform capable to produce traffic by accurately replicating the workload of current Internet applications. The RYU controller has also been examined, in particular the `REST_APIs` provided by RYU, a very useful way to obtain statistics on the switch. Finally, the OpenFlow protocol has been analyzed, in particular the OpenFlow ports, flow tables, OpenFlow channel and messages used. On the ports of the OpenFlow switch queues have been set, differentiated according to the Differentiated Service Code Point (DSCP) number in the following way:

- **Default Queue**: DSCP values equal to 0, 1 and 3;
- **Premium Queue**: DSCP values equal to 2, 4, 6 and 7;
- **Gold Queue**: DSCP value equal to 5.

The packets in the Default Queue are routed with maximum rate of 20 *MB/s*, the packets in the Premium Queue are routed with maximum rate of 80 *MB/s* and the packets in the Gold Queue are routed with maximum rate of 80 *MB/s*.
Queues have been set on the switch in order to apply a predictive control strategy on the bandwidth to allocate to the queues through Machine Learning techniques. For this reason some of these techniques have been studied in particular Regression Tree and Random Forests. A predictive model of the switch behaviour is fundamental in order to apply advanced control techniques like Model Predictive Control (MPC). In the analyzed case, the optimization problem is a quadratic optimization problem i.e. a problem of optimizing (minimizing or maximizing) a quadratic function of several variables subject to linear constraints on these variables. A quadratic optimization problem can be solved using Quadratic Programming (QP) solvers; and so even if the function is non-linear it can be solved with low computational costs and reaching a global minimum.
In future works, it could be very interesting to compare the MPC using RT and RF prediction models with non linear ones, for example Neural Networks (NNs).

A
Appendice A

This appendix shows the python script `real.py`. In this script, an existing python code (`Controller_commands.py`) has been imported. Some functions of it have been used which will be described later. The script starts defining the bandwidth of the queues, explained also in the previous chapters:

- **Default Queue**: works at 20% of the total max rate
- **Premium Queue**: works at 80% of the total max rate
- **Gold Queue**: works at 100% of the total max rate

The total max rate corresponds to $100 Mb/s$ converted into b/s. After defining the bandwidth of the queues and obtaining the datapath ID, the functions of `Controller_commands.py` has been used in the following order:

- `switch_ports_name(datapath)`. This function returns the name of the switch ports i.e. eth0, eth1 and eth2.
- `ovsdb_addr(datapath)`. This function is important to access to OVSDB. It corresponds to this call:

```
curl -X PUT -d '"tcp:169.254.153.160:6632"'
    http://localhost:8080/v1.0/conf/
        switches/0000000000000002/ovsdb_addr
```

- `set_queue`(`datapath`, `port_id`, `max_rate`, `queue_rate_list`). This function is important for setting queues, in the first function (`funz_eth1`()) corresponds to:

```
curl -X POST -d '{"port_name":"eth1",
    "type":"linux-htb", "max_rate":"100000000",
    "queues":[{"max_rate":"100000000"}]}'
    http://localhost:8080/qos/queue/0000000000000002
```

  while in the second (`funz_eth2`()) to:

```
curl -X POST -d '{"port_name":"eth2",
    "type":"linux-htb", "max_rate":"100000000",
    "queues":[{"max_rate":"20000000"},
    {"max_rate":"8000000"}, {"min_rate": "100000000"}]}'
    http://localhost:8080/qos/queue/0000000000000002
```

- `set_Queue0`(`datapath`, `port_number`, `IP_flag`, `IP_dst`). This function is used in `funz_eth1`() to set the rules to forward packets according to the DSCP numbers. For eth1 all packets are sent in Queue 0, the only existing queue, for this reason the function is different from `funz_eth2`().

- `set_Telecom_queue`(`datapath`, `port_number`, `IP_flag`, `IP_dst`). This function is used in `funz_eth2`() to set the rules to forward packets according to the table shown in 3.2.

Two different functions (`funz_eth1`(),`funz_eth2`()) have been written instead of one, because the call, `http://localhost:8080/qos/queue/0000000000000002`, explained in 3.1.3, visualizes only the last queue set. For this reason, wanting to have the statistics on the queues (`funz_eth2`()), function `funz_eth1`() has been run before function `funz_eth2`().

The code is:

```
from subprocess import call
import threading
import subprocess
import random
import os
import time
import datetime
import json
import sys
import ditg

from Controller_commands import *

time.sleep(1)
max_rate_queue=100
max_rate_queue=max_rate_queue*1000000
Default=str(max_rate_queue*20/100)
Premium=str(max_rate_queue*80/100)
Gold=str(max_rate_queue*100/100)

def funz_eth1():
        NET = get_switchis()
        if NET != "NO NET" and NET!="[,]":
                i=1
                while i<NET.find("]"):
                        mom_NET=NET[i:]
                        datapath=NET[i:i+mom_NET.find(",")]
                        i=i+mom_NET.find(",")+2
                        port_id = switch_ports_name(datapath)
                        time.sleep(0.2)
                        ovsdb_addr(datapath)
                        IP_Flag=True
                        for index in range(0,len(port_id)):
                                port =
                                    port_id[index][port_id[index].find("h")+1:]
                                if port_id[index]=="eth1":
                                        set_queue(datapath,
                                            port_id[index],
                                            str(max_rate_queue),
                                            "{\"max_rate\":
                                            \""+str(max_rate_queue)+"\"}")
                                        IP_Destination="169.254.207.222"
                                        set_Queue0(datapath, port,
                                            IP_Flag, IP_Destination)
                                else:
                                        time.sleep(0.1)

def funz_eth2():
        NET = get_switchis()
        if NET != "NO NET" and NET!="[,]":
                i=1
                while i<NET.find("]"):
                        mom_NET=NET[i:]
                        datapath=NET[i:i+mom_NET.find(",")]
                        i=i+mom_NET.find(",")+2
                        port_id = switch_ports_name(datapath)
                        time.sleep(0.2)
                        ovsdb_addr(datapath)
                        IP_Flag=True
                        for index in range(0,len(port_id)):
                                port =
                                    port_id[index][port_id[index].find("h")+1:]
                                if port_id[index]=="eth2":
                                        set_queue(datapath,
                                            port_id[index],
                                            str(max_rate_queue),
                                            "{\"max_rate\":
                                            \""+Default+"\"},
                                            {\"max_rate\":
                                            \""+Premium+"\"},
                                            {\"min_rate\":
                                            \""+Gold+"\"}")
                                        IP_Destination="169.254.181.98"
                                        set_Telecom_queue(datapath,
                                            port, IP_Flag,
                                            IP_Destination)
                                else:
                                        time.sleep(0.1)


funz_eth1()
funz_eth2()
```

Acknowledgements

Non volevo festeggiare ma volevo festeggiare. Si è vero da un lato ho sempre ripetuto che la festa alla laurea magistrale non la volevo fare, non mi andava di organizzare il tutto: la festa, dove farla, chi invitare, i vestiti ma dall'altro lato sapevo che volevo farla. Putroppo un qualcosa di più grande di me ha deciso per me e per questo motivo tra Skype, Teams, Whatsapp, Instagram sto festeggiando oggi la laurea in un modo un po' più alternativo. Volevo dirvi, però, che anche se oggi non siete qui con me, vi sento comunque molto vicini.

Ringrazio prima di tutti i miei genitori, che mi supportano e sopportano sempre, che "tanto come scegli, scegli bene" e che soprattutto hanno sempre creduto in me.

Grazie a mia sorella, perchè ripetere su Teams era un modo un po' più per vederci piuttosto che ripetere, grazie per essere una amica.

Grazie alla mia zia preferita, sempre presente e sempre disponibile e grazie per tenerci così tanto alla mia laurea.

Grazie alla mia nonna italiana, perchè nonostante non si ricorda niente la mancetta la domenica non se la scorda mai.

Grazie alla mia nonna francese, perchè è sempre e comunque presente in ogni chiamata, in ogni "castagna".

Grazie alla mia amica di sempre, perchè passano gli anni, aumenta la distanza ma la nostra amicizia rimane sempre la stessa.

Grazie a te, per le avventure il weekend, per le risate, per i Lindor nascosti, per i litigi sul cibo o sulla corda data male, perchè ci basta un Berlingò e una corda per andare dove ci pare.

Grazie alla mia compagna di birrette della domenica, che ultimamente stavo solando un po' ma che finito tutto questo so che ci rifaremo.

Grazie al gruppo "Cuori/Cuormix" e agli amici di Rocca, per le "vasche", per le

partite a biliardino, perchè senza di loro il mese di Agosto alla Rocca non sarebbe lo stesso.
Grazie ai miei compagni di università, per avermi rallegrato tutte le giornate universitarie e per aver sopportato le mie lamentele giornaliere.
Grazie al mio correlatore, per avermi aiutato e capito.
Grazie a lei professore, per i preziosi consigli, per la disponibilità e per la pazienza dimostratami.

References

[1] Open Networking Foundation. “Software-defined networking: the new norm for networks”. In: *ONF white paper* (2012).

[2] Fei Hu, Qi Hao, and Ke Bao. “A survey on software-defined network and openflow: From concept to implementation”. In: *IEEE Communications Surveys and Tutorials* 16.4 (2014), pp. 2181–2206.

[3] Nick McKeown et al. “OpenFlow: enabling innovation in campus networks”. In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[4] Pedro Amaral et al. “Machine Learning in Software Defined Networks: Data Collection and Traffic Classification”. In: ().

[5] Leo Breiman. *Classification and regression trees.* Routledge, 2017.

[6] Leo Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32.

[7] Ján Drgoňa et al. “Approximate model predictive building control via machine learning”. In: *Applied Energy* 218 (2018), pp. 199–216.

[8] Carlos E Garcia, David M Prett, and Manfred Morari. “Model predictive control: theory and practice—a survey”. In: *Automatica* 25.3 (1989), pp. 335–348.

[9] Alberto Bemporad, Francesco Borrelli, Manfred Morari, et al. “Model predictive control based on linear programming˜ the explicit solution”. In: *IEEE transactions on automatic control* 47.12 (2002), pp. 1974–1985.

[10] Francesco Smarra et al. “Data-driven switched affine modeling for model predictive control”. In: *IFAC-PapersOnLine* 51.16 (2018), pp. 199–204.

[11] Yong Li and Min Chen. “Software-defined network function virtualization: A survey”. In: *IEEE Access* 3 (2015), pp. 2542–2553.

[12] OpenFlow Switch Specification. *1.4. 0.* 2013.

[13] Martin Casado et al. “Ethane: Taking control of the enterprise”. In: *ACM SIGCOMM computer communication review* 37.4 (2007), pp. 1–12.

[14] *UDOO X86.* 2019. URL: `https://www.udoo.org/`.

[15] Kathleen Nichols et al. *RFC2474: Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers.* 1998.

[16] Jozef Babiarz, Kwok Chan, and Fred Baker. “Configuration guidelines for DiffServ service classes”. In: *Network Working Group* (2006).

[17] A. M. Langellotti et al. *Notiziario Tecnico Telecom Italia.* Tech. rep. Telecom Italia, 2004.

[18] *RYU controller.* 2019. URL: `http://www.osrg.github.io/ryu`.

[19] Simon Horman. “An Introduction to Open vSwitch”. In: *LinuxCon Japan, Yokohama, Horms Solutions Ltd., Tokyo* (2011).

[20] *Open vSwitch.* 2019. URL: `https://www.openvswitch.org/`.

[21] *Converting a Raspberry Pi to a OpenFlow Switch.* 2017. URL: `https://sumitrokgp.wordpress.com/2017/05/18/converting-a-raspberry-pi-to-a-openflow-switch/`.

[22] *Common Configuration Issues.* URL: `http://docs.openvswitch.org/en/latest/faq/issues/`.

[23] *QoS Ryubook 1.0 documentation.* 2019. URL: `https://osrg.github.io/ryu-book/en/html/rest_qos.html`.

[24] Americas Headquarters. “Cisco Nexus 1000V Quality of Service Configuration Guide, Release 4.0(4)SV1(3); Chapter: DSCP and Precedence Values”. In: (2009).

[25] *IP Precedence and DSCP Values.* URL: `https://networklessons.com/quality-of-service/ip-precedence-dscp-values`.

[26] *ryu.app.ofctl rest.* 2019. URL: `https://ryu.readthedocs.io/en/latest/app/ofctl_rest.html`.

[27] Alessio Botta et al. “D-ITG 2.8. 1 Manual”. In: *Computer for Interaction and Communications (COMICS) Group* (2013), pp. 3–6.