TRABAJO DE FIN DE GRADO

Título:	Desarrollo de un Sistema de Diagnóstico de fallos basado en Redes Bayesianas para Redes Definidas por Software
Título (inglés):	Development of a Fault Diagnosis system based on Bayesian Networks for Software Defined Networks
Autor:	Álvaro Martínez Carmena
Tutor:	Álvaro Carrera Barroso
Departamento:	Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	Juan Quemada Vives
Vocal:	Joaquín Salvachúa Rodríguez
Secretario:	Gabriel Huecas Fernández-Toribio
Suplente:	Santiago Pavón Gómez

FECHA DE LECTURA: 26 de Enero de 2017

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

TRABAJO DE FIN DE GRADO

DEVELOPMENT OF A FAULT DIAGNOSIS SYSTEM BASED ON BAYESIAN NETWORKS FOR SOFTWARE DEFINED NETWORKS

Álvaro Martínez Carmena

Enero de 2017

Resumen

Esta memoria es el resultado del trabajo de fin de grado, cuyo objetivo ha sido realizar un sistema para el diagnóstico de fallos de red basado en redes bayesianas y apoyado en una infraestructura de redes definidas por software (SDN).

Este sistema proporciona, forzando diferentes situaciones en una red simulada, un conjunto de posibles fallos y la probabilidad de que estén ocurriendo. Esto es posible gracias a distintas herramientas como OpenFlow, POX, Mininet, o SMILE, que permiten simular una red, sacar información de ella para procesarla y determinar mediante un modelo bayesiano el resultado del diagnóstico.

Para poder llevar esto acabo se ha definido la topología, con sus recursos y reglas, y simulado un escenario de consumo de streaming en el cual se han generado los fallos. Posteriormente se han recogido los datos de la red a través de un controlador y procesado por medio de un módulo de monitorización, durante las diferentes situaciones generadas.

Los datos recolectados de la red simulada han sido utilizados para alimentar un algoritmo de aprendizaje, permitiendo así generar un modelo de diagnóstico en una red bayesiana, que una vez integrado en el sistema ofrece la causa de fallo más probable de entre el rango de posibles.

A continuación, se han generado tres modelos de diagnóstico diferentes con el fin de evaluarlos en el escenario en cuestión. Por último, se proponen diferentes líneas de trabajo futuro para mejorar el sistema propuesto.

Palabras clave: SDN, OpenFlow, Redes Bayesianas, Streaming, Diagnóstico de fallos, Servidores, Clientes, Enlaces, Python, Mininet

Abstract

This document is the result of the final degree project, whose objective is to develop a diagnosis fault system based on bayesian networks for Software Defined Networks (SDN).

This system provides, provoking some situations in the simulated network, a set of possible faults and their probability of happening. This is possible thanks to different technologies such as OpenFlow, Mininet, POX, or SMILE, that allows to simulate a network and take data in order to process and obtain by means of a bayesian network the diagnosis result.

To carry this out, the topology has been defined, with a set of networking rules and system resources, and a streaming service has been simulated, in which we have generated the faults. After that, the network data have been collected thanks to the controller, and processed by a monitoring module, during the different generated situations.

The collected data of the simulated network have been used to feed a learning algorithm, allowing in this way to generate a diagnosis model in a bayesian network, that after being integrated in the system, it offers the most likely root cause failure of the possible range.

Then, three diagnosis models have been generated in order to evaluate them in the scenario in question. Finally, some possible future lines of work to improve the diagnosis system are presented.

Keywords: SDN, OpenFlow, Bayesian Networks, Streaming, Fault diagnosis, Servers, Hosts, Links, Python, Mininet

Agradecimientos

Me gustaría agradecer en estas líneas a la gente que me ha ayudado y apoyado durante mi realización del trabajo de fin de grado.

A mis padres y mis hermanos, por su interes y apoyo durante la realización del proyecto.

A mis amigos y compañeros de la *Escuela*, y de fuera de ella, que siempre me han animado en mi trabajo. En especial a mi compañero Oscar Araque que nos facilitó su proyecto sobre el que parte este trabajo, además de prestarse siempre a cualquier duda cuando lo necesitábamos.

Al Grupo de Sistemas Inteligentes en general y en particular a: Carlos, por ofrecerme la posibilidad de realizar el trabajo con ellos; y a mi tutor, Álvaro, que además de llevar el seguimiento de mi trabajo y proporcionarme realimentación, siempre ha estado dispuesto a concertar reuniones ya fuese para revisar avances o tratar de resolver problemas derivados del TFG.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

This chapter provides an introduction to the purpose and main objectives of this project. It also provides an overview of the context of Software Defined Networks, and the description of the project structure, defining the content of the remaining chapters.

1.1 Context

The industry in SDN and OpenFlow has grown in recent years. That is because network management operators can manage devices resources and suit them to the changing needs of the network. Among others, the management capabilities offered by SDNs allows operators to make changes in the network configuration to heal faults or critical situations with less effort than in the classic telecommunication networks.

All of this with the purpose of not having to perform all these actions in a classical network infrastructure; and avoiding the costs, licenses and effort to deal with it. But the adoption of SDN is not only important for applying changes in a physical network, but also because the impact it may have on the Telecommunications market, Bell Labs company has estimated that the replacement of classical networks with Software Defined Networks (SDNs) could save a 25.000 million of euros in operative costs.

1.2 Project description

This thesis aims at developing a Software Defined Network (SDN) scenario based on OpenFlow standard. In that network scenario, fault diagnosis will be performed during the consumption of different services, such as video or audio streaming.

This project will consist of the following main phases:

Definition of Topology and service to analyze: In this phase, we will define the topology and service in which diagnosis cases will be based. The network topology will be defined with a set of networking rules and system resources. Then, that network will be implemented using a network simulator based on OpenFlow standard.

Data collection: During this phase, different statistics from the network will be collected for analysis through a network controller. Different situations will be provoked to collect information for regular network operation and for specific faults.

Diagnosis model training: This phase will produce as output a diagnosis model based on statistics collected during the previous phase. This training phase will use the processed statistics collected from the network to generate a Bayesian network, which will use network variables as inputs and will offer the most probable cause of fault among the set of faults simulated during the previous phase.

Application of Diagnosis model during the network operation: The model generated in the previous stage will be integrated with the network controller used during the earlier phases to collect information from the network in order to automate a fault diagnosis process when an anomaly or symptom is detected. According to the patterns learnt in the previous stage, it will generate a diagnosis forecast in real time.

1.3 Project goals

The main goals of this project are:

- Learning SDN and OpenFlow basic concepts.
- Defining and implementing a streaming scenario.
- Learning the basis of the management and the monitoring of SDN using the developed scenario as benchmark.
- Learning the use of bayesian networks to reason about the fault root cause to diagnose the network during its operation.

1.4 Structure of this Project

In this section we will provide a brief overview of all the chapters of the thesis. It has been structured as follows:

Chapter 1 provides a summary, structure and main goals which will be approached in this project. It also provides an overview of the main term effects of using the SDN technology today.

Chapter 2 contains an overview of the different enabling technologies on which the development of the project will rely.

Chapter 3 describes the architecture of the fault diagnosis system, the main modules that make it up, and the process from the data collection to the obtained diagnosis.

Chapter 4 describes the different equipment and the distribution of the simulated network, as well as its rules. It also explains the different generated faults, how to generate them, and the implementation of all the modules presented in Chapter 3.

Chapter 5 describes the evaluation of the system. Firstly, three different diagnosis models are presented, and then, it explains the experimentation phases of the system: the generation of each model and the evaluation of them.

Chapter 6 presents the conclusions, achieved goals and the found difficulties during the realization of the project. Some ideas for future development are also given, in order to improve or continue this work.

Finally, the entire implementation of the code is available on GitHub:

https://github.com/gsi-upm/sdn-diagnosis

CHAPTER 2

Enabling Technologies

This chapter introduces which technologies have made possible this project. Initially, it will be necessary the OpenFlow technology, that can be divided into switches and controller. The library used to handle the controller is POX. And finally it will be also necessary the network emulator Mininet to run the network scenario and Virtualbox to make this possible.

2.1 OpenFlow

Openflow [1] project borns in 2011, founded by companies like Google, Microsoft or Facebook, with the aim of managing networks, their protocols, topology, performance, and different functionalities in order to be applied to real models. All of this by software, without the need to create a real network and the costs involved. This suppose an important improvement in the way the extensive testing are held.

It's possible to check the forwarding plane of the network by means of the controller and switches.

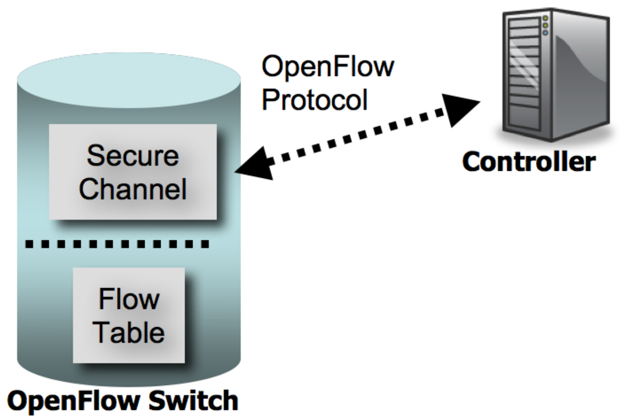

Figure 2.1: OpenFlow management channel

2.1.1 OpenFlow Switch

The openflow switch [2] has two levels (as can be seen in Figure 2.1), data path and control path. The former resides on the switch like in a normal one, and the latter is moved to a separate controller. These high-level routing decissions are transmitted by the openflow protocol, which defines different messages like packet-received, errors, modify-forwarding-table, and stats such as flow stats, port stats or queue stats.

2.1.2 OpenFlow Controller

The Controller [3] uses the OpenFlow protocol to connect and configure the network switches. It is the central point that allows handling the SDN flows and extract the stats and data from switches. There are different controllers that can be used in OpenFlow, such as Beacon, Floodlight, NOX , POX, Ryu, Trema or OpenDayLight.

To this project it will use POX due to the fact that the project in which this work is based is developed in POX, and also because its language is fairly intuitive.

2.2 POX

As indicated above, in Section 2.1.2, POX [4] will be the controller chosen for this project. POX provides a framework for communicating with the switches by OpenFlow protocol. It is implemented on Python [5], which will be benefitial because of the ease and simplicity that python offers when it comes to handle chain characters, dictionary or lists, some of the most used types in this project.

POX has some components, which are additional Python programs that can be invoked when POX is started, such as:

-py: starts the python interpeter

-forwarding.l2learning: makes the OpenFlow switches operate at Layer 2.

-forwarding.l2pairs: like the forwarding.l2learning, the switches will operate at L2, but installing rules based on MAC addresses.

-openflow.spaningtree: this component builds a spanning tree topology, and disables flooding on switch ports that don't belong to the tree, in order to avoid loops.

-openflow.webcore: starts a web server withing the POX process.

-messenger: provides an interface to interact with external process by means of JSON messages.

-misc.arpresponder: to answer ARP request.

-openflow.discovery: send LLDP messages from the OpenFlow switches in order to discover the network topology.

-openflow.keepalive: allows POX to send periodic echo request to the switches.

But one of the main purposes of using POX is for developing OpenFlow control applications, acting as a controller for an OpenFlow switch. In this context there are two ways of communication with the switch:

2.2.1 Asynchronous events

Events raised without a request from the controller, these messages are sent automatically by the switches to the controller when something happens in the network.

ConnectionUp	Establishment of a new control channel with a switch
ConnectionDown	Connection to a switch has been terminated, either it has been closed or because the switch has restarted
PortStatus	A switch port has been modified
FlowRemoved	A table entry is removed on the switch either because timeout or explicit deletion
PacketIn	A packet arrives at a switch port
ErrorIn	An OpenFlow error has ocurred on a switch

Table 2.1: Asynchronous events

2.2.2 Synchronous events

Events raised because a request from the controller, these messages are sent each time the controller requests it. Usually after a certain time interval.

FlowStatsReceived	Information about individual flow entries in a switch port, level 3
AggregateFlowStatsReceived	Information about multiple flows entries, level 3
TableStatsReceived	Information about active entries or number of packets looked up in tables
PortStatsReceived	Information about the ports, transmitted and received packets, error or dropped packets, level 2
QueueStatsReceived	Information about the queues, transmitted packets from the queue, time the queue has been alive,...

Table 2.2: Synchronous events

2.3 Open vSwitch

Open vSwitch [6] is an Openflow implementation designed to be used as a virtual multilayer switch. This allows to make an effective network automation, besides having standard management interfaces and protocols such as NetFlow, sFlow or SPAN.

2.3.1 Open vSwitch Components

The main components used in the project are:

- ovs-dpctl: to configure the kernel module switch. This component can create, remove or modify Open vSwitch data-paths.
Some examples of possible configurations:

add-dp dp1: creates a device named dp1

del-dp dp1: removes the indicated device.

add-if dp1 eth1: creates an interface for a particular device, in this case the eth1 on the device dp1.

set-if dp1 eth1 "options": reconfigures the port with the specified options.

del-if dp1 eth1: removes the interface eth1 from the device dp1.

- ovs-vsctl: to consult and update the ovs-vswichd configuration, i.e:

show: Prints a brief overview of the switch database configuration.

list-br: Prints a list with the configured switches/bridges.

list-ports "switch": Prints a list with the configured ports of the indicated switch.

list interface: Prints all the interfaces, their configuration and statistics.

list queue: Prints the configuration of the queues created.

list qos: Shows the quality of service configuration of a port. Records are identified by port name.

- ovs-ofctl: for monitoring the network and get the net-stats in real time, also to add certain rules in routing flows, modify tables or port states.

dump-port "switch" "port": Prints the port stats mentioned in 2.2.2.

dump-flows "switch": Prints the flow stats mentioned in 2.2.2.

dump-aggregate "switch": Prints the aggregate flow stats mentioned in 2.2.2.

dump-table "switch": Prints the table stats mentioned in 2.2.2.

queue-stats "switch" "port" "queue": Prints the queue stats mentioned in 2.2.2. The port and the queue can be selected, if is not selected it prints all the queue stats alive in the ports.

add-flow "switch" "flow": Adds a flow entry to switch table.

mod-port "switch" "port" up/down: Changes the port state of the switch to up or down.

2.4 Mininet

Mininet [7] is a network emulator that allows to create a realistic virtual network, running switches, controllers, hosts and links on a virtual machine.

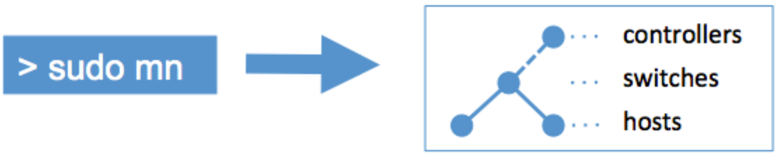

Figure 2.2: Mininet

Mininet hosts run in Linux and the switches support OpenFlow to connect between themselves and with the controller.

This network emulator provides facilities for monitoring, testing, debugging, and other different tasks in order to have a complete network analysis.

To simulate the links and devices configuration, as well as errors or different alterations that are intended to perform in the network, or even to test connectivity between different hosts, it will be necessary some mininet commands [8] shown in the following sections.

2.4.1 Constructor/clean commands

These commands are executed in order to start the emulation or remove the records (before or after the emulation).

- **sudo mn:** starts mininet emulator with a default configuration. Is possible to custom the topology and its configuration assigning different values to the parameters and selecting the number and the distribution of the different switches and hosts. With

the option ”–custom=[path]”
There are also some topologies that can be selected without the need to make your own configuration, adding the option ”–topo=[parameter]”, such as linear, minimal, single, reversed or tree.

- **sudo mn -c** cleans records emulation.

2.4.2 CLI commands

These commands are available once sudo mn has been executed and the emulation has been initiated (Command line interface). Mininet is enabled to execute shell comands on the emulated switches, hosts and servers, this requires to enter the device name before the instruction and its parameters.

- **quit:** ends the emulation.
- **exit:** ends the emulation and close the program.
- **help:** shows some info about the mininet commands.
- **dump:** shows info about the devices, like name, type, IP address and active port.
- **net:** shows all the links and the devices ports corresponding to both ends.
  Ex: h1-eth1:s8-eth3
- **ports:** shows the ethernet ports that a switch has.
- **xterm dev:** opens a terminal for the specified device.
- **iperf host1 host2:** measures the maximum bandwith between two hosts.
- **host1 ping host2:** host1 sends a ICMP message to host2. Several options can be selected, for example, to send a certain number of packets the options would be ”h1 ping -f -c [number] h2”.
- **switch ”switch” start:** starts or initiate a switch that is not running.
- **switch ”switch” stop:** stops the operation of a switch, when the switch is stopped this cannot communicate with the controller.
- **link dev1 dev2 up/down:** enables or disables the link between to network nodes.
- **sh ”command”:** executes an ubuntu shell comand.

- **source "path":** executes mininet commands from a file previously created. It can be introduced relative or absolute paths.

- **px "command":** executes python commands.

2.5 Traffic Control

Traffic Control (tc) [9] is a tool that Linux uses in order to configure the linux kernel packet scheduler. tc has several elements (shapping, scheduling, classifying, dropping, marking), but the only one to be treated and used in the project is the scheduling, associated with the qdisc (queueing discipline) component.

2.5.1 Queuing discipline

Queuing discipline (qdisc) [10] is a scheduler for the output interfaces, when traffic arrives at a device interface it needs to be classified to be treated and sent in the corresponding way.
The default scheduler of linux is FIFO, but it's possible to add or change the disciplines in order to handle the traffic in the desired way. Some kind of disciplines that will be used in this project are:

netem: to emulate or modify traffic egress properties of an interface switch. The modifiable parameters are delay, loss, duplication and re-ordering.
Example of use:
`tc qdisc change dev s2-eth2 parent 5:1 handle 10:` **`netem loss 25`**
In this case the discipline has been changed, the delay will be the indicated in the topology and now a 25 % probability of loss packets has been added.

pfifo: the default discipline when a standard topology has created without specifying parameters, the queueing theory used is the packet first in first out. It has only one parameter and is the limit or buffer of the queue.
`tc qdisc add dev s2-eth2 parent 5:1 handle 10:` **`pfifo limit 1000`**
A pfifo system with a 1000 packets buffer has been added to the s2-eth2 disciplines.

2.6 VirtualBox

Virtualbox [11] is the virtualization software used to install Mininet and deploy the network in the project. It could also have used some Linux distribution or another program, but it has preferred this choice because its easy use and for being a free program.

In the image below can see that it has chosen an image of the virtual machine Mininet, and other different features selected in Virtualbox to run this VM.

Figure 2.3: Virtualbox

2.7 SMILE

Structural Modeling, Inference, and Learning Engine (SMILE) [12] is a C++ library for graphical decision-theoretic methods, such as bayesian networks, influence diagrams and structural equations models. This library can be integrated into user applications like GeNIe (Section 2.8), and wrappes to use it in other programming languages such as Java or .NET.

2.8 GeNIe

Is a software of BayesFusion, LLC, specifically a graphical user interface to SMILE that allows to work with model building and learning. It is a native programme for windows, but it can be used on Mac Os or Linux under Windows emulators.

This program let us to generate the necessary bayesian networks for the diagnosis system. It accepts csv data files as input, and after selecting the alogorithm and some parameters like max parent count, iterations, link probability, class variable or k-fold, the bayesian network will be generated in a xdsl file as an output.

Once the xdsl is created, is possible to validate it introducing another input data file, and it will give some validattion results such as accuracy, confusion matrix, ROC cruve and calibration. It is also possible to interact with the network, changing manually the values of the nodes and getting the probabilities of the class variable values.

We can find more information about its use in [13].

2.9 JPype

JPype [14] is a python library that allows to use Java class libraries in Python through a Java virtual machine, which has to be started in the python program, obviously having Java installed before is required.

In the project it will be necessary because SMILE librarie is not available for Python, but it is for Java (as stated in Section 2.7). Once Jpype is installed and imported, we must indicate the smile.jar path before the creation of the Java virtual machine, in order to use this library during the execution.

CHAPTER 3

Architecture

This chapter describes the architecture of the proposed fault diagnosis system. The main modules which compose it are described. Finally, the complete diagnosis process is presented: since data collection to a final diagnosis result.

3.1 Overview

The system can be divided into three modules: Event handler from POX, statistics processor and bayesian inference module, the two former ones are responsible for collecting, processing and adapting data from a switchs-hosts network and sending it to the latter, the bayesian inference module, so that this learns such behaviour and is able to diagnose possible network failures.
In Figure 3.1, a general overview of the system architecture is shown. Where the colour scale shows the grade of completion of the process.

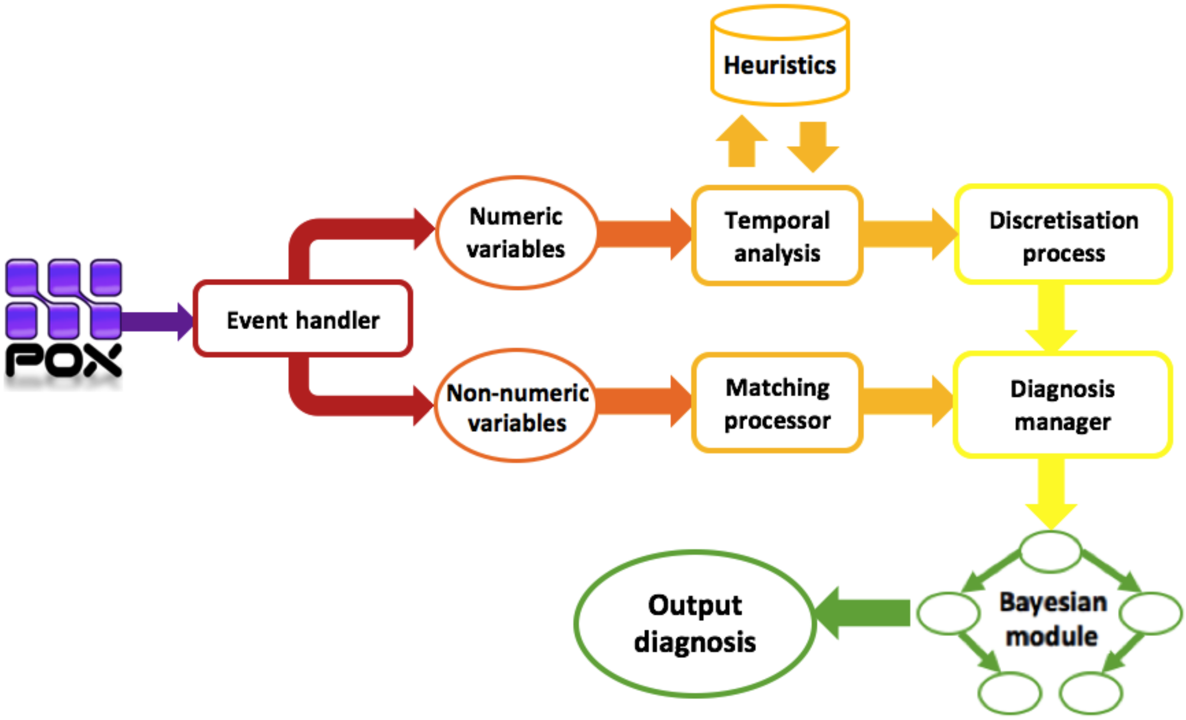

Figure 3.1: Architecture overview

3.2 Event handler from POX

The first module of the architecutre is the event handler, which communicates the diagnosis system with pox controller, which in turn is directly connected with the switches.

The event handler deals two different types of events received from POX controller: asynchronous, these are the ones that happens with a changing frequency, in other words, network situations that automatically produces an event, such as link up or down, stop functioning of a switch or new switch added, as described in Section 2.2.1; and synchronous events, these happens because it is requested from the controller, with a fixed frequency. For example, the sending statistics of ports, tables or queues indicated in Section 2.2.2. When any of the above events happen, the handler receives it thanks to the associated listener, and takes the information of the given event in order to send it to the module responsible for its processing.

3.3 Statistics processor

This module is the most important of the system, since it has to analyze all the received information, discard the one that is not useful, and process the rest, saving some data that can be used later and operating it. All this information must be presented in the appropiate way to the bayesian module, and this is also task of the processing module.

The output data has to be presented in discrete variables. Therefore, depending on whether the information received is non-numeric or if otherwise is numeric, it will have to be processed differently.

In total we will process 7 variables: 4 non-numeric -the state of the 3 ports and the switch state-, and 3 numeric -transmitted, received and lost packet from the switch-.

3.3.1 Non-numeric variables

The non-numeric variables require an easier processing than in the case of the numeric ones. In particular, they indicate the switch ports states and the general state of each switch (true or false). It is necessary to know which states and in what moment have to be collected and associate to the corresponding switch or port.

3.3.2 Numeric variables

The numeric variables go through more stages, first of all, it is necessary to collect only the required information, there are many information that it doesn't need for the analysis, in particular we have to search among the data the number of received and transmitted packets for each switch port.

The stats that arrives from POX are the total stats since the beginning of the simulation, and the objective is to perform an analysis in real time, so it is necessary to make different operations every certain period of time in order to transform those stats measured in the total time into stats measured in a time interval.

The number of lost packets per port is not provided from POX, so we have to calculate it, making difference between the transmitted packets from an output port and the received packets from the input port of the next switch.

All these values must be discretized, associating to each range of values different levels, based on the consistency of the amounts obtained for the transmitted and received packets. And in the case of the lost packets based on the percentage of lost packets over the total transmitted.

Finally, the diagnosis manager will send the data already processed to the bayesian module.

3.4 Bayesian reasoning inference

The last module of the system architecutre is the bayesian reasoning inference. Its objective is, by means of the input variables received from the previous module, be able to reason and give a final diagnosis. It has two stages:

Learning stage: It receives many lines of simulation (one for each time interval measured) with all the variables mentioned above and one extra, the root cause failure or the network situation. With all this data the bayesian module will learn the knowledge of the network and it can be applied after.

Reasoning inference stage: The module receives again the variables, but this time without the root cause failure, and the bayesian network will provide a diagnosis of the network situation and the success probability based on what was learned previously.

CHAPTER 4

Testbed

This chapter describes the different equipment that makes up the simulated network, as well as its connections and rules. It also describes: the different faults that will be applied on the system, how these faults are generated, and how the system is implemented to analyze them.

4.1 Simulated network description

Before implementing our system described in Section 3, we need to define a simulated network topology that serves as a starting point for the testbed. The network to be defined will be formed by: 1 server, 3 hosts and 8 switches interconnected in two rings, whose distribution is shown in the left picture of Figure 4.1.

The server provides a multicast streaming video to the hosts, and the traffic routing through the switches follows the forwarding rules defined before the deployment of the network. In the right picture of Figure 4.1 can be seen the routing from server to hosts selected for this case.
S8 distributes all the traffic between S6 and S7. S6, in turn, forwards 2/3 of the server flows to the hosts 1 and 2, while H1 is provided by S7.

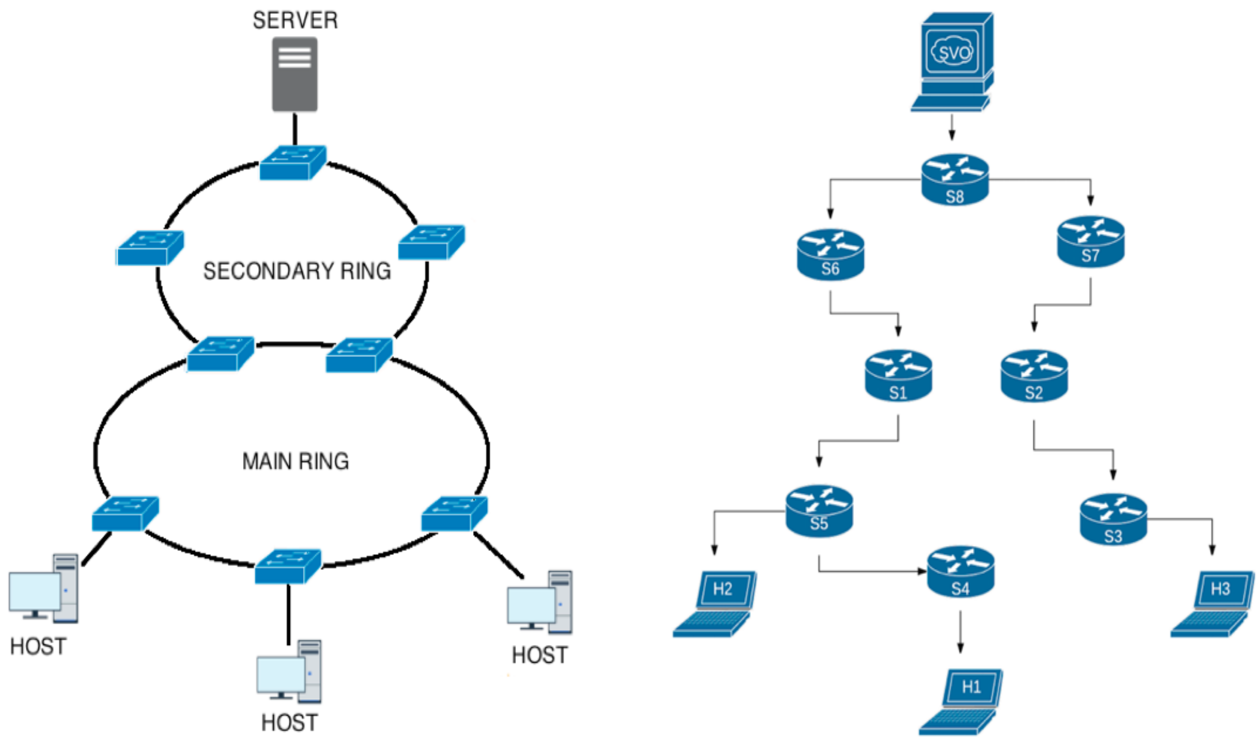

Figure 4.1: Network topology and routing

4.2 Fault types

During the simulation we will force the network to 5 different situations, one of correct or normal functioning, and others with simple faults that will cause different symptons in the clients, such as cut in the video signal or decrease the image quality.

Normal functioning: the network runs propperly and the client displays the video in good conditions.

Server down: the link between server and the switch server is down, there will not be traffic through the network, so the client's symptom will be a frozen video image.

Switch malfunction: some switch of the network is losing packets, and the video quality in the client involved will be reduced and with interruptions.

Switch down: a switch crashes and therefore the traffic and the video signal will be stopped, the symptom is also a frozen video image for the affected client.

Link down: a link betwen two switches falls, so the sympton in the client is the same as in server down and switch down, video playback will be interrupted.

4.3 Simulation manager

Once the topology described in Section 4.1 is deployed,, the simulation manager is in charge of starting the streaming video servers and clients and generating the corresponding faults in the network.

4.3.1 Starting streaming server and clients

To start the video in the server and clients we will use the vlc library that allows to open a video in the different hosts of the network.

For the server is necessary this shell-command: `vlc <video/directory/path>` which opens the video indicated in the path directory with the vlc library.

In the clients: `vlc -q http://10.0.0.4:80/test`, that again, it opens the video with vlc, but now, the video which is running in the URL requested (the IP address of the server). After this, the streaming service is already underway, therefore, the network is in the normal functioning situation.

4.3.2 Generating fault situations

In order to analyze the behaviour of the network switches the simulation manager will generate some faults, and at the same time this situation will be reported. To simulate and report the faults explained in Section 4.2 we have to use the following command lines.

Server down: First, we have to force the ethernet that connects the switch server with the server to drop, this can be done either with a shell ifconfig command or with the mininet CLI commands explained in Section 2.4.2. Then, with a python command a file text called rcf.txt (root cause failure) is opened and written on it the situation that has just been generated (with the purpose of being read by the diagnosis module), "server down" in this case. Finally, with the python function time.sleep it remains in this state as long as necessary for the analysis. In Listing 4.1 we can see the necessary lines.

Listing 4.1: Server down

```
sh ifconfig sX-ethX down
px archivo=open('rcf/file/directory/path','w')
px archivo.write('server_down')
px archivo.close()
py time.sleep(60)
```

Switch malfunction: In this case, the objective is to make a switch port loss packets, for this we can rely on the tc tool of linux, more specifically in the scheduler qdisc described on Section 2.5.

For instance, for a 20% lost rate, we use the code shown in Listing 4.2.

Listing 4.2: Switch malfunction

```
sX tc qdisc change dev sX-ethX parent 5:1 handle 10: netem loss 20
px archivo=open('rcf/file/directory/path','w')
px archivo.write('sX_malfunction')
px archivo.close()
py time.sleep(60)
```

Switch down: Now, the goal is to stop a switch, so we use the mininet CLI command `switch sX stop` , as it shows in Listing 4.3.

Listing 4.3: Switch down

```
switch sX stop
px archivo=open('rcf/file/directory/path'', 'w')
px archivo.write('sX_down')
px archivo.close()
py time.sleep(60)
```

Link down: The process is similar as in 'server down', because it is also about a link down. So we replace the switch and ethernet of the server with the corresponding one, or simply indicate the two switches of the chosen link. Then, we write in the rcf file either "rx_link_down" if the given link is the received link of the switch or "tx_link_down" if is the transmission one. The used commands are In Listing 4.4.

Listing 4.4: Link down

```
link sX sY down
px archivo=open('rcf/file/directory/path','w')
px archivo.write('rx/tx_link_down')
px archivo.close()
py time.sleep(60)
```

4.3.3 Killing streaming service and reverting the fault

As the objective is to perform a continuous analysis of the network, the forced situation will need to be reversed and the streaming service started over. So the streaming process will be killed with command `dev killall vlc`, where dev is the device (sv0 for the server, or h1/h2/h3 for the clients). After that, the command to revert the fault is introduced and also the updating of the rcf file.
An example to kill the streaming service of a 'server down' situation is given in Listing 4.5.

Listing 4.5: Killing service

```
sv0 killall vlc
h1 killall vlc
h2 killall vlc
```

```
.
.
.
hn killall vlc
sh ifconfig sX-ethX up
px archivo=open('/home/mininet/sdn-diagnosis/monitor/rcf.txt', 'w')
px archivo.write('Normal_functioning')
px archivo.close()
```

Now, we could start the streaming service again to deal with another situation of the network.

4.4 Architecture implementation for testbed

The implementation for testbed is what makes the system works, comprises three separated modules, the event handler, the statistics processor and the bayesian module explained in the architecture of the system (Chapter 3). To connect the event handler with the statistics processor is necessary to use a pipeline, this way two different programs or process can access to the same file at the same time, one as a writer (the event handler module) and the other as a reader (statistics processor). The pipeline is created with the command `mkfifo <path/directory/file>` before starting the modules.

4.4.1 Event-handler from POX

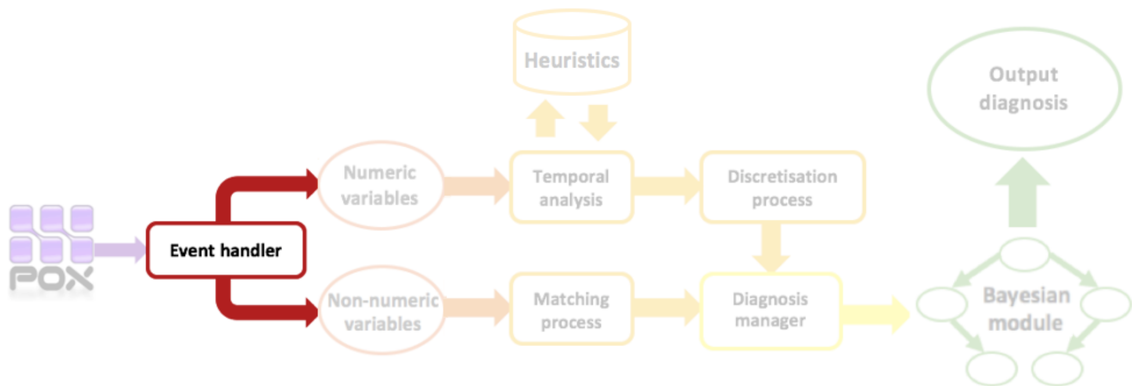

Figure 4.2: Event-handler from POX

Like it was explained in Section 3.2 two types of events (synchronous and asynchrnous) can occur. The first ones happens automatically when something changes in the system,

so we would only have to add a listener to the corresponding kind of event, with the following line: `core.openflow.addListenerByName("TypeOfEvent", _handler)` . Where "TypeOfEvent" is the name of the event, such as ConnectionDown, ConnectionUp or PortStatus, while _handler is the name of the function that handles this event, this function has to be implemented in the most appropiate way to work with this data later.
For example, in Listing 4.6 is shown how we take the information of ConnectionDown event and store it in the pipeline mentioned at the beginning of this section, where _to_pipe is a method that writes the given parameter in the pipe file, like is done in the codes of Section 4.3.2 with normal text files.

Listing 4.6: ConnectionDown handler

```
def _handle_ConnectionDown (event):
log.info("Switch %s has come down.", dpid_to_str(event.dpid))
dpid = dpid_to_str(event.dpid)
data = {'type':"switch_down", "data":{'switch':dpid, 'down': True}}
data = json.dumps(data)
data += '#'
_to_pipe(data)
```

4.4.2 Statistics processor

The process of this module is the longest of the implementation and will have two threads, the main thread will read data from the pipe, and the secondary thread will process and update this data.

4.4.2.1 Reading data from event-handler

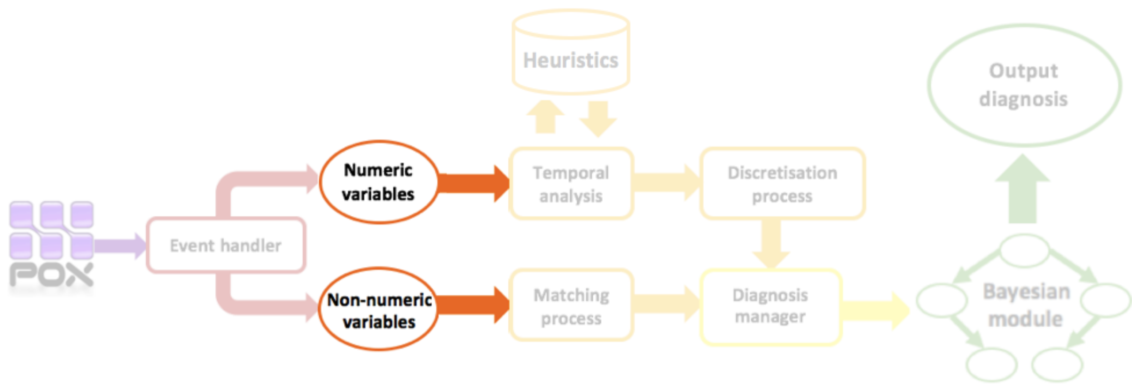

Figure 4.3: Reading from event-handler

The main thread will open the pipe and copy the data sotred by the event-handler (stats from switch ports, switch and port states) to a local dictionary. An example of how to do this for the case of stats from switch ports is in Listing 4.7, where stats is the local dictionary.

Listing 4.7: Reading from pipe

```
while True:
  with open('/dev/shm/poxpipe','r') as pipe:
  data = pipe.read()
  texts = data.split('#')
  for text in texts:
    if len(text) > 0:
      text = json.loads(text)
      if text['type'] == 'switch_portstats':
        dpid = text['data']['switch']
        d = stats[0]
        d['switches'][dpid]['port_stats'] = text['data']['stats']
        stats[0] = d
```

4.4.2.2 Processing and updating data

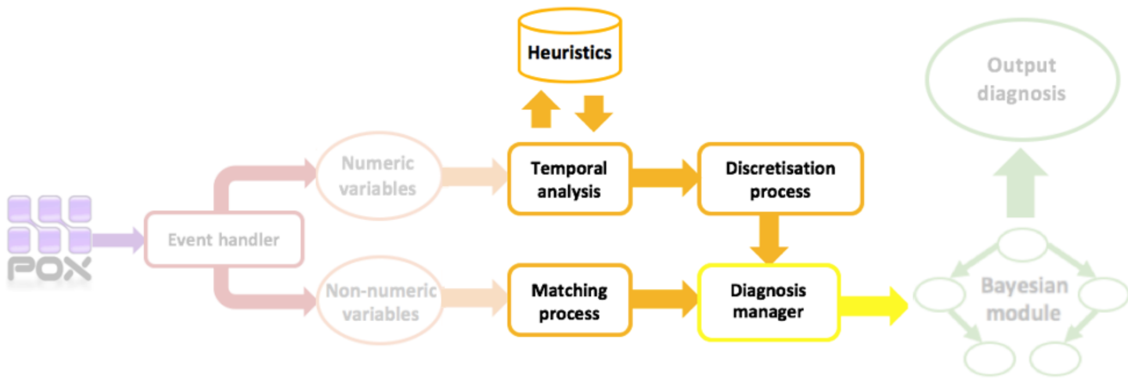

Figure 4.4: Processing and updating data

In the secondary thread we have to process and update the data from our local dictionary where is all the saved data taken from the pipe. As it was explained in Section 3.3, the data has to be adapted to an interval time and then converted into a non-numeric variable for the case of numeric stats; or just parse for the case of non-numeric ones.

We will use an auxiliary dictionary to update the current data every time interval period, it will be defined as follows: `{'tx_packets': 0, 'rx_packets': 0, 'ok_packets': 0, 'eth1': True, 'eth2': True, 'eth3': True, 'state': True}` , keeping the data of interest(transmitted packets, received packets, packets transmitted without errors, and the states of ports and the given switch).

Starting with the numeric variables process, the refresh interval that has been selected is 14 seconds. So for example to calculate the transmitted packets level every 14 seconds we go to the Listing 4.8, where stats_before is our auxiliary dictionary, that how can it be seen is updating at the end of each iteration in order to obtain in the next first operation (txpackets) the number of packets transmitted in that period. Because we remember that statistics comming from pox give us the total number of packets since the begining of the simulation, and that is not what we are interested in.
The called function packets_level is a simple Python function that classifies the number of packets in a level using if conditions.

Listing 4.8: Temporal analysis of transmitted packets

```
while True:
  time.sleep(14)
  txpackets = stats[0]['switches'][dpid_p]['port_stats'][2]['tx_packets']
       - stats_before[0]['sw_stats']['tx_packets']
  txlevel = packets_level(txpackets)
  #updating stats_before
  d = stats_before[0]
  d['sw_stats']['tx_packets'] = stats[0]['switches'][dpid_p]['port_stats
      '][2]['tx_packets']
  stats_before[0] = d
```

The two other numeric-variables (received and lost packets) have the same process structure, but in the case of lost packets it has to take the difference between transmitted packets and received packets by the next switch.

Going on to the non-numeric ones, the process is simplier, since we only have to search in the stats dictionary the state of the ports and switch in question. For example, for the case of a switch state, the function appears in the code 4.9 would be enough. In this code dpid_p is the id of the switch analyzed, and switch_down is the key that has the status of the switch, so if this key is not yet in our stats dictionary it means that this switch has not been modified and therefore will be functioning, if the key appears it means that it has already been modified at least once, then its value is checked and the corresponding boolean value is returned.

Listing 4.9: Switch state

```
def sw_state():
if 'switch_down' in stats[0]['switches'][dpid_p] and stats[0]['
    switches'][dpid_p]['switch_down'] == True:
  aux = False
else:
  aux = True
return aux
```

The function to get the port states is very similar, but checking in the link_status if the link modified belongs to the switch we are analyzing.

This way, we can already send all these values non-numerically along with the situation of the network that appears in the rcf file (referred in Section 4.3.2) to the bayesian module by means of a csv file, if this is in the learning stage; or send only the variables (directly), if is already in the reasoning inference one. In Listing 4.10 can be seen how to introduce these parameters in a csv.

Listing 4.10: Updating csv

```
def _update_csv(rxLvl, txLvl, lostLvl, auxPort, outPort, inPort, sw, rcf):
  outputCsv = open('netstats.csv', 'a', newline='')
  output = csv.writer(outputCsv)
  data = [(rxLvl, txLvl, lostLvl, auxPort,  outPort, inPort, sw, rcf)]
  output.writerows(data)
  del output
  outputCsv.close()
  logger.info('csv updated')
```

4.4.3 Bayesian module

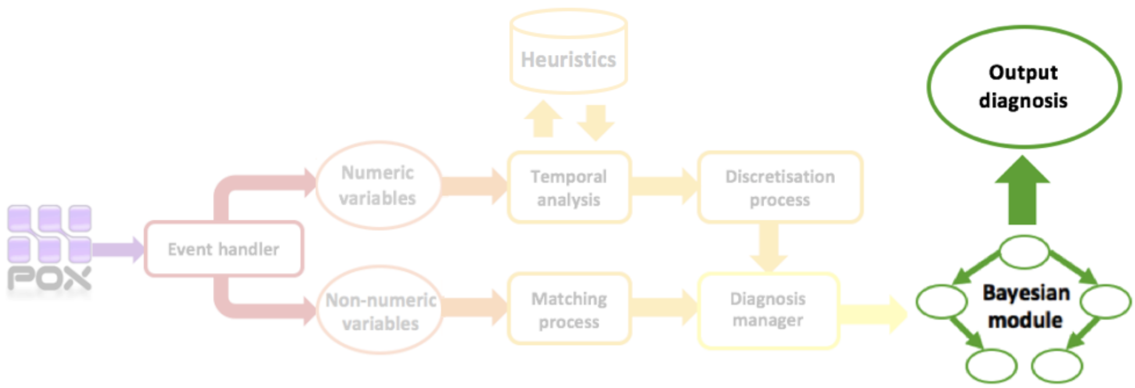

Figure 4.5: Bayesian module

As it was explained in Section 3.4, this last module has two different stages, learning and inference. For the learning stage, the data in the csv file mentioned in Section 4.4.2.2 will serve for training the network, in order to be used in the inference one. Which will give us the predicted situation of the network and the probability of success.

4.4.3.1 Learning stage

The bayesian network will be trained with the program GeNIe SMILE, mentioned in enabling technologies, Section 2.7. To obtain a bayesian network it is necessary to give the program a csv file containing the simulation lines with which we want our network to learn. As we said above, this csv is the one obtained in the previous module, an example of a fwe lines in Figure 4.6.

rx_packets_level	tx_packets_level	lost_packets_level	auxPort_up	outPort_up	inPort_up	sw_up	root_cause_failure
Very_low	Very_low	High	True	False	True	True	tx_link_down
High	High	Negligible	True	True	True	True	Normal_functioning
High	high	Negligible	True	True	True	True	Normal_functioning
Very_low	Very_low	Negligible	True	True	True	True	Normal_functioning
Very_low	Very_low	Negligible	True	True	False	True	rx_link_down

Figure 4.6: Example of csv file

Once the csv is loaded in the GeNIe SMILE, it has to be selected the desired parameters for the new network, such as learning algorithm, link probability, max parent count, iterations, k-fold and especially the class variable, in this case is the root cause failure, because the objective is to take a forecast of the failure from the other variables.

After an automatic process we will obtain an .xdsl file, that is our bayesian network to be used in the second stage.

4.4.3.2 Inference stage

Its implementation requires to execute a java virtual machine (JVM) in our python script, since python doesn't have the smile libraries, which are necessary to work with bayesian networks. So first, the output .xdsl of the previous phase it has to be loaded once the JVM is started. As shown in Listing 4.11, first the paths JVM and smile library are passed as parameters to start the JVM, then the Network package and the .xdsl file are loaded.

Listing 4.11: Starting JVM and loading bayesianNetwork.xdsl

```
jvmPath = jpype.getDefaultJVMPath()
jvmArg = "-Djava.class.path=/home/mininet/smile.jar"
startJVM(jvmPath, jvmArg)
logger.info("JVM started")
net = JPackage('smile').Network()
net.readFile("/home/mininet/sdn-diagnosis/monitor/bayesianNetwork.xdsl")
logger.info("Bayesian network loaded")
```

Now, we can pass the processed data of the simulation to the bayesian module, and this will return the forecast and the success probability. But this has to be implemented (in Listing 4.12). Simply, we set the evidence of the different variables, and then, with a 'for loop', the value of the "root_cause_failure" node with highest probability is searched and returned.

Listing 4.12: Getting forecast and probability

```
def _get_forecast(rxlevel, txlevel, lostlevel, auxPortState, outPortState,
    inPortState, swState):
  net.setEvidence("rx_packets_level", rxlevel);
  net.setEvidence("tx_packets_level", txlevel);
  net.setEvidence("lost_packets_level", lostlevel);
  net.setEvidence("auxPort_up", auxPortState);
  net.setEvidence("outPort_up", outPortState);
  net.setEvidence("inPort_up", inPortState);
  net.setEvidence("sw_up", swstate);
  net.updateBeliefs()
  rcfStatesIds = net.getOutcomeIds("root_cause_failure")
  prob = 0
  for outcomeIndex in range(0, len(rcfStatesIds)):
    if net.getNodeValue("root_cause_failure")[outcomeIndex]>prob:
      prob = net.getNodeValue("root_cause_failure")[outcomeIndex]
      forecast = net.getOutcomeIds("root_cause_failure")[outcomeIndex]
  return forecast, prob
```

It is important to say that the GeNIe interface is not necessary to make the first stage, because we already have the SMILE library in our python script (by means of JPype). But GeNIe makes the things easier since the results of the bayesian network generated can be seen graphically, interacting or analyzing the different results of validation that the program offers. So this way, we can assure that the generated network is valid for our analysis, we will see that in the following section.

CHAPTER 5

Evaluation

In this chapter we explain the different generated models based on the following simulation scenarios: specific switch diagnosis model, generic switch diagnosis model and global diagnosis model. Then, the experimentation and evaluation of those models are described.

5.1 Collecting data for diagnosis models

We have selected three different models to perform the fault diagnosis of the network.
Model 1: Specific switch, which is focused on a single switch and is shown in Section 5.1.1.
Model 2: Generic switch, which takes some data from different switches of the network, but without distinguish the analyzed switch, this model is explained in Section 5.1.2.
Model 3: Global, which takes all the data of every switch of the network, this is presented in Section 5.1.3.

These models have different implementations, the specific switch one is the described in Section 4.4, and the generic switch and global model (more complex) are available on Github along with the rest of the system code, in files called monitorGeneric.py and monitorGlobal.py of the referred link in Section 1.4.

5.1.1 Model 1: Specific switch

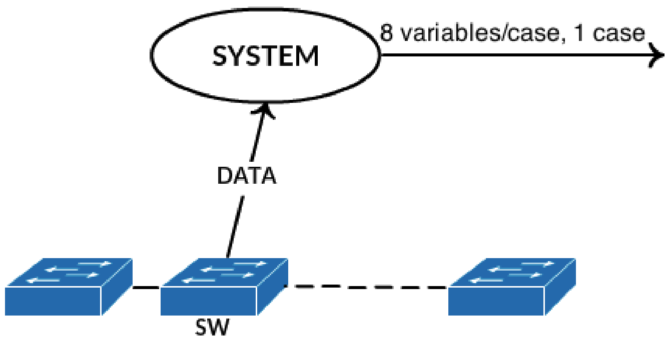

Figure 5.1: Specific switch model

In this model, the switch to be analyzed is a specific switch of the defined topology in the previous chapter, Figure 4.1. We will store data from one switch, so the bayesian network will learn taking this as a sole reference, and it will also be necessary to generate the fault situations in a way that affects the switch at issue, in order to obtain a meaningful analysis. Therefore, we will have 8 variables, including the root cause failure, as we can see in Figure 5.1.

5.1.2 Model 2: Generic switch

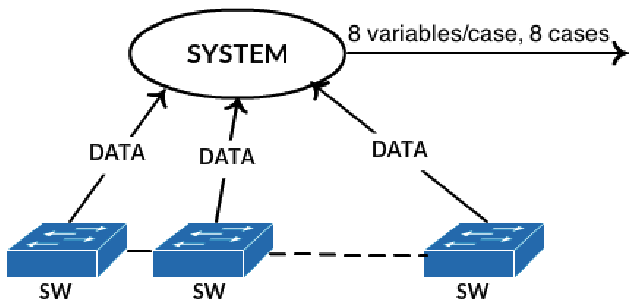

Figure 5.2: Generic switch model

This second model is a combination of the two others, since we have to take data from all the switches as in the third model (which will be explained below), but we save the data in the csv file as it were only one. Therefore, the bayesian module will have the same number of nodes as in the first model, but now receiving 8 cases per simulation (from each one of the switches) instead of 1 as in the specific model, as is shown in Figure 5.2. To perform correctly this analysis we will generate different faults in several switches of the network.

5.1.3 Model 3: Global

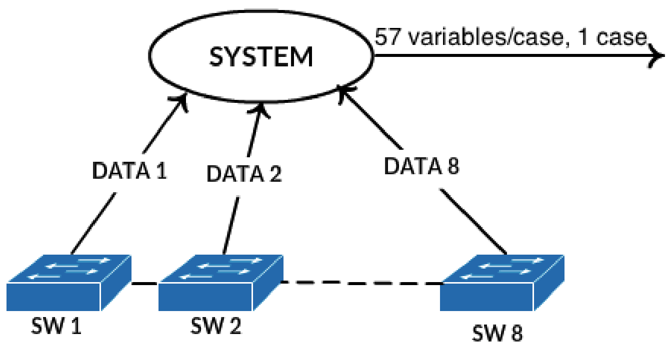

Figure 5.3: Global model

The last model will also take stats from all the switches but now our bayesian network will have 57 nodes (56 of switches and the root cause failure), becasue now it distinguishes among the different switches. So in each simulation it will be obtained one case with all the network data instead of one (or eight) of a particular switch. In Figure 5.3 we can see the idea more clearly. For this model is mandatory to generate all the possible faults in every switch because otherwise the bayesian learning module will not accept the input csv file (the nodes have to change its value in at least one line).

5.2 Diagnosis model generation

To generate the three models we need the csv files mentioned before, with a significant number of simulations, and the bayesian networks are generated with the SMILE GeNIe program, in the process indicated in the learning stage, Section 4.4.3.1.

We will select the following parameters for the three networks:
Learning algorithm: Bayesian search; Max Parent Count: 8; Iterations: 20; Sample Size: 50; Seed: 0; Link Probability: 0.1; Prior Link Probability: 0.1; Max Search Time: 0; K-fold: 10; Class variable: root_cause_failure.

We obtain two kind of networks depending on the model, for the specific and generic (8 variables) we have the one in Figure 5.4, and for the global (57 variables) in Figure 5.5. Where the different nodes (the analyzed variables) are represented with circles and the relations among them with arrows.

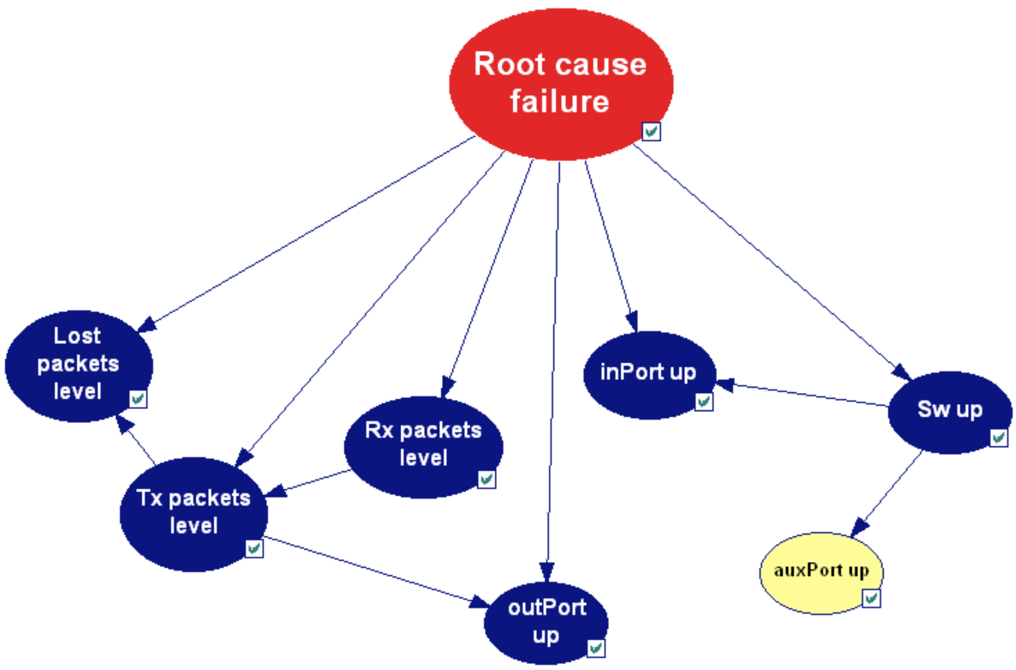

Figure 5.4: Bayesian network of specific/generic model

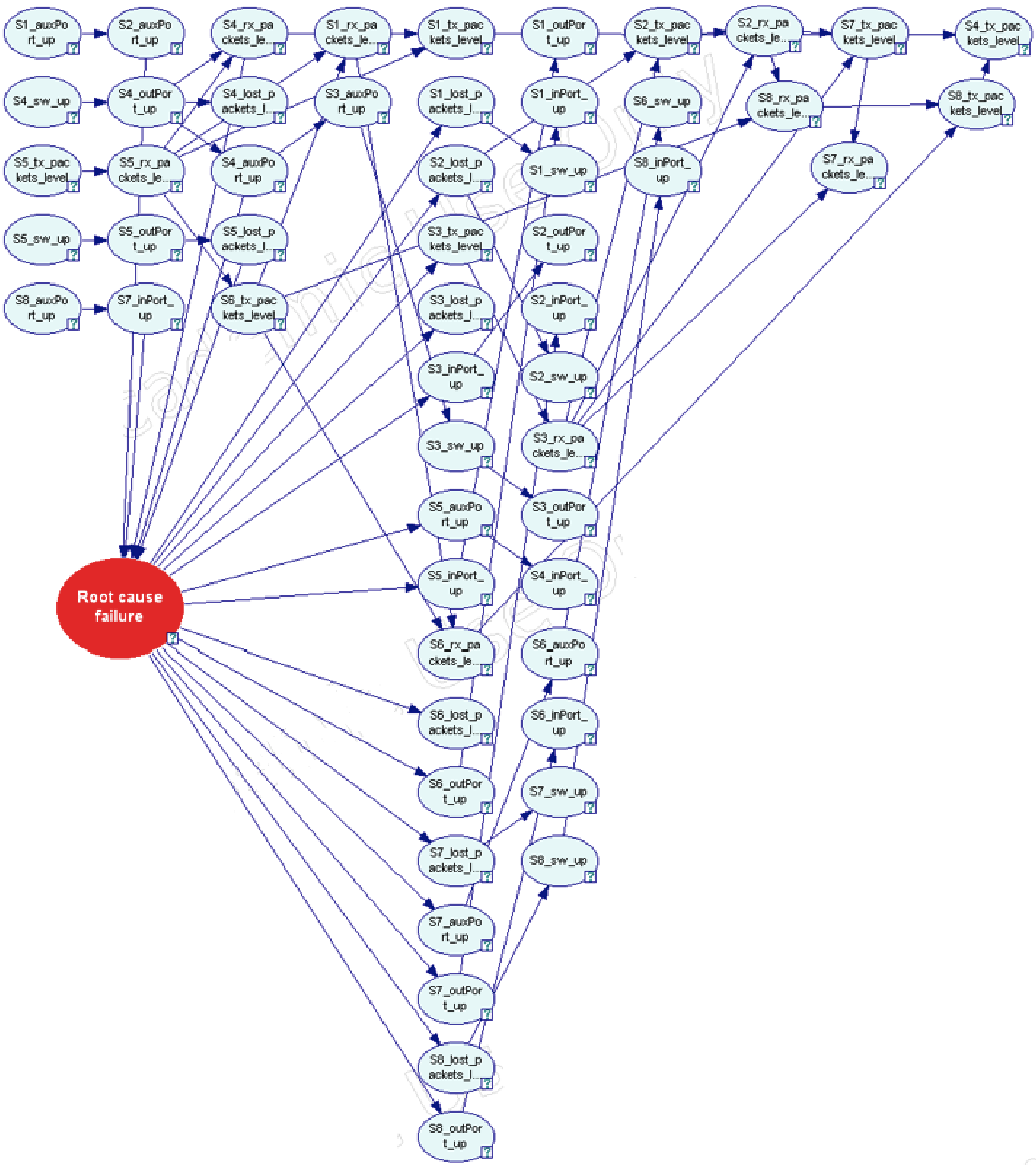

Figure 5.5: Bayesian network of global model

5.3 Diagnosis model evaluation

Once we have the networks we already can introduce in our program the corresponding one in order to diagnose each model, as it was explained in Section 4.4.3.2. But it is interesting to test before the efficiency of the generated module. To test this, we will validate the model with a data file of simulations, and the results explained in the introduction to GeNie 2.8 will be analyzed. The two more representative parameters are the accuracy and the confusion matrix.

If the obtained results are not the desired ones or can be improved, we will apply a python script to remove the noise in the input data csv and re-generate the model. Because if a variable has much more lines than the others the network will diagnose worse.

In the code 5.1 can be seen the implementation of the script for a noise elimination of 50 % in the lines of "Normal_functioning". Where inputFile.csv is our old csv and outputFile.csv is the one that we will obtain. The code basically opens both files, one for reading and the other for writting, then chek the lines that have the value "Normal_functioning", and creates a random number between 0 and 100, if this is higher than 50 it will copy the line into the new .csv (aproximately half the time).

Listing 5.1: Data tidying

```
inputCsv = csv.reader(open("inputFile.csv", "r"))
outputCsv = csv.writer(open("outputFile.csv", "a"))
aux = 0
for line in inputCsv:
  if "Normal_functioning" not in line:
    outputCsv.writerow(line)
  else:
    randomNumber = random.randrange(100)
    if randomNumber>[50]:
      csvSalida.writerow(line)
    else:
      aux = aux + 1
print(aux,"deleted lines")
```

5.3.1 Evaluation of Model 1: specific switch

The obtained parameters (accuracies of the different values of the root cause failure node and confusion matrix) are presented in Table 5.1 and 5.2. In the confusion matrix the rows indicates the real situation, while the columns shows the predicted situation.

Root cause failure	True positives	Cases	Accuracy
Normal functioning	230	262	87,78%
Rx link switch down	15	17	88,23%
Server down	36	53	67,92%
Switch down	46	46	100%
Switch malfunction	46	60	76,66%
Tx link switch down	15	18	83,33%
Total	**388**	**456**	**85,08%**

Table 5.1: Specific model accuracies

	N.func	Rx.L.down	Serv.down	Sw.down	Sw.malf	Tx.L.down
N.func	**230**	0	**12**	0	**18**	**2**
Rx.L.down	**2**	**15**	0	0	0	0
Serv.down	**17**	0	**36**	0	0	0
Sw.down	0	0	0	**46**	0	0
Sw.malf	**14**	0	0	0	**46**	0
Tx.L.down	**3**	0	0	0	0	**15**

Table 5.2: Confusion matrix of specific model

The results are quite good but the worst ones have been obtained in the server down with a 67%. As the confusion matrix indicates, "Normal_functioning" is diagnosed many times like a "server down" situation, this is because of the noise mentioned at the begining of this chapter, in Section 5.3. So we can apply the script in Listing 5.1 to the "Normal_functioning" varible, for example with a 50% probability of removing a sample, and generate the model again. Now we obtain better results:

Root cause failure	**True positives**	**Cases**	**Accuracy**
Normal functioning	103	128	80,46%
Rx link switch down	15	17	88,23%
Server down	49	53	92,45%
Switch down	46	46	100%
Switch malfunction	53	60	88,33%
Tx link switch down	15	18	83,33%
Total	**281**	**322**	**87,26%**

Table 5.3: Specific model accuracies after data tidying

	N.func	**Rx.L.down**	**Serv.down**	**Sw.down**	**Sw.malf**	**Tx.L.down**
N.func	**103**	0	**13**	0	**11**	**1**
Rx.L.down	**2**	**15**	0	0	0	0
Serv.down	**3**	0	**49**	0	**1**	0
Sw.down	0	0	0	**46**	0	0
Sw.malf	**7**	0	0	0	**53**	0
Tx.L.down	**1**	0	**2**	0	0	**15**

Table 5.4: Confusion matrix of specific model after data tidying

The results have improved overall, especially the server down one (until 92%). Despite having lost some accuracy in Normal functioning, we have now a more reliable network with all the variables above 80% of success.

5.3.2 Evaluation of Model 2: Generic switch

Accuracies and confusion matrix for the generic model are shown in Table 5.5 and 5.6.

Root cause failure	**True positives**	**Cases**	**Accuracy**
Normal functioning	1224	1262	96,98%
Rx link switch down	36	39	92,31%
Server down	8	160	5%
Switch down	23	24	95,83%
Switch malfunction	13	41	31,71%
Tx link switch down	29	39	74,35%
Total	**1333**	**1565**	**85,17%**

Table 5.5: Generic model accuracies

	N.func	**Rx.L.down**	**Serv.down**	**Sw.down**	**Sw.malf**	**Tx.L.down**
N.func	**1224**	**15**	**3**	0	**8**	**12**
Rx.L.down	**3**	**36**	0	0	0	0
Serv.down	**152**	0	**8**	0	0	0
Sw.down	**1**	0	0	**23**	0	0
Sw.malf	**28**	0	0	0	**13**	0
Tx.L.down	**9**	0	0	0	**1**	**29**

Table 5.6: Confusion matrix of generic model

The results in this module are not good, the "server_down" and "switch_malfuncion" situations have very low success probabilities so usually they will not diagnosed correctly. As we can see in the confusion matrix 5.6 these situations are diagnosed most of times like "Normal_functioning", this is because the values are very unbalanced, the lines of this value are much higher than the rest (80% of the total).

Therefore, we will improve the model removing this noise, we will introduce this time a percentage of 85%.

Generating the model after this we obtain the next results:

Root cause failure	True positives	Cases	Accuracy
Normal functioning	160	186	86,02%
Rx link switch down	39	39	100%
Server down	133	160	83,12%
Switch down	23	24	95,83%
Switch malfunction	31	41	75,60%
Tx link switch down	33	39	84,61%
Total	**419**	**489**	**85,68%**

Table 5.7: Generic model accuracies after data tidying

	N.func	Rx.L.down	Serv.down	Sw.down	Sw.malf	Tx.L.down
N.func	**160**	0	**18**	0	**6**	**2**
Rx.L.down	0	**39**	0	0	0	0
Serv.down	**25**	0	**133**	0	**1**	**1**
Sw.down	0	**1**	0	**23**	0	0
Sw.malf	**8**	0	**1**	0	**31**	**1**
Tx.L.down	**6**	0	0	0	0	**33**

Table 5.8: Confusion matrix of generic model after data tidying

Now, the results have improved a lot, all the situations have an accuracy of at least 75% and therefore is an applicable model. Although the results are worse than in the specific model (because of the different behaviour of the switches) there are two positive points, this model can be applied to every switch of the network while the specific can just be applied for one; and its generation is much faster (8 cases per simulation).

5.3.3 Evaluation of Model 3: Global

In this model, we remember that there are 21 possible situations, so we will just show the accuracies (in Table 5.9), since the confusion matrix is too long (20x20 matrix) and does not provide meaningful data, because the confusion in this model is very low.

RCF	TP	Cases	Acc	RCF	TP	Cases	Acc
N. functioning	247	272	90,80%	S6 malfunction	15	24	62,50%
S1 down	19	19	100%	S6-S1 down	17	20	85%
S1 malfunction	22	31	70,96%	S7 down	13	13	100%
S2 down	26	26	100%	S7 malfunction	17	24	70,83%
S2 malfunction	17	26	65,38%	S7-S2 link down	16	19	84,21%
S3 down	25	25	100%	S8 down	21	23	91,30%
S4 down	16	16	100%	S8 malfunction	28	38	73,68%
S5 down	16	16	100%	S8-S6 link down	25	28	89,28%
S5-S4 down	17	18	94,44%	S8-S7 link down	21	23	91,30%
S6 down	25	25	100%	**Total**	**617**	**706**	**87,39%**
Server down	14	20	70%				

Table 5.9: Global model accuracies

The results obtained are quite good bearing in mind that this is a model that diagnose every faults in the network, but the generation speed is low (because it is required to generate 21 different situations). The lower accuracies are in the "malfunction situations", that are sometimes diagnosed as "Normal_functioning".

To further improve the results we will select as in the specific situation a 50% of noise suppresion, obtaining the results shown in Table 5.10. We can see a little improvement in the mentioned values, having all the accuracies above 70%. There is less confusion in the "malfuncion situations" before the noise suppresion.

RCF	TP	Cases	Acc	RCF	TP	Cases	Acc
N. functioning	113	125	90,40%	S6 malfunction	17	24	70,83%
S1 down	19	19	100%	S6-S1 down	17	20	85%
S1 malfunction	24	31	77,41%	S7 down	13	13	100%
S2 down	26	26	100%	S7 malfunction	20	24	83,33%
S2 malfunction	22	26	84,61%	S7-S2 link down	16	19	84,21%
S3 down	25	25	100%	S8 down	21	23	91,30%
S4 down	14	16	87,5%	S8 malfunction	29	38	76,31%
S5 down	16	16	100%	S8-S6 link down	26	28	92,85%
S5-S4 down	17	18	94,44%	S8-S7 link down	21	23	91,30%
S6 down	25	25	100%	**Total**	**495**	**559**	**88,55%**
Server down	14	20	70%				

Table 5.10: Global model accuracies after data tidying

5.3.4 Discussion

As general conclusions of the analysis of these three models, we can stand out the following aspects. The specific and generic model have the advantage that they would have a manager for each switch, and the global an unique manager for all the network, so if the manager stops working it would crashes all the diagnosis system. But in the other hand, the global model collects information for all the network and their diagnostics are the most accurate of the three, as we have seen in the previous tables.

In generation terms, the global model is the more complex, since you have to take data from all the switches and generate all the possible faults, this is an incovenient. The specific and generic do not need so much information and they are easier to generate. But for example, if we chose the specific model, possibly it does not work as well on all the switches, because switches have different behaviours. And the generic, despite having seen that works for all the switches, does not have the precision of the global one.

CHAPTER 6

Conclusions and future work

In this chapter we will describe the conclusions extracted from this project, the achievements and possible paths about future work.

6.1 Conclusions

The development of this fault diagnosis system shows the potential possibilities in management and applications that SDNs offers. Combining this technology with machine learning tecniques as Bayesian search in our case, it is possible to diagnose different fault situations in a network in a very precise way.

One of the most important aspects is the scalability, since the point is to apply the system to real networks, in which the complexity is much bigger than the scenario considered in this work. It would be necessary an automatic model applicable to any network, regardless the number of servers, hosts or links.

Focusing on the objectives fixed in the introduction, we can highlight the following goals achieved:

- Learning SDN and OpenFlow concepts: During a period of experimentation with these technologies before starting working on the project, and also new knowledge have been acquired during the realization of the work.
- Defining and implementing a streaming scenario: Starting from a given scenario and taking that as the basis of the project, we have larned to define and adapt it to the different developed analysis; adding, modifying or deleting the different devices, describing the rules of the traffic routing, and starting or finishing a streaming service.
- Learning the basis of the management and the monitoring of SDN using the developed scenario as benchmark: During the implementation of the project, and having already learned the concepts and skills mentioned above, we have genereted the system, and acquiring the knowledge to monitor the network in real time. All of this thanks to the facilities offered by the used technologies, which allows to obtain and process many interesting data of the network.
- Learning the use of bayesian networks to reason about the root cause failure to diagnose the network during its operation: Studying how the bayesian model works and applying it in the context of the project, we have obtained an efficient inference stage of this module, by means of a previous training phase.

The main problems found during the project have been:

- Understanding how the controller POX sends the network information: Since it uses dictionaries of many dimensions, which can result a bit complex if you have never worked with this controller before.

- The implementation of module processor: The manipulation and adaptation the data to the different time intervals and to the three proposed models, with the aim of making this information meaningful to the system.

- The integration of the bayesian module in the system: Because, as it was mentioned in Section 4.4.3, Python has not the corresponding libraries, so it was necessary to combine Python with Java, by means of JPype, and make them to work in the same script. This brought some compatibility issues.

6.2 Future work

There are several lines than can be followed to continue and extend features of this work. In the following points some of them are presented.

- Introduction of queue stats as variables: Since POX is still in progress, the queue functionalities offered do not work properly in some Open vSwitch versions, and there are problems for their correct configuration. When this issue will be solved, it would be interesting to add queue stats such as transmitted packets by queue, or dropped packets by queue in the different switches. This would open possibilities for new analytical situations in the network.

- New network situation, link congestion: With the introduction of additional servers and making them send a large amount of random traffic to some switch, we could get this switch congested and to cause losses in the video flow, creating a new situation in which the client would watch the video with stops and less quality. To make this situation, the point commented before (queue stats variables) is necessary, to take the number of dropped packets or transmitted by queue.

- New network situation, reduction of a link capacity: In this case, we just would decrease the capacity of a link. The consequence for this situation it would be the same as in the link congestion, the queue would start functioning and it could have losses too. But the cause is different, the switch congestion is due to the low capacity in the link, not because the sending of too much traffic.

- Analysis the traffic network at layer 3 (by means of FlowStats): Analyzing the network at layer 3 it would open many possibilities in the system, mainly in the generation of the models and during data processing, because you can work with IP addresses, viewing the source and destination of some flow, or differentiating the kind of traffic depending on the listening ports used in the hosts.

- Machine learning between the temporal analysis and herusitics module. The idea is to introduce a new module with a machine learning technique in order to classify the received levels in basis of the ones received before. This way, the module is learning all the time and the range of level based in heuristics is continually adapting to the new situations.

- Increase the complexity of the network: In order to generate new situations and make a more realistic analysis, it could be added a larger number of equipment in the network, changing the topology distribution or adding different services such as VoIP.

Bibliography

[1] O. N. Foundation, "Openflow switch specification version 1.5.1," 2015.

[2] "Archive openflow web, http://archive.openflow.org/wp/learnmore/."

[3] "Sdx central web, https://www.sdxcentral.com/sdn/definitions/sdn-controllers/openflow-controller/."

[4] "Openflow stanford university, https://openflow.stanford.edu/display/onl/pox+wiki."

[5] "Python web, https://www.python.org."

[6] "Open vswitch web, http://openvswitch.org."

[7] "Mininet web, http://mininet.org."

[8] J. L. H. Ramirez, "Guía de implementación y uso del emulador de redes mininet," 2015.

[9] "The linux documentation project web, http://tldp.org/howto/traffic-control-howto/components.html."

[10] "The linux foundation wiki, https://wiki.linuxfoundation.org/networking/netem."

[11] "Virtualbox web, https://www.virtualbox.org."

[12] "Bayes fusion web, http://www.bayesfusion.com."

[13] M. J. Druzdzel, "Smile:structural modelling, inference, and learning engine and genie: A development environment for graphical decission-theoretic models," 1999.

[14] "Sourceforge project web, http://jpype.sourceforge.net."