UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO DE INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

APPLICATION OF BAYESIAN REASONING FOR FAULT DIAGNOSIS OVER A CONTROLLER OF A SOFTWARE DEFINED NETWORK

FERNANDO BENAYAS DE LOS SANTOS

2018

TRABAJO FIN DE GRADO

Título:	Aplicación de razonamiento Bayesiano para el diagnóstico de fallos sobre un controlador de una Red Definida por Software
Título (inglés):	Application of Bayesian Reasoning for Fault Diagnosis over a controller of a Software Defined Network
Autor:	Fernando Benayas de los Santos
Tutor:	Álvaro Carrera Barroso
Departamento:	Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:

Vocal:

Secretario:

Suplente:

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

TRABAJO FIN DE GRADO

APPLICATION OF BAYESIAN REASONING FOR FAULT DIAGNOSIS OVER A CONTROLLER OF A SOFTWARE DEFINED NETWORK

Fernando Benayas de los Santos

Junio de 2018

Agradecimientos

Me gustaría dar las gracias a todas las personas que me han ayudado en el desarrollo de este proyecto.

A mis padres, por pagar en dinero, paciencia y cariño todo esto. Al Grupo de Sistemas Inteligentes, y en particular a Álvaro, por creer en mí inexplicablemente. A mis compañeros del laboratorio, por ayudarme a escribir estos agradecimientos.

Pero, por encima de todo, gracias a los que estuvieron de principio a fin.

Va por vosotros.

Acknowledgement

I wish to thank everyone who has helped me in the development of this project.

Thanks to my parents for paying for this with money, patience and affection. Thanks to the Intelligent Systems Group (and especially thanks to Álvaro) for inexplicably believing in me. Thanks to my co-workers, for helping me in writing these acknowledgements.

But above all, thanks to the ones who stayed from the beginning to the end.

This is for you.

Resumen

Factores como el progresivo aumento en consumo y calidad de servicios de vídeo, la adopción de tecnologías 5G, la creciente presencia de dispositivos IoT y el consumo de servicios en la nube están creando un considerable problema de sobrecarga de tráfico en las redes de telecomunicaciones actuales. Por ello, es necesaria la implementación de políticas de tráfico flexibles. Esto permitiría una gestión dinámica del enrutamiento, basada en parámetros tales como la carga de tráfico, o la Calidad de Servicio. Para conseguir esta gestión dinámica es necesario el uso de las tecnologías de Redes Definidas por Software (SDN).

Las Redes Definidas por Software nos permiten desacoplar el plano de control del plano de enrutamiento y centralizar el primero, permitiendo acceso a éste a través de una Application Programming Interface (API). Esto simplifica el proceso de introducción de políticas de tráfico en la red al delegar en el controlador la tarea de inyectarlas en los elementos de la red, permitiendo una gestión más ágil de la misma.

A pesar de las ventajas mencionadas, el uso de SDN puede conllevar problemas de debilidad ante fallos. La capacidad de cambiar frecuentemente las políticas de tráfico conlleva la posibilidad de introducir frecuentemente configuraciones erróneas o corruptas. Esto podría perjudicar seriamente la adopción de SDN. Por tanto, es necesario un sistema que permita diagnosticar estos fallos.

El sistema propuesto se basa en en el aprendizaje de una red Bayesiana a partir de los datos extraídos y procesados de una SDN, buscando relaciones causales entre valores de los datos y estado de la red. Para emular el funcionamiento de una SDN real, se ha diseñado una simulación con varias redes libres de escala unidas, en las que se han inyectado varios tipos de tráfico. Dicha red Bayesiana será utilizada posteriormente para diagnosticar nuevos fallos introducidos en la red, razonando con los datos extraídos de esta.

Los resultados presentan una alta tasa de acierto: obtenemos una precisión del 92.2%, una exhaustividad ("recall") del 91.9%, una exactitud del 97.8% y un Valor-F de 91.2% para el mejor de los modelos probados.

Palabras clave: Software Defined Network (SDN), red Bayesiana, diagnóstico de fallos, Machine Learning.

Abstract

Current trends, such as the increasing quality and consumption of video services, the adoption of 5G technologies, and the growing presence of IoT devices are overloading current telecommunication. This is further aggravated by the adoption of cloud services. In order to face these challenges, flexible networking policies are needed. This would enable dynamic, fast re-routing, depending on parameters such as traffic load and Quality of Service. This dynamic network managing is made possible by the Software Defined Networks (SDN). SDN decouple the forwarding plane and the control plane, centralises the latter and exposes it through an API. Therefore, by delegating on the controller the task of injecting network policies into the network elements, the process of creating and injecting traffic policies into the network is simplified, allowing for a more agile management.

Notwithstanding the advantages mentioned before, SDN have some resiliency issues. Frequent changes in networking rules entails constant possibilities for faulty traffic policies to be introduced in the network. This could severely hinder the benefits of switching from legacy to SDN technology. Hence, a system that allows users auditing these faults is needed.

The proposed system is based on a Bayesian network learned from labelled and processed data obtained from SDN, looking for causal relationships between data values and current state of the network. In order to obtain labelled data, SDN has been simulated, creating multiple scale-free connected networks, where multiple traffic types have been implemented. The resultant Bayesian network is then used to diagnose the status of the network from new labelled and processed data.

Regarding the results, we have obtained a precision value of 92.2%, a recall value of 91.9%, an accuracy value of 97.8%, and a F1-Score value of 91.2% in the best of the models tested.

Keywords: SDN, Bayesian network, fault diagnosis, Machine Learning.

Contents

List of Figures

List of Tables

Glossary

API: Application Programming Interface
AUC: Area under the ROC curve
BDeu: Bayesian Dirichlet equivalence uniform
BFD: Bidirectional Forwarding Detection
BIF: Bayesian Interchange Format
CAGR: Compound Annual Growth Rate
CLI: Command Line Interface
CPT: Conditional Probability Table
CSP: Cloud Service Provider
CSV: Comma Separated Values
DAG: Directed Acyclic Graph
GUI: Graphical User Interface
HTML: Hypertext Markup Language
HTTP: Hypertext Transfer Protocol
IoT: Internet of Things
ISP: Internet Service Provider
JVM: Java Virtual Machine
LLDP: Link Layer Discovery Protocol
LOS: Loss of Signal
MD-SAL: Model-Driven Service Abstraction Layer
MSO: Multiple Services Operator
NFV: Network Function Virtualisation
PGM: Probabilistic Graphical Model
PSM: Problem-Solving Method
P2P: Peer to Peer
REST: Representational State Transfer
RTP: Real Time Transport Protocol
RTSP: Real Time Streaming Protocol
SDN: Software Defined Network
SMILE: Structural Modeling, Inference and Learning Engine

SMTP: Simple Mail Transfer Protocol
TCP: Transmission Control Protocol
UDP: User Datagram Protocol
VLC: VideoLAN Client
VM: Virtual Machine
YANG: Yet Another Next Generation

CHAPTER 1

Introduction

In this chapter, first we depict the motivations that led us to the developing of this project in Section 1.1. Then, in Section 1.2, we enumerate and explain the goals of this project. Finally, we describe the structure of the document in Section 1.3.

1.1 Motivation

Current telecommunication networks are facing challenging times. According to Cisco [1], mobile data traffic will increase dramatically over the following years. Monthly mobile traffic will grow from 7.2 exabytes in 2016 to 49 exabytes in 2021. This is partially due to a consistent increase in the number of devices, which is expected to continue, reaching 11.6 billion mobile devices by 2021. Also, mobile devices are growing smarter: although the number of devices is expected to grow, a fast decline in the number of non-smart devices is expected, diminishing from 3.3 billion to 1.5 billion. This shows that people are updating their non-smart mobile devices into smart ones, which generate and consume more network traffic.

Furthermore, network traffic increment is not limited to mobile networks and devices. Cisco predicts that global IP traffic will reach 3.3 ZB per year by 2021 [1]. It also shows

that an important role in this growth is being played by video traffic due to trends such as increasing quality in video streams, live Internet video replacing traditional TV broadcasting and increasing penetration of Video-on-Demand services. Lastly, the emergence of Virtual Realty and Augmented Reality will increase 20-fold in the 2016-2021 period.

All these factors create an overload on telecommunications networks. An approach to solve or ease this issue could be escalating link capacities and network devices processing capabilities. This approach is already being explored, focusing on a progressive expansion of optical fiber links and an increase of network devices processing power [2]. However, this path is severely limited by what is physically possible. Therefore, some additional measures must be taken to ease this load.

In order to solve this issue, the computer networking industry is adopting the use of SDN to allow a more flexible management of the network. Therefore, we could relieve network elements particularly affected by congesting traffic flows and enable fast, reactive re-routing. In legacy networks, the process of changing traffic policies is highly consuming both in time and personnel: every new policy must be thoroughly planned in order to avoid any possible incompatibility with current policies. Once is planned, each change must be individually injected in each network element involved in the policy. Hence, dynamic, reactive management of traffic policies is not possible, since executing each change would take too much time. Here is where SDN features could provide an advantage compared to legacy systems.

Particularly, the capability of SDN to separate the control plane from the data plane and centralise the latter dramatically increases flexibility. Since the process of pushing changes in each network element is now being managed by the SDN controller, a fast implementation of each change introduced in the network policies is plausible. This is done by using a protocol specifically designed for communications between the controller and each node, the OpenFlow protocol, which is being standardised. This feature also adds another advantage to network design: in legacy networks, proprietary communications protocols increase complexity in multi-vendor networks. This complexity is addressed by the use of OpenFlow as the standard protocol for communications between each node (regardless of vendor) and the network controller.

These advantages are being noted by the communication networks community. The Business Services division of Orange remarks the “agility and speed” that SDN brings to current networks [3]. Cisco considers SDN technology to be “at the heart of the of a new generation of networking technologies that transforms the way a business operates in the digital age” [4]. Furthermore, according to Deloitte [5], SDN will change the way Telcos

operate their networks, "in the same way that IP and the Internet transformed the sector twenty years ago".

However, the implementation of a flexible management system using SDN features can aggravate some issues in network management. Particularly, the injection of frequent changes in networking rules made by software systems entails constant possibilities for faulty traffic policies to be introduced in the network. In legacy systems, due to its stiffness, network policies had to be thoroughly designed and carefully introduced in each device; therefore, the risk of introducing faulty configurations was severely limited. Since policy changes can be now quickly and easily introduced, less time and effort is being spent in network management: therefore, the resiliency of SDN must be improved.

The design and implementation of a fault diagnosis system based on Bayesian reasoning is presented in this work. In order to test the diagnosis system, a complex network is simulated, injecting traffic that resembles realistic traffic present in current networks, such as P2P, chat, e-mails or video streaming. In this network, some faults are generated, in order to create a "faulty status" in which we collect data. Once we have enough data of each possible status in the SDN, a Bayesian network is learnt from it, finding causal relationships between data values and SDN status. Then, this Bayesian network is used to diagnose possible faults in the network.

1.2 Project Goals

In order to develop the system mentioned in the previous section, the following goals are defined:

1. To define and develop a simulated SDN complemented with traffic streams within the network, generating faulty configurations in order to generate errors in the network.
2. To collect information on the status of the network, trying to find data fields that would represent status changes.
3. To use these data to perform supervised learning of a Bayesian network through the use of machine learning techniques.
4. To test this Bayesian network, using new datasets obtained from new SDN simulations.

1.3 Structure of the Document

The rest of the document is structured as follows.

In Chapter 1, a brief introduction and the project goals are presented. Then, in Chapter 2, the background of this project is presented. Also, the systems and tools used to implement this project are described. Next, Chapter 3 presents and explains the proposed architecture. Then, in Chapter 4 we detail the implementation of a prototype that follows the proposed architecture. Next, in Chapter 5 test the prototype detailed in the previous chapter is tested, the results obtained are analysed. Finally, in Chapter 6, some conclusions are presented and some possible paths for future work are explored.

CHAPTER 2

Enabling Technologies

To develop the system proposed, first we have searched for similar work in the research community, both in the use of of Bayesian networks for diagnosis purposes and the development of faults management systems. In order to develop the system proposed in this document, we have made use of a wide variety of technologies. We applied technologies that allow the implementation of the SDN paradigm, such as OpenFlow and Open vSwitch. We have also applied technologies related to Bayesian reasoning for the diagnosis of the network. Finally, we have benefited from techniques for the simulation of computer networks environments.

2.1 Introduction

In this chapter, we provide a background on the status of current research in the fields of Bayesian-based diagnosis and fault management in SDN in Section 2.2. Then, in Section 2.3, we describe the basis of the SDN technology. Next, the fault diagnosis process is described in Section 2.4. Finally, the most relevant technologies used in the network simulation are depicted in Section 2.5.

Tools and software libraries used in the developing of this project are listed in Appendix

C.

2.2 Background

In this section, first we provide a background in faults management in SDN in Section 2.2.1. Then, we describe current work in Bayesian-based diagnosis in Section 2.2.2.

2.2.1 Faults Management in SDN

Most of the research in fault diagnosis in SDN scenarios focuses on links/switches failover within the data plane, many aiming at agility improvement by decentralising tasks. For example, Kempf et al. [6] proposes that link failover is detected (and fixed) in switch level instead of the controller using Bidirectional Fowarding Detection (BFD) instead of Link Layer Discovery Protocol (LLDP) in order to improve restoration time. A similar approach is taken by Kim et al. [7], which also use switches for tasks that do not require full knowledge of the network, switch and link failures are detected by LLDP, and repaired by calculating multiple paths with updated information. Cascone et al. [8] uses "heartbeat" packets sent by nodes to detect faulty routes: when a node does not receive them and packet rate drops below a threshold, a node can request heartbeats to detect if the link is down. Then, a "link down" message is sent along the primary path until a node can found a detour for that path.

Similarly, Capone et al. [9] proposes an extension to the OpenFlow protocol, called "OpenState". It introduces state machines into OpenFlow switches, triggering transitions by packet-level events. Therefore, when a node receives notification of a packet affected by a failure in some node or link, it activates a state transition that creates a pre-computed detour for that flow. This alternative path is calculated considering link capacities, and cycle and congestion avoidance.

In contrast, some effort has been put also in centralised management of link failures. Sharma et al. [10] propose that restoration and protection recovery methods for in-band networks are fully implemented at the controller, instead of delegating them to the switches, whose only involvement in the process is to detect failures through Loss of Signal (LOS) and BFD.

Another interesting approach to malfunctioning link detection is the one proposed by SDN-RADAR [11], which relies on agents, acting as clients that report bad performance on services consumed by them. When bad performance is reported, the system extracts and

stores the path defined for the network flow related to each client and service, finding the link that appears in the most number of 'faulty' flows. Withal the original approach on failure detection using agents feedback, this work focuses only in malfunctioning links.

In the previous research works, failure was only considered in links or switches. Howbeit, this concept is severely limited, since failures could happen in any level of a SDN system: consequently, some research has been made in fault detection in SDN where failures at other levels are considered. Tang et al. [12] propose a system mapping by combining the network topology with the "Policy View" of each service. These views are reports consisting in a pair of logical end nodes, traffic pattern specifications for the service, and a list of required network functions for the provisioning of the service. This system mapping is known as "Implementation View". Using this view and a SDN reference model, a belief network is built for each service. Many belief networks are combined to infer the root cause and the fault location. Following the same global approach, Beheshti and Zhang [13] propose a system that studies all the possible locations of the controller and their related routing trees (Greedy Routing Tree, a modified shortest-path tree for more resiliency), choosing the location with the most resilient tree.

A different method is proposed by Chandrasekaran and Benson [14], considering SDN application failures instead of link/nodes ones. In order to achieve fault tolerance for SDN applications, a new isolating layer, called "AppVisor", is defined between applications and SDN controller. It separates the address space between both controller and applications by running them in different Java Virtual Machine (JVM), so software crashes are contained into their JVM, but keeps communication open between them. It also defines a module which enables network transforming transactions between application and controller to be atomic and invertible. Therefore, an application crash in the middle of a network transforming message does not affect the network itself. Lastly, it defines a module (the Crash-Pad module) that takes application snapshots before it process an event, so it reverts to the previous state in case of failure during the event processing. In order to avoid repeating the same failure, the snapshot is modified. This can be done by either ignoring the failure-inducing event or modifying it accordingly to pre-defined policies.

There is also other interesting works for our proposal in other fields related with different SDN aspects. For instance, one common approach consists in dealing with the issue of a centralised controller being a single point of failure, as presented by Botelho et al. [15]. They propose an architecture with a back-up controller and replicated data stores. A similar system is proposed by Li et al. [16]. In this paper, many controllers are defined for a single network, but in this case the network is also protected against "hacked". Katta et

al. [17] consider a situation where a controller fails during handling of a switch state. This is because the entire event-processing cycle is managed as a transaction, instead of just keeping a consistent state between the controllers. Nevertheless, all of them are passive systems. In contrast, an active approach to the single point of failure issue has been proposed by Bouacida et al. [18]. In this case, different classification algorithms are able to predict if work loads at the controller (as the ones caused by the simultaneous introduction of multiple new flows) are short or long-term loads, which is quite useful since the latter is capable of bringing down the controller and shutting down the network if none actions are taken.

Finally, it is interesting the approach proposed by Sánchez et al. [19], where a self-healing architecture for SDN is presented, including monitoring of data, control and service levels. In this architecture, self-modelling techniques are applied to dynamically build diagnosis models synthesised as Bayesian networks. These models are based on topology information requested from the controller. Then, the diagnosis result is sent to the recovery block, which chooses the appropriate strategies to fix the failure diagnosed. In order to complement this approach, our proposal focuses more on the behaviour of the network than in the topology itself, as described below.

As we can see, most of the related work focuses on managing link failures, or failures that completely disable SDN elements (either links, nodes, SDN controllers, or applications). We would like to take a different approach, extending failure diagnosis to faulty traffic configurations (which could hamper the provision of specific services within the network). We also intend to address the design of a self-healing service.

2.2.2 Diagnosis Based on Bayesian Networks

Bayesian networks' predictive abilities are being heavily used in medical diagnosis. For example, in the field of oncology, multiple Bayesian network techniques are used to help in the diagnosis of cancer. In [20], two Bayesian network approaches are compared (learning the Bayesian network directly from training data using the K2 learning algorithm, and using some environmental knowledge to create a Naïve Bayes classifier) in the diagnosis of breast tumors as malignant or benign based on information provided by a cytology on cells from the tumor. In both approaches, good results are obtained (an Area under the ROC curve (AUC) of 0.971 and 0.964 respectively). Another approach would be to design "by hand" the network instead of using a supervised learning algorithm. For example, Lakho et al. [21] create a Bayesian network using knowledge obtained from health experts. This Bayesian network is then used to diagnose the probability of suffering Hepatitis B, C or D, obtaining an overall accuracy of 73.33% over the test data.

However, the capability of Bayesian networks to encode casual relationships is not used only in medical diagnosis. Bayesian networks are used in diagnosis in a broad spectrum of topics. For example, in ceramic shell casting, Bayesian networks are used to predict the root cause of deformations in ceramic shells based on the shape and metrics of those deformations, as shown in [22]. In [23], a Bayesian network is used to model the availability analysis and fault diagnosis of a Private Mobile Radio network. Generally, when a casual relationship between two variables can be established, a Bayesian network can be used to infer the probability of the value of a variable conditioned on the value of others.

2.3 Software Defined Networks

In this section, we describe the technologies used in the implementation of the SDN used in the system proposed. First, we define the protocol used in the communications between controller and nodes in Section 2.3.1. Then, we define the virtual switches used as nodes in our system in Section 2.3.2.

2.3.1 OpenFlow

The SDN paradigm is based on the migration of intelligence from network nodes into a separate entity called "network controller". This entity takes on the tasks of designing and implementing network policies that allow for a correct functioning of the network. In order to do this, the network controller must have a way of communicating with the nodes within a network, the same way the nodes need a protocol to reach the network controller whenever new traffic policies are needed.

The protocol that standardises such communications is known as the OpenFlow protocol. This protocol is adapted to the structure of traffic policies in SDN networks, so it can provide access to the forwarding plane of SDN nodes for both monitoring and configuring, thus enabling the SDN paradigm. This protocol also allows for the network controller to communicate with nodes from different vendors. In order to allow this, OpenFlow requires a specific switch architecture, which can be seen at Fig. 2.1.

2.3.2 Open vSwitch

The elements tasked with routing data in SDN are known as "switches". However, in SDN, the barrier between network routing elements operating at level three of the OSI model

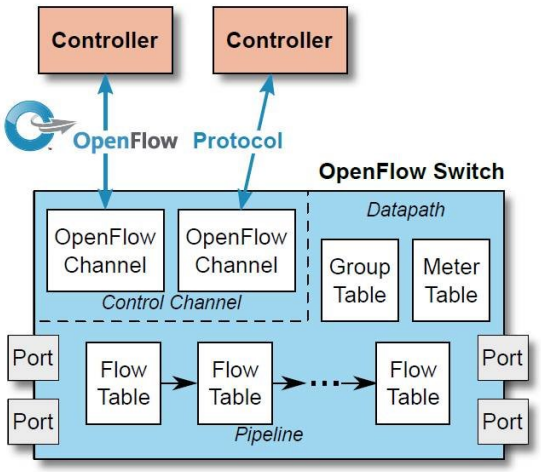

Figure 2.1: Architecture of an OpenFlow switch

(such as legacy routers) and network routing elements working at level two (such as legacy switches) disappears, since all the intelligence on routing policies and status of the network is hold in the control plane, and the network routing elements' task is to route according to a set of rules provided by the controller, independently of the OSI level associated to each rule.

Therefore, whenever we mention routing elements as "switches", it must be understood that we are not referring to the "legacy" conception of switch (a level-two routing element) but to an element that merely implements rules on every OSI level.

However, since we do not run a real, physical SDN network, but a computer simulation of one, we do not use real switches. Instead, we use virtual switches. Particularly, we use Open vSwitch [24], an open-source virtual switch that supports OpenFlow-based communications. An image of the architecture of Open vSwitch can be seen in Fig. 2.2.

As we can see, the core of the switch element in our network is the Openvswitch kernel module. In this kernel, multiple "datapaths" (similar to bridges) are implemented. Each datapath can have multiple "vports" defined (similar to ports within a bridge).

In the userspace, a daemon ("ovs-vswitchd") is created in order to control all Open vSwitch switches defined in the local machine. This daemon is the only access point to the Openvswitch kernel module. It communicates with the Open vSwitch database server ("ovsdb-server"), which provides interfaces to multiple Open vSwitch databases ("ovsdb"), where multiple configuration variables (such as Flow Tables) are stored. In order to manage these databases, a command-line tool is defined ("ovsdb-tool"). Finally, multiple programs to manage datapaths ("ovs-dpctl"), running Open vSwitch daemons ("ovs-appctl"), daemon

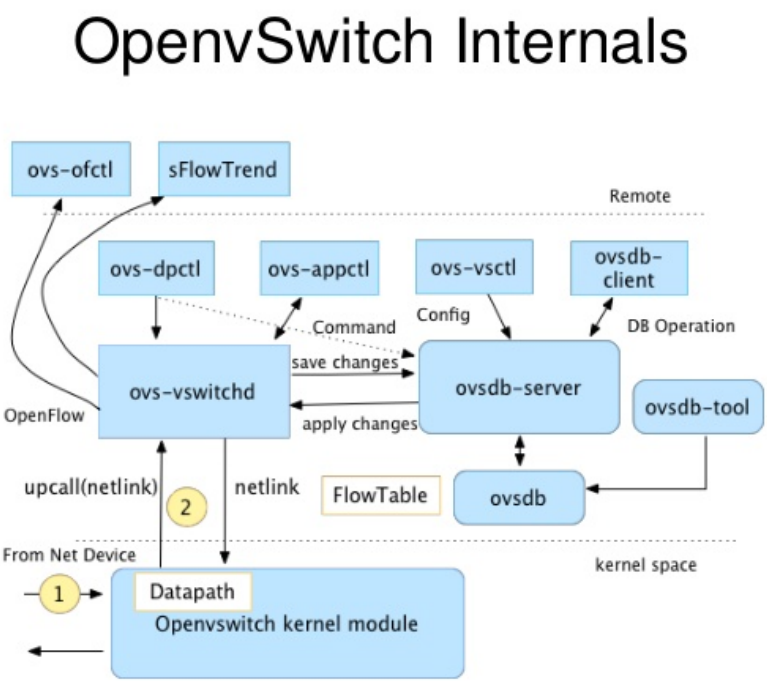

Figure 2.2: Architecture of an Open vSwitch

configuration variables stored in the database ("ovs-vsctl"), and the Open vSwitch database server ("ovsdb-client") are defined.

Each switch communicates with the SDN controller through an interface connected to a specific port in the machine hosting the controller, where the OpenFlow protocol is supported. Therefore, whenever the switch does not have an entry in any flow table that matches a packet received through any of the vports from other vswitches, this packet is sent in an OpenFlow message to the controller. Then, the SDN controller sends through another OpenFlow message with a flow rule for that package.

2.4 Fault Diagnosis

In order to take on the task of fault diagnosing, we can consider the approach provided by Benjamins et.al. [25]. In this paper, they propose the decomposition of tasks into subtasks by using a Problem-Solving Method (PSM). For example, we define the Fault Diagnosis Method as a method which divides the fault diagnosis task into three subtasks.

The first of these subtasks is the symptom detection task. We perform the monitoring of the variables using a "classification" approach. Depending on the value of the observed variables, we may classify the variables vector as a symptom or not.

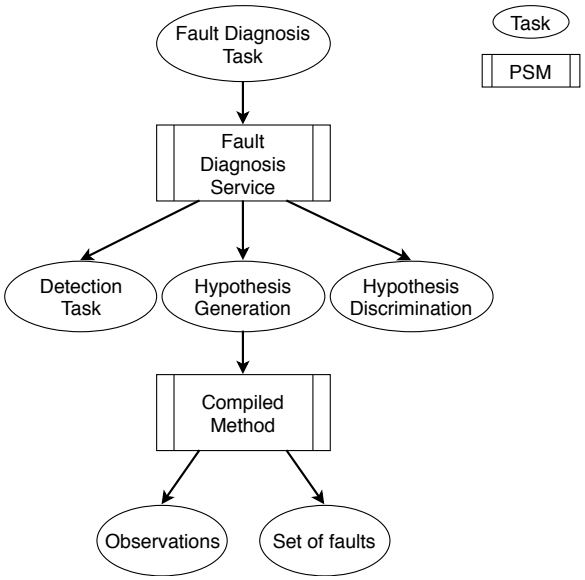

Figure 2.3: Use of PSM in the Fault Diagnosis Task

Then, regarding the hypothesis generation, we have followed a "Compiled Method" approach. In this method, we obtain a series of abstracted observations from raw data, and we select some possible faults that could provoke those observations.

At the end, we have a set of possible faults that provoked those observations. Finally, we perform hypothesis discrimination. In this step, we select the most probable fault according to the value of the observed variables, discarding the rest.

In order to perform the three steps mentioned before (detection, hypothesis generation and hypothesis discrimination) we make use of Bayesian reasoning. By applying Bayesian reasoning, we can detect entries that do not comply with a fault-free status of the network (detection tasks). Then, we assign a probability of reflecting each fault in the network to each entry (hypothesis generation). Finally, we select the fault with the highest probability as the final decision on the diagnosis of the network (hypothesis discrimination).

The functioning of Bayesian reasoning is explained in Section 2.4.1. Then, in Section 2.4.2, we describe the machine learning technique used in the learning of Bayesian networks.

2.4.1 Bayesian Reasoning

Bayesian reasoning consists in the use of Bayesian networks in the modelling of causal relationships in order to perform Bayesian inference and be able to predict the probability of certain value in a variable according to some evidence.

In order to describe the functioning of a Bayesian network, first we must explain the concept of "probabilistic reasoning". When we have a set of variables that describe our environment, and we want to 'reason' over them (to find a model that properly explains the functioning of our environment) we can use several approaches. For example, we could implement a set of hard-coded rules to infer values for each variable: this is known as "rule-based reasoning". On the contrary, we could also use real data obtained from the environment in order to find conditional probability relationships between the values of the variables. This is known as "probabilistic reasoning".

Once we have chosen probabilistic reasoning, we focus now on the scenario we want to reason over. A common relationship between the variables of a scenario is the one defined by X $\rightarrow$ Y, where X is a non-observable event, Y is an observable evidence, and the arrow shows that Y depends on X. In such cases, it is useful to find the probability P(X|Y=y), since it allows us to infer the value of X given the observable Y. In order to find this probability, we use Bayes' Theorem:

$$P(X|Y) = \frac{P(X,Y)}{P(Y)} = \frac{P(Y|X)P(X)}{P(Y)} \tag{2.1}$$

where P(X|Y) is known as *posterior probability*, P(X|Y) is the probability of observing Y given the event X and is known as *likelihood* for a given x$\in$X, P(X) is the probability distribution of X, it is known as *prior probability* and it represents our prior knowledge on the scenario, and P(Y) is the probability distribution of the evidence. If P(Y) can not be known, we can omit it, thus obtaining the *unnormalized posterior probability*.

We can immediately see the usefulness of this theorem: given observable evidences in our scenario, we can infer the status of unobservable variables. Probabilistic reasoning based on this idea is known as *Bayesian inference*. We can see that it fits our problem, since we want to infer the unobservable status of the network given some observable variables. However, Bayesian inference has a considerable drawback: there is not any criteria on choosing the prior probability. Therefore, selecting an arbitrary prior probability that wrongly represents the situation could severely hamper the reasoning process.

Now that we have introduced the concept of Bayesian reasoning, we can describe Bayesian networks. A Bayesian network is defined by two elements. First, a DAG is created. In this graph, all variables involved in the reasoning process are represented. In order to show the conditional relationships between them, these variables are connected using directed links (hence, the "Directed" in DAG) where the variable in the end point of the link depends on the variable in the opposite side of the link. Once we have represented

conditional relations using a DAG, we must now represent how these variables depend on each other. This is done by associating a Conditional Probability Table (CPT) to each variable, where the probability of every possible value of each variable depending on every possible value of its "parent" variables is shown.

Given these two elements, and using Bayesian inference, we can infer the value of any variable in the network. An example can be seen at Fig. 2.4 and Table 2.1.

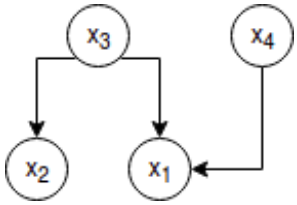

Figure 2.4: Simple DAG with four variables

$x_3 = 0$	$x_3 = 1$
0.8	0.2

$x_4 = 0$	$x_4 = 1$
0.8	0.2

x_2 \ x_3	0	1
0	0.9	0
1	0.1	1

x_1 \ x_3, x_4	0,0	0,1	1,0	1,1
0	0.9	0	0	0
1	0.1	1	1	1

Table 2.1: CPTs for the previous network

For example, if we want to infer the probability of x_1 being 1 given the value of x_4 is 1 (that is, finding p(x_4= 1| x_1= 0)), we would need to find

$$P(x_4 = 1 \mid x_1 = 1) = \frac{p(x_4 = 1, x_1 = 1)}{p(x_1 = 1)} = \frac{\sum_{x_2,x_3} p(x_1 = 1, x_2, x_3, x_4 = 1)}{\sum_{x_2,x_3,x_4} p(x_1 = 1, x_2, x_3, x_4)} \tag{2.2}$$

Now, using the chain rule of probability and the conditional independencies expressed in the DAG, we can simplify the expression

$$\frac{\sum_{x_2,x_3} p(x_1 = 1, x_2, x_3, x_4 = 1)}{\sum_{x_2,x_3,x_4} p(x_1 = 1, x_2, x_3, x_4)} = \frac{\sum_{x_3} p(x_1 = 1 \mid x_3, x_4 = 1)p(x_3)p(x_4 = 1)}{\sum_{x_3,x_4} p(x_1 = 1 \mid x_3, x_4)p(x_3)p(x_4)} \quad (*) \tag{2.3}$$

$(*) : given f(x), \sum_y p(y \mid x)f(x) = f(x)$

Now we substitute the values stored in the CPTs. Thus, we have

$$P(x_4 = 1 \mid x_1 = 1) = 0.132 \tag{2.4}$$

2.4.2 Machine Learning for Bayesian Models

We have seen how, using a Bayesian network, we can infer the value of a variable. However, we have supposed that both the DAG and CPTs are given. In the real world, as in the case of network diagnosis, the adequate Bayesian network model is not known. Therefore, we may have to infer the best choice for the given data. In that case, we must use some algorithm to "learn" them from data obtained from the environment we want to model.

This can be done by using structural learning techniques, which allows us to find a DAG that accurately represents the conditional relationships within the data provided. Then, CPTs can be filled using the statistical structure of the dataset used for learning. One type of structural learning techniques family is the one known as *score-based learning*. In score-based learning techniques, the objective is to find the DAG that fits best the data according to a scoring system. Depending on the scoring function, we can define a wide variety of structural learning techniques.

In order to learn the DAG of our Bayesian network, we have selected a score-based learning algorithm known as *Bayesian Search*. This algorithm begins with a random DAG, and, following a hill climbing procedure, searches the full space of one-step changes to the current DAG, and selects the DAG with the highest score.

A variety of functions can be used as a scoring function. However, a common approach is to use the accuracy as a scoring function. This scoring function measures the accuracy of the proposed model in classifying each entry of the dataset provided according to a target variable. In order to prevent overfitting, we apply 10-fold cross-validation in the evaluation process.

When all proposed changes have been evaluated, if there is not any change that improves the accuracy of the current proposed model, the hill climbing process ends. If there is a change that improves the score, the process is repeated with the new model. This process is repeated until a predefined number of iterations is reached.

One important drawback of this algorithm consists in the heavy dependence on the initial random DAG used as first step. Therefore, when the maximum number of iterations is not high the space of possible DAGs searched is very limited. In order to prevent this, we use the Bayesian search algorithm with a predefined number of random restarts. Thus, when the hill climbing process ends, we store the highest-scoring model, and then we "restart" the algorithm, creating a different initial random DAG. This allows to broaden the space of models tested, and reduce dependence on the initial model.

2.5 Network Simulation

In this section, we will describe some technologies used in the creation of the network simulation. First, in Section 2.5.1, we depict the traffic generated in the network. Then, we explain the topology implemented in Section 2.5.2.

2.5.1 Traffic Generation

We would like to create a network simulation that resembles computer networks that we find in real life. In order to do this, we should not just create a simulation with connectivity between all users. We should also create network traffic, so we can test how faults affect services provided within the network.

Therefore, the first service that we are going to create is a streaming video service. This helps us in creating a constant flow between the hosts where the service is provided and the users. The streaming service will be implemented as a point-to-point service.

Next, we would like to implement small, "chat" traffic between the end users. End users will listen for messages sent randomly by other users in the network, and they will reply to them. As a result, small Transmission Control Protocol (TCP) packages will be travelling from one end user to another within the network. In a similar fashion, Peer-to-Peer (P2P) traffic has been defined. End users will be sending audio files to each other randomly.

Also, web traffic will be implemented. End users will connect to ports in certain hosts where a web page will be provided to them. Therefore, we can simulate the effect of faults in the provision of web service. Finally, we will also implement a mail service. Certain hosts will listen for mails sent by end users, and will also send mails to other end users. Finally, a broadcast service will also be created in the network simulation.

2.5.2 Scale-free Networks

A scale-free network is a connected graph where each node has a number of links originated from it that exhibits a power law distribution.

$$P(k) \propto k^{(-\gamma)} \tag{2.5}$$

where k is the number of links originating from that node, and γ is some exponent that controls the speed at which $P(k)$ decays as k increases. The number of links adjacent to a

node i, k_i, is also known as the *degree* of a node.

In order to create a scale-free network, first we start with a small number of connected nodes n_0 following the Power Law previously mentioned. Then, each new node is added with $n < n_0$ links to the already existing nodes. When choosing which nodes to connect the new node with, the concept of "Preferential Attachment" is followed. This concept explains that the probability of a new node to be connected to an existing node i with a degree k_i is

$$P \propto \frac{k_i}{\sum_i k_i} \tag{2.6}$$

As we can see, the higher the degree of a node is, the higher the probability of a new node connecting to it. The power law and preferential attachment rules tend to create nodes with many links, known as "hubs", where is most probable that new nodes will be connected. Emerging from those hubs, there are many trailing tails of nodes with progressively fewer connections.

Scale-free networks present multiple features: for example, due to its nature, zooming in any part on the network does not change its shape; we will always see the structure described in the previous paragraph. This structure is also fault-tolerant: if an error occurs in a random node, the vast majority of them have few connections. Even if a failure occurs in a hub, the remaining hubs will maintain the connectivity of the whole network.

However, these are not the reasons why we are interested in scale-free networks. The concept of scale-free networks originated from a research project by Barabási et al. [26]. While researching in the topology of complex networks, they used a Web crawler to map the topology of the World Wide Web (WWW). They expected to find a random connectivity model, in accordance with the then-accepted random graph theory of Erdős-Rényе. However, they found that the topology of networks within the WWW did not conform to this theory; instead, they followed the topology described by scale-free networks. Since computer networks follow the scale-free model, we will do so too.

CHAPTER 3

Architecture

In this chapter, we outline the architecture proposed in this project. First, we define a layer of possible data sources from which we can learn and diagnose the status of the network. Then, we outline a layer for the development of data connectors tasked with collecting data from the data sources and storing it. In order to store it, we use a Data Lake, which is a module that stores raw data. Next, we define a processing module in order to adapt the data so as to ease the learning of the causal relationships within it. Then, we design a reasoning module which performs the diagnosis. We also provide some insight in the sequence of collecting and processing a batch of data, and we detail the activities involved in the diagnosis process.

3.1 Introduction

First, in Section 3.2, the architecture of the system proposed in this paper is presented. Next, the modules involved in data management are explained in Section 3.3. Finally, in Section 3.4, we depict the modules involved in fault management.

3.2 Architecture Overview

The general architecture of the system is shown in Fig. 3.1. At the base of our architecture lies the SDN, which is the object of our diagnosis. Then, we create a set of modules over it that collects, stores and process data extracted from different sources related to the network. Finally, we create a fault diagnosis module; it takes on the task of diagnosing the network.

We can make a distinction in our architecture between two types of modules: data modules and fault management modules. Data modules are designed for the purpose of implementing the necessary tasks in the managing of the data in our system: collecting, storing and processing. On the other hand, the fault management modules are designed to provide the ability of managing faults.

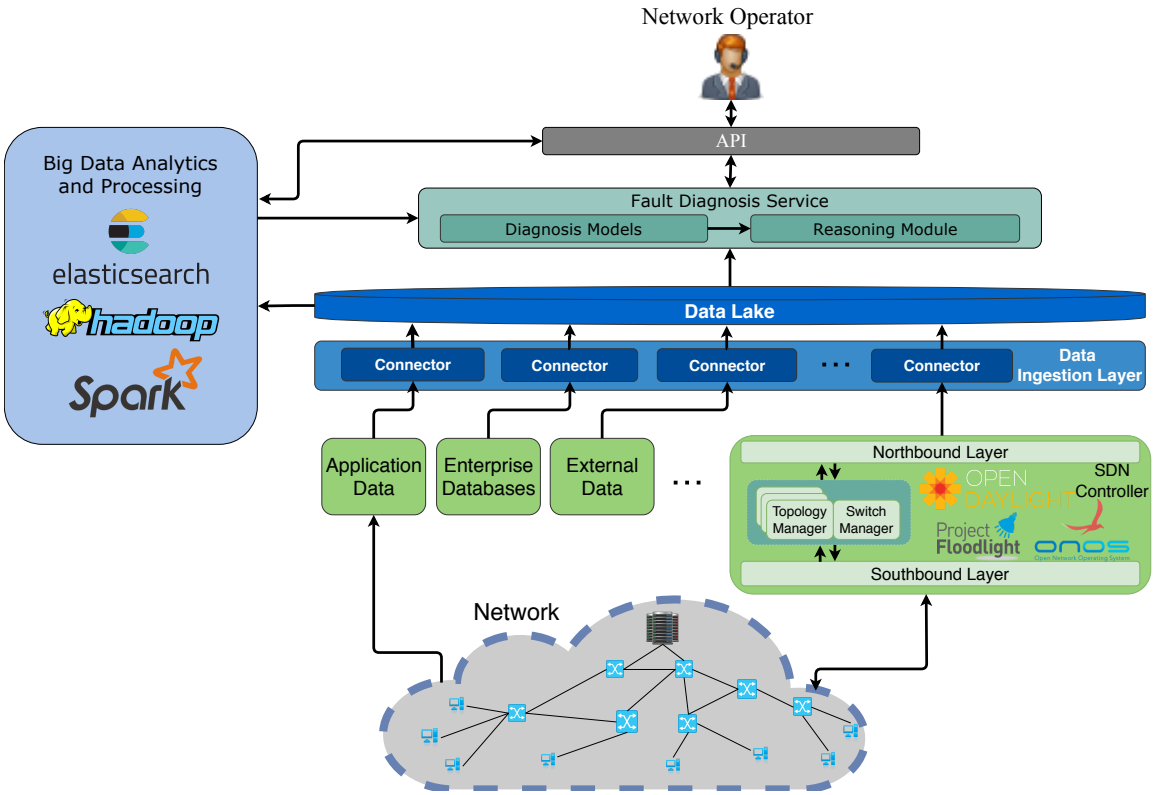

Figure 3.1: Overview of the general architecture

Following the distinction previously mentioned, we can describe each module in Fig. 3.1 as a data-managing or fault-managing module. Specifically, the Fault Diagnosis Service module takes on the task of diagnosing the presence of faults; therefore it is fault management module. The rest of the modules implement functions related with data management: the data sources, such as the SDN Controller, hold information on the status of the network.

Such information is collected by Connector modules within the Data Ingestion Layer. The collected data is then stored in the Data Lake module. Finally, the Big Data Analytics and Processing module takes on the task of processing the data downloaded from the Data Lake before sending it to the Fault Diagnosis Service module.

3.3 Data Management

In this section, modules involved in management of the data are explained. First, in Section 3.3.1, we comment on the possible data sources that could be used in our system. Then, we describe the data collecting and storing in Section 3.3.2. Finally, we explain the data processing in Section 3.3.3.

3.3.1 Data Sources

For the purpose of performing reasoning, first we need both evidences and a Bayesian network. And, in other to learn model, we need data for both training and evaluation of the model created. Therefore, we need to select data sources, having in mind that while we could take a quantitative approach and collect as much data as possible, we are more interested in a qualitative approach. We would like to collect data that holds causal relationships between its values and the situations we want to diagnose.

In order to do this, we can select from a myriad of data sources. The first data source that we can consider is the network controller. Since it holds the intelligence of the network environment, it could be considered as the primary and most important source of data for our diagnosing and healing system. It communicates directly with each network element, so it is able to have an updated knowledge on the status and statistics of each element.

However, we could also use other types of data sources. For example, we could use data sent by the apps running in the user's device, in order to detect faults in traffic rates that could lead to the discovery of a faulty link or an unoptimised traffic policy. This data could consist on traffic rates, statistics on the types of packages, or reports on the status of last-mile links.

Another data source that could be considered is user feedback. Depending on the type of problem reported by end users, we could obtain information on what type of fault happened in our network. We could also cross-reference this data with the data obtained from the app in order to help in valuing the importance of such reports.

Even though some possible data sources are presented in this section, the flexibility of this architecture allows for the use of many other data sources, such as historical databases, enterprise databases or other kinds of external data sources.

3.3.2 Data Ingestion Layer and Data Lake

Once we have selected the data sources, we must now find a way to collect data from them. Therefore, we define the Data Ingestion Layer. This layer takes on the task of collecting data from each source and storing it into the Data Lake.

Since each data source offers a different way of accessing the data, and each data type needs a different mapping, we need to define a different connector for each data source. The connectors could collect data periodically from the source when possible, in batches, or they could also do it in streaming, in order to keep a constant flow of data on the status of the network running.

Once we have collected the raw data, we need a place for its storage. Therefore, we define the Data Lake. The concept of the Data Lake consists in having a place where we can store any kind of raw data, independently of its origin, structure or purpose. Hence, we can store all the information that we have on the current status of the network in one place.

We could use directly this raw data as training data for the creation and parameterisation of the Bayesian network. However, raw data usually tends to be noisy, it usually has useless information and can hold fake causal relationships. Therefore, the next step should be processing the data.

The functioning of each data connector is controlled by the Data Ingestion Layer. This functioning is described in Fig. 3.2. Each connector receives the order from the Data Ingestion Layer to start collecting batches of data. Then, depending on the content of those batches, we are able to detect a failure in the data sources; in that case, we also store that information, since it could be useful in the reasoning process. Then, we test the connectivity to the Data Lake. If the Data Lake is reachable, we store either the collected data or the report on the status of the data source. Finally, we start collecting data again.

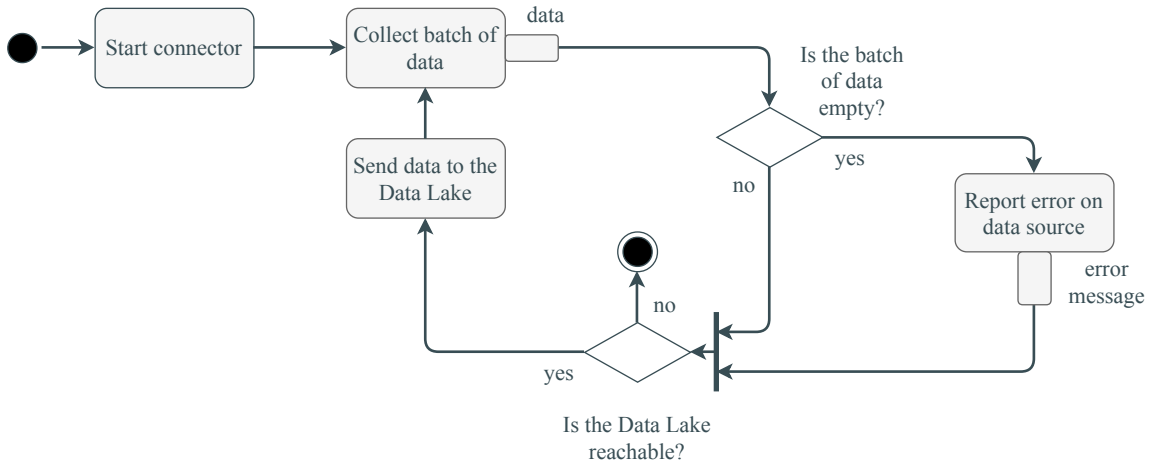

Figure 3.2: Activity diagram describing the functioning of the data connectors

3.3.3 Data Analysis and Processing

As commented previously, data must be processed in order to be used in the supervised learning process of the Bayesian network and in the reasoning module. Moreover, raw data contains much non-relevant information for the reasoning process. That information could be not only useless, but also pernicious, since it could lead to wrong reasoning processes with a lot of noise and false causal relationships. Also, some variables may have missing values due to network failures altering the ability to monitor them. To avoid such cases, historical data could be used as knowledge base.

Therefore, independently of the data source and the type of data, collected data must analysed. Big Data software platforms, such as Elasticsearch, Spark or Hadoop, can be used to index and classify high volume of collected data in order to simplify further processing. This would also facilitate conversion to multiple formats, which would enhance collaboration with external data units and diagnosis modules, improve general flexibility, and ease the running of analysing tasks.

Such tasks could consist in noise reduction, treatment of missing values, time-based analysis or mitigation of fake causal relationships within the data. This processing could be done in streaming or by batches, independently of the way data is collected and stored. Once we have the processed data, we are ready to learn a Bayesian network based on this data or to use it to run the reasoning module.

We can see in Fig. 3.3 how the data processing module is involved in the managing of the data. Specifically, the data processor periodically sends queries to the Data Lake in order to obtain data. Then, this data is processed and sent to the fault diagnosis service in

order to perform diagnosis. This process is executed in parallel to the process of collecting the data, performed by the data connector. As we can see in Fig. 3.3, the data connector sends queries to the data source. This data source then process each query and sends the requested data to the data connector. Finally, the data connector sends the collected data to the Data Lake for its storing. In the next iteration of the data processor, this data will be picked up by the data processor module for its processing.

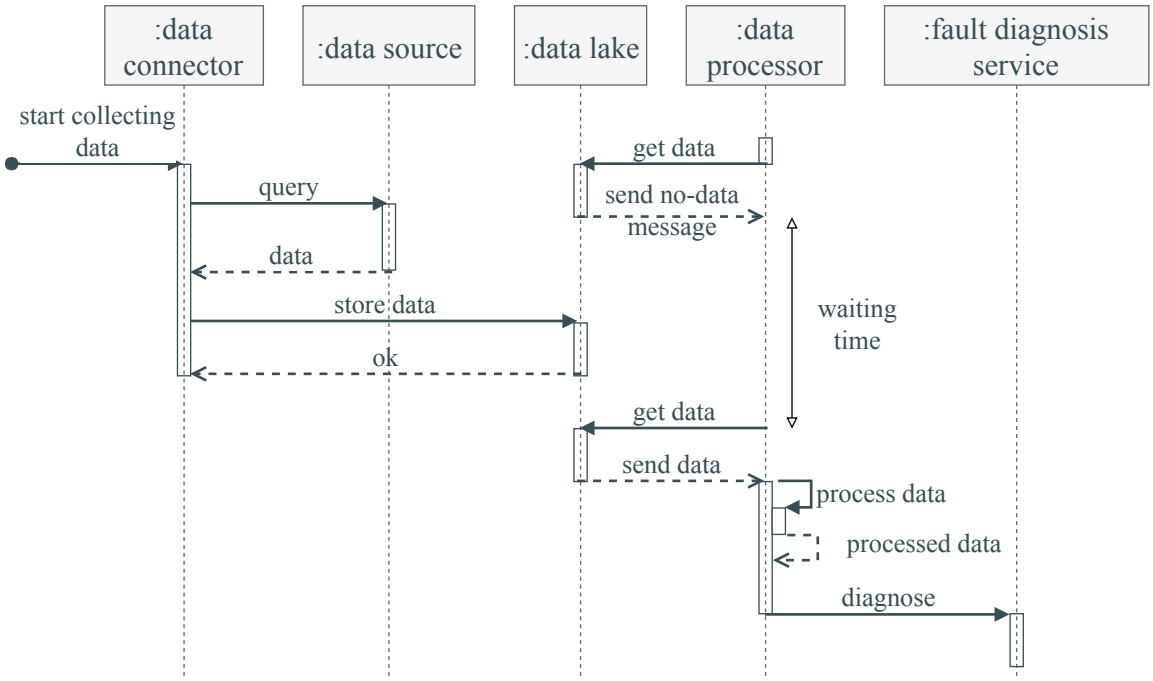

Figure 3.3: Sequence of collecting and processing a batch of data

3.4 Fault Management

In this section, the process of fault management using the data previously collected and analysed is described. First, in Section 3.4.1, we describe the process of learning a model that is able to diagnose our network. Then, we describe the use of this model in Section 3.4.2.

3.4.1 Learning from Network Faults

Once we have explained the data collecting, storing and processing, we reach a point where we have enough data in order to learn a Bayesian network and perform reasoning. In order perform learning, we need "labelled" data. labelled data is composed by entries that have been tagged with the status that they represent.

If we apply machine learning algorithms using these entries, we can model causal relationships between the values of each feature in an entry and the status that this entry represents. This type of learning is known as "supervised learning". Once we have performed supervised learning, we obtain a set of causal relationships that can be implemented in a Bayesian network. As we have seen in Section 2.4.1, a parameterised Bayesian network allows us to encode conditional dependencies between features, and its conditional probabilities.

3.4.2 Fault Diagnosis

Once we have a parameterised Bayesian network, we implement it in the reasoning module, which takes data provided by the processing module and uses it as evidence to perform the diagnosis of the status of the network.

Considering the conditional relationships and probabilities, we are able to assign a probability to each possible status depending on the data provided. As a result of this process, we obtain a set of hypothesis. Then, by performing hypothesis discrimination, we are able to obtain the most probable status of the network. In the hypothesis discrimination step, we could just select the most probable status, or we could also consider other variables, such as historical data on past faults, or human intervention. The diagnosis status is then sent to the network operator.

In Fig. 3.4, we can see how the fault diagnosis activity could be performed depending on the presence of both a Bayesian network and labelled data. Following the principles defined in Section 2.4, we can divide the diagnosing process in three steps: detection task, hypothesis generation and hypothesis discrimination. In order to perform the diagnosis, we need to generate a model first.

In order to generate the model, we need labelled data for the purpose of performing supervised learning. Once we have labelled data, we can generate a Bayesian network model so as to use it in the hypothesis generation and discrimination process. In the hypothesis generation process, we analyse the data with the aim of obtaining abstracted knowledge on the network. This knowledge will help us in generating a set of possible faults that could be affecting the network.

Now that we have a set of possible faults, we perform hypothesis discrimination. In order to do this, we need to define a probability for each possible status of the network. According to the probability of each status, we decide on which hypothesis is the most probable. However, if such probability does not surpass a threshold, we need to run more

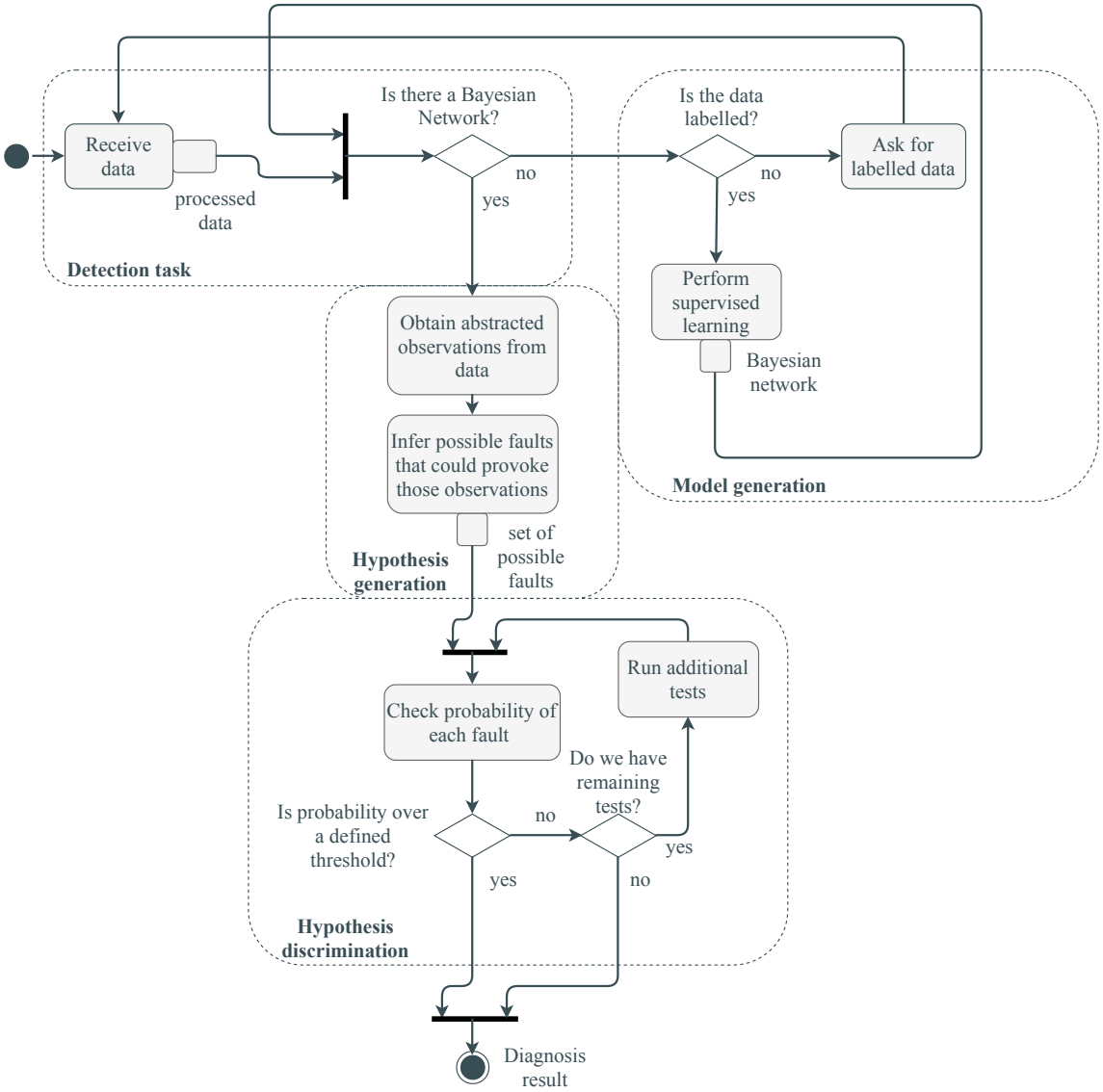

Figure 3.4: Activity diagram of the diagnosing process

tests on the network to obtain new data that could help us in the hypothesis discrimination process. With this data, we update the probability of each fault.

Once we find a fault with a probability that surpasses the defined threshold, we select that hypothesis as the correct one. However, if we do not surpass such threshold and there is not any test left to run, we ignore the threshold condition. Either way, the diagnosis process ends once we have selected a specific fault as the final hypothesis.

CHAPTER 4

Prototype

Once we have depicted the architecture in Chapter 3, we describe the implementation of such architecture. We have developed a network simulation module with the ability to lay out testbeds for testing our diagnosis service. Then, we have implemented a data connector that collects data from the network controller and stores it in the Data Lake, which has also been implemented. Next, we created a processing module based on Python scripts that collects data from the Data Lake. Finally, we implemented the Fault Diagnosis module, where we learn the Bayesian network and use it to perform inference.

4.1 Introduction

In this chapter, we are going to detail the implementation of the prototype for our diagnosis system. First, we are going to outline the architecture of the prototype implemented in Section 4.2. Then, in Section 4.3, we describe the design and implementation of the network simulation used as a testbed. Next, we depict the modules involved in the managing of the data in Section 4.4. Finally, in Section 4.5, we are going to portray the functioning of the reasoning module.

4.2 Deployment

We have developed the architecture that can be seen at Fig. 4.1. We have used Docker in the deployment the architecture, due to its ability to provide reproducibility and isolation of the environment where each module would be running, without the hardware requirements of a Virtual Machine (VM). Furthermore, we have used Docker-Compose, a tool that takes on the task of coordinating the deployment of multiple Docker containers simultaneously.

We have focused on using the network controller as the data source for information on the status on the network, and we have chosen Opendaylight as the network controller. The Data Lake has been implemented using Elasticsearch in order to allow for a flexible indexing and storing of the data.

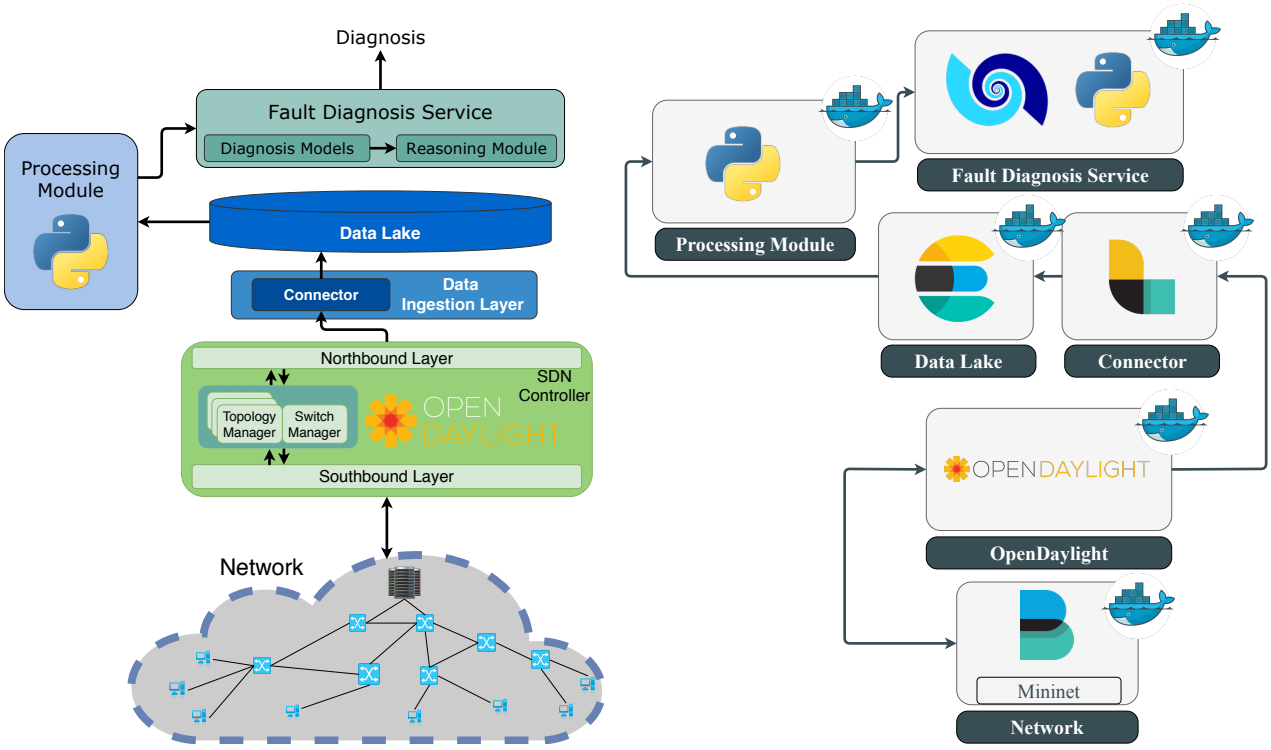

Figure 4.1: Overview of the prototype architecture (left) and its implementation using Docker containers (right)

Once we have stored the data, we can start the processing module; this module collects data from the Elasticsearch database and processes it, saving the processed data in Comma Separated Values (CSV) files that will be used by the reasoning module as evidences in order to perform reasoning. This processing module has been implemented by developing Python scripts that perform the necessary steps in order to obtain the data that we are

interested in.

Finally, in the reasoning process, using a Bayesian network that we have previously learned using similar data, we perform inference to find the probability of the network being in each status. Then, we obtain a diagnosis on the network based on the probability of each entry representing a status.

4.3 Network Simulation

In this section, the implementation of the network simulation from which we collect data is described. First, we describe the topology selected for the creation of simulated networks in Section 4.3.1. Then, in Section 4.3.2, we show the implementation of the traffic simulation mentioned in Section 2.5.1, Finally, we describe the design and implementation of the faults introduced in the network in Section 4.3.3.

4.3.1 Topology

We have shown in Section 2.5.2 that real-life network topologies follow scale-free models; however, this alone is not enough to design a network that resembles real life telecommunication networks. Besides a scheme of connection between nodes inspired in scale-free networks, a higher-level architecture must be implemented.

This higher level is based on the notion that telecommunication networks usually present a central, highly interconnected nucleus of network elements that takes on most of the traffic within the network. This "core" network usually holds connections to multiple datacenters that provides different services. Then, smaller, "access" networks are connected to this core; these smaller networks act as access points for users. Each access network is connected to the core network by multiple network elements interconnected with both networks considering availability issues.

Once we have selected the guidelines that we will follow in the implementation of the topology, we can create the network. We have implemented the network through the use of the Mininet Python API, which allows for the configuration of many details of the network, from its topology to the type and configuration of its elements.

The detailed topology of our network is based on the Scale-Free topology family. Therefore, the first step consist in creating a random (if no seed is provided) scale-free network. To do this, we use Networkx [27], a Python module that automatises the creation of scale-free

graphs based of a set of parameters, such as number of nodes.

Using this module, we create a graph that will be the basis of the core network, and then a new graph for each access network connected to the core. However, this is not enough, since the Mininet Python API needs an object of the "Topo" class representing the topology that is to be created.

Hence, we create an empty Topo object for each network, and iterate over the elements of each graph, creating a Switch object for every node in the graph, and a Link object for every edge (particularly a TCLink object, since it allows us to take traffic control measures, hence the TC prefix). This process must be done carefully, since both the core network and any access network attached share the same namespace.

In the case of core networks, an extra processing step is needed. Access networks usually have switches with only a single connection to another switch. This is due to prevision of demand growth by marketing departments: extra switches single-connected are laid out in prevision that demand will grow in that area. However, this is not the case in core networks: every switch must be connected to, at least, other two switches, since any switch in the core network have the sole purpose of routing between switches, not acting as an access point for users. Therefore, a "trimming" operation is needed, where any switch left single-connected is connected to another switch within the core network. The result of such process can be seen in Fig. 4.2.

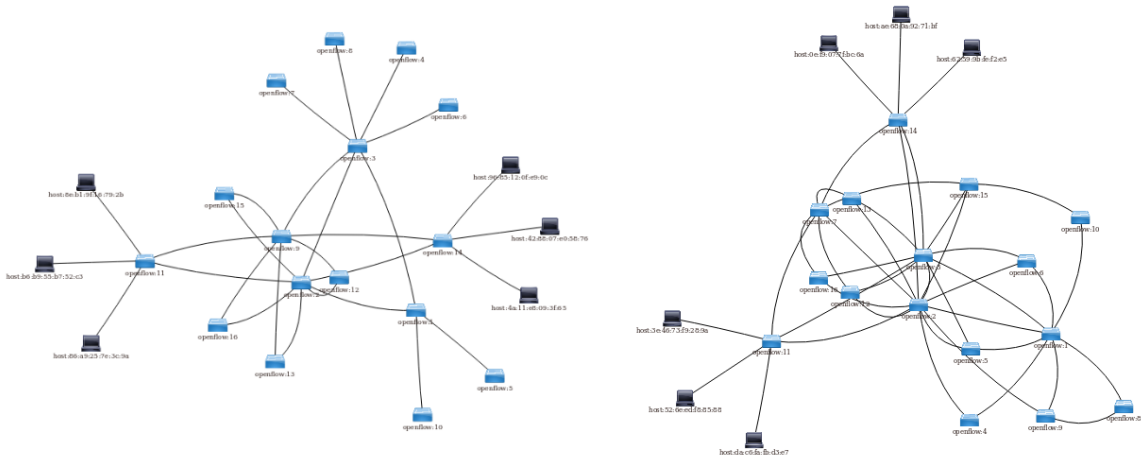

Figure 4.2: Untrimmed (left) and trimmed (right) core networks

Then, for each access network, a configurable number of hosts are added. These hosts will act as users in our network, generating traffic between themselves and either other users or the datacenters. Also, a predefined number of groups of three hosts will be added to the core network. These hosts will act as datacenters, where some services will be offered.

Finally, each access network is connected to the core network by a variable number of switches. These switches are connected to both the core and the access networks in such a way that provides high availability against single points of failure.

After creating the topology, we must specify the SDN Controller that will manage the network created. This step is very important, since the SDN Controller will be the source of information on the network status, and depending on the choice of controller the data received will vary greatly in both structure and information contained. Therefore, the processing step would have to vary greatly too if we want to keep using similar reasoning models.

Once a controller has been added to the simulation, we ping every pair of hosts (both datacenter and user hosts) to check full connectivity within the network. In this process the controller creates the network policies that will ensure the correct functioning of this network. Since, just after the creation of the network flow tables within switches are empty, every ping package that arrives to a switch is sent to the controller, where the module tasked with routing management uses a routing algorithm to create the rules that will allow communications between the sender and the receiver of such package. Then these rules are pushed into the switches.

4.3.2 Traffic Simulation

For the purpose of mimicking real life networks, some services must be implemented, as we discussed in Section 2.5.1. This will not only test the correct implementation of the simulation, but also it will provide a "testbed" where the impact that faults have over the provision of services within the network can be assessed. The services will be mostly provided by a number of datacenters connected to the core network. Users will be connected to these datacenters, generating traffic within the network.

First, the streaming service mentioned in Section 2.5.1 is implemented. A sample video stored in the datacenters is sent to users subscribed to that service through the Real Time Streaming Protocol (RTSP). This protocol is an application-level, point-to-point streaming protocol that controls media sessions between end points. Since this protocol is designed to control multiple data delivery sessions, the process of establishing a streaming service within the network is eased. We just need to focus on make it available in order for the users to connect to datacenters and establish a streaming session. This is done by opening a port in the datacenter where users connect to. We use the implementation of the RTSP offered by VLC, a tool for video and audio streaming and processing described in Section

C.4.

RTSP uses Real Time Transport Protocol (RTP), another application-level protocol, as a transport-assisting protocol. This protocol provides multiple functionalities to the transport of data streams, such as coordination between multiple data sources or data synchronization control tools. In this case, User Datagram Protocol (UDP) is used as the transport-level protocol; however, TCP could be used too.

Regarding the chat traffic mentioned in Section 2.5.1, a sample TXT file containing a small message is used as a data source. Then, this message is sent using Socat to establish a TCP connection with another random user within the network every 60 seconds.

Within the connection previously mentioned, the message is sent. All hosts in the network are configured to listen for TCP connections in a specified port associated to this chat-like traffic. In a similar way, P2P traffic has been also included, sending a small audio file between pairs of hosts within the network.

Also, a "fake" Hypertext Transfer Protocol (HTTP) server has been implemented within the datacenters in the core network in order to provide web services. Again, using Socat, we listen to TCP connections to a specific port within the datacenter, associated to web traffic. For each connection received in this port, a child process is created to handle the connection. In this child process, a response is sent following the typical scheme for HTTP responses: an HTTP 200 OK message, a message with the Content-Type variable, and a sample document written in Hypertext Markup Language (HTML).

The e-mail service has also been implemented. A server has been implemented using Socat to listen to TCP connections in ports associated with mail traffic. For each connection, depending on the port and the content of the message from the client side, either a Simple Mail Transfer Protocol (SMTP) message or a file representing an e-mail is sent.

Finally, asymmetric broadcast traffic has been simulated within the network, in order to emulate a live TV broadcasting service. This is done by using Socat in the hosts of the datacenters to send a file over the UDP protocol to a specific port of the network address 10.255.255.255, which is the default address for broadcast traffic within the network. When the client side receives a broadcast message, it responds with a message acknowledging the receiving of the file.

4.3.3 Fault Generation

Now we have a fully operational simulated network. Thus, the next step is to generate faults within it, in order to obtain enough data to learn a Bayesian network that models our SDN scenario. Specifically, the following possible statuses of the network are defined:

- **S0** - No faults - OK status.
- **S1** - Shutting down a node.
- **S2** - Disconnecting a datacenter server from the network.
- **S3** - Modifying the out-port rules in a node.
- **S4** - Modifying the in-port rules in a node.
- **S5** - Adding idle-timeouts in a node.
- **S6** - Adding hard-timeouts in a node.
- **S7** - Changing flow priorities in a node.
- **S8** - Forcing a node to drop LLDP packets.
- **S9** - Modifying both out-port and in-port rules in a node.

Faults are created by modifying current flow rules within a switch through request to the SDN Controller Representational State Transfer (REST) API (such as statuses S1 and S3 to S9) or by using the Mininet API (such as status S2) to modify the network topology.

4.4 Data Management

In this section, we will describe how the data collecting, storing and processing is implemented. First, in Section 4.4.1, we will explain the sources we have selected for the monitoring of the network status. Next, we explain how we have implemented the data collection module in Section 4.4.2. Then, in Section 4.4.3, we describe the configuring of the Data Lake. Finally, in Section 4.4.4, we detail the steps followed in the processing of the data.

4.4.1 Data Sources

Selecting the data sources is one of the most important steps of the designing process. It is vital for the correct functioning of the system that we collect data which holds causal relationships between its values and the faults we would want to diagnose. Therefore, in the case of the prototype, we would want to collect data from certain modules within

Opendaylight, selecting the ones that would show changes when certain faults happen within the network. Hence, we would want to collect data from the "network-topology" and "opendaylight-inventory" data models of the Opendaylight network controller, since these models contain information that will show changes in their values related to the faults described at Section 4.3.3.

The network-topology data model holds information on the elements that compose the network, the way they are interconnected and some basic knowledge on their physical status. On the other hand, the opendaylight-inventory model holds information on the configuration of each network node. Specifically, they hold all the information related to the configuration of node flow tables and its statistics.

While many other data models and sources could be used in order to provide more information in the searching of causal relationships during the supervised learning process, the data obtained from those sources would not add too much extra information, and could actually generate noise that could affect the reasoning process.

4.4.2 Data Collection

Once we have decided which sources we want to use in our model, and which information we want to collect in each source, we must now decide the method for collecting the data and storing it in the Data Lake. We have to design a data collector specifically designed for each data source, since each data source may present different ways of accessing to the data.

In our prototype, we take advantage of the northbound REST API that the Opendaylight network controller provides in order to collect the data. Through this API, we obtain access to all the data structures that have been defined in the controller by Yet Another Next Generation (YANG) models, representing the configuration and status of every element of the network.

Data in Opendaylight is stored in two different databases: the "operational" database and the "config" database. The operational database stores data that represents the current status of the network. Hence, we are going to collect data from this database. On the other hand, the "config" database holds data (usually in the form of node and traffic configurations) that we want to push into the network.

In order to collect the data, we create a collector module. This collector module monitors the log of the simulation running in the network module. Once it detects that a simulation has begun, it starts collecting data from Opendaylight by sending HTTP queries to its

REST API. Opendaylight replies by sending back data models in JSON format.

We monitor the simulation logs for the purpose of labelling the data collected from Opendaylight. We need labelled data in order to perform supervised learning of Bayesian network models, since labelled data allows us to find correlations between the data extracted from Opendaylight and the labels of each data entry.

The monitoring of the logs in the simulation is done by using Filebeat [28]. Filebeat is a tool that tracks a set of logs and is capable of shipping the log entries to another machine. We use Filebeat to track the log of the network simulation, and send its entries to the data mining module (the connector in the architecture). Then, at the connector's end , we use Logstash [29]. Logstash is a tool that allows us to collect log entries sent by Filebeat (or "beats") and perform multiple actions according to them.

In our case, when a new beat arrives at the collecting module, we index that entry in the Data Lake, and then we run queries in order to obtain data on the network topology, and on the configuration of each node in the network. All this data is also stored in the Data Lake.

4.4.3 Data Storing

Now that we have a module that takes on the task of collecting the data from the Opendaylight network controller, we need an implementation of the Data Lake. In order to do this, we have chosen Elasticsearch [30]. Elasticsearch is an open-source, RESTful, distributed search and analytics engine; we have chosen it due to its flexibility, the availability of Graphical User Interface (GUI) tools that allow for a graphical representation of the data, and the variety of integration tools that eases the process of interacting with other modules.

In Elasticsearch, we have mapped three types of elements: simstate documents, node documents and topo documents. Each one of them holds a different type of data: simstate stores log entries, node stores opendaylight-inventory query results, and topo stores network-topology query results. Once we have defined the mapping of the data, we run Elasticsearch. While Elasticsearch is running, we can push data entries (also called "documents") which are stored.

4.4.4 Data Processing

At this point, we have already extracted data from the data source and stored it in the Data Lake. We could skip the processing step, and use the raw data to learn a Bayesian model

that is able to diagnose the status of the network. However, this could lead to a faulty supervised learning process, and ultimately to a faulty Bayesian network that represents fake causal relationships.

Raw data tends to be noisy. Since we are working over an environment of fault diagnosis, due to these faults some values could be missing from the collected data. Depending on the data sources, we could also perform supervised learning over data that holds fake causal relationships. If we want to avoid this, we need to perform some data processing.

In order to do this, we have developed a processing module based on Python scripts. Specifically, we run queries in order to download the data from the Data Lake. Then, we store it in CSV files, and we perform data processing using them.

First, we discard useless data. For example, since each node has many flow tables, some of them could be empty or have outdated entries. Therefore, we need to discard those flow tables, in order to simplify the learning process. Once we have selected the variables we are interested in, we discretise their values. Finally, we perform some time-based analysis, in order to detect unexpected changes and add a level of abstraction to the data. Specifically, given a time window, we analyse the data looking for unexpected changes in the values of some variables.

4.5 Fault Management

Now that we have the processed data, we will describe the reasoning module. First, we will depict the learning process, in Section 4.5.1. Then, in Section 4.5.2, we will illustrate the implementation of a reasoning module based on the Bayesian network learned for reasoning over the simulated network.

4.5.1 Supervised Learning of the Bayesian Network

We have used GeNIe to apply the Bayesian Search algorithm to the data obtained from the simulated network and used a learning dataset. This tool is further described in the Appendix “Software Tools and Libraries”. We have also provided some background knowledge. Specifically, we have defined the influence of every node in the class node (the “err_type” node, which holds the result of the diagnosis). This technique is known as “Naive Bayes”.

Once we have the Bayesian network that will model causal relationships, it is validated using a different dataset obtained from a different simulation and using 10-fold cross-

validation as the test dataset.

This is done by exporting a .net file from GeNIe that holds information on the Bayesian network created.

4.5.2 Reasoning with Bayesian Model

GeNIe allows us to obtain the Bayesian network that models correlations found in training data. However, in order to perform reasoning over the Bayesian network with new evidences, we need to use a tool that allows us that inference in a Python script.

In order to do this, we have selected PGMPy [31] as the engine that will perform Bayesian inference. PGMPy allows for the use of a variety of algorithms for exact Bayesian inference, such as Variable Elimination, Belief Propagation or MPLP. We will be using Variable Elimination. We will also make use of the pyAgrum [32], which is a library for Graphical Universal Modelling. It allows for the integration of GeNIe and PGMPy, since they respectively export and accept different formats of Bayesian networks. Using these tools, we have developed a set of python scripts that perform the reasoning tasks.

Specifically, we have created a reasoning module that, given a .net file describing a Bayesian network, creates a "Reasoning" object which takes on the task of predicting the values of the model. Thus, when we feed evidences to this object, we are able to find the probability of any value in any variable. Then, the value with the highest probability is selected and presented as the result of the reasoning process.

CHAPTER 5

Evaluation

Once we have developed required modules of the proposed architecture, we run network simulations to collect data and perform a supervised learning process. After that, we validate the Bayesian network models generated. Results obtained from such validations show a good performance of the models tested, reaching values around 90% for F1-Score.

5.1 Introduction

In this chapter, we are going to evaluate the results obtained in the validation of our diagnosis system. First, in Section 5.2, we are going to show the Bayesian network models learned from collected data as training datasets. Then, we present the diagnosis results using evaluation datasets in Section 5.3.

5.2 Models

Using our network simulation module, we have obtained two datasets from two different simulations; the first dataset will be used as a "training" dataset, in order to run the

Bayesian search algorithm over it and obtain a Bayesian network. In the process of applying the Bayesian search algorithm for finding the Bayesian networks, we have configured the following parameters:

- Max Parent Count: 8. We limit the number of parent nodes a node can have, in order to limit memory consumption and time employed in the reasoning process.
- Iterations: 20. As we know, Bayesian search performs a hill-climbing process starting out from a random network. In order to search the full space of possible Bayesian networks, we restart the hill-climbing process once we have made a number of changes to the current network. The space searched increases with the number of iterations;
- Link probability: 0.1. It influences the connectivity of the starting random network.

Regarding the scoring function of the Bayesian search algorithm, we have selected the accuracy over the training data as the scoring function, and we have applied 10-fold cross-validation in the finding of the accuracy. Once we have used the training datasets to generate Bayesian networks, we use the "test" datasets, to evaluate the overall quality of our Bayesian models.

We have followed different processing methods and provided different levels of background knowledge, therefore developing different models. These models are presented in Sections 5.2.1, 5.2.2, 5.2.3 and 5.2.4.

5.2.1 Model 1: Attribute-Level Model

In the case of the first model, we have followed a more thoroughly processing method for the data. Specifically, in the time-based processing step, instead of just looking for any change in a flow rule, we have analysed multiple attributes that compose the flow rule, and their evolution through and specified time window. Therefore, we are able to detected unexpected changes in specific attributes of each flow rule.

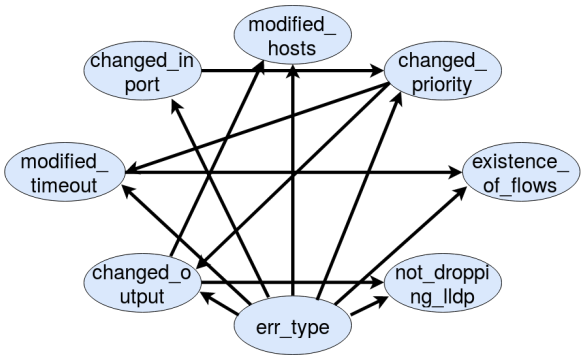

Figure 5.1: DAG learned for the first model

The structure of the Bayesian network obtained from such processing can be seen at Fig. 5.1.

5.2.2 Model 2: Fast-Learning Model

For the learning of this model we have followed the same process and principles as the ones followed in Section 5.2.1. However, in this case, we have also avoided providing any background knowledge. We have also modified some parameters of the learning algorithm in order for it to be faster. Specifically, we have reduced the number of iterations to 10 and the maximum number of parent nodes to 4, in order to further limit the duration of the learning process. The Bayesian network obtained from such processing and learning can be seen at Fig. 5.2.

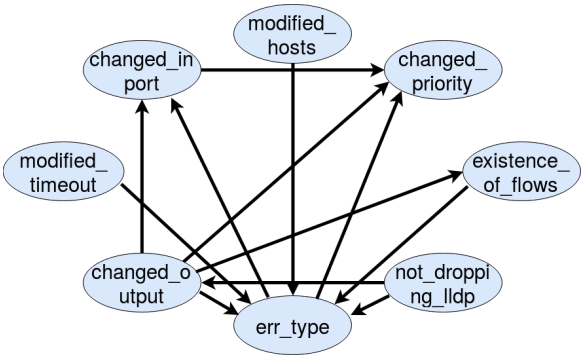

Figure 5.2: DAG learned for the second model

5.2.3 Model 3: Flow-Level Model

Regarding the third model, we have skipped the processing step mentioned in Section 5.2.1: in this case, we do not analyse which attributes have changed within each flow rule.

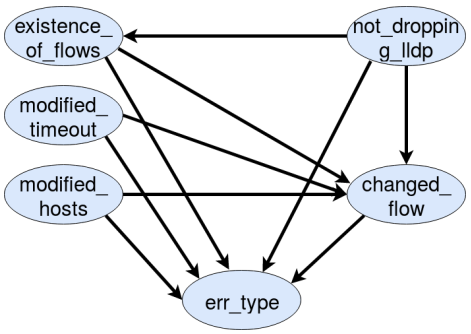

Figure 5.3: DAG learned for the third model

Instead, we analyse the flow rule as a whole, therefore being only able of detect if the flow rule has changed, but not which attribute within it has changed. As a result, instead of variables such as "changed_inport" or "modified_timeout" we use "changed_flow". We have used the same parameters for learning the Bayesian network as the ones mentioned in Section 5.1. The Bayesian network obtained from such data can be seen at Fig. 5.3.

5.2.4 Model 4: Pure Naive Bayes Model

In the case of the fourth model, we have taken a "pure Naive Bayes" approach: we only assume conditional relationships between the variable to be predicted and the rest of the variables in the model. This can be seen at Fig. 5.4

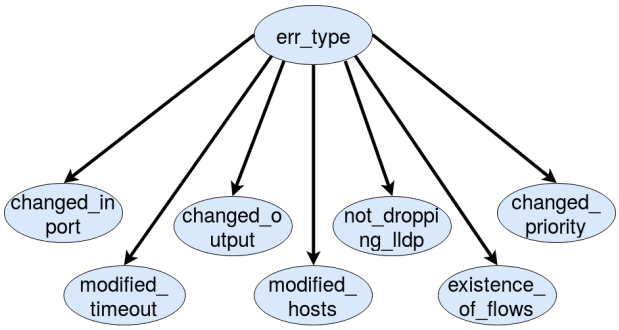

Figure 5.4: DAG learned for the fourth model

5.3 Results

The states (network faults) presented in this section are defined in Section 4.3.3.

Regarding the metrics used to evaluate or model, we apply: *Accuracy, Recall, Precision and F1-Score.*

- **Accuracy** represents the fraction of predictions that our model got right in a specific class; however, this metric does not consider the predictions that it got wrong.
- **Recall** tells us the proportion of entries of a certain status that were identified correctly.
- **Precision** shows us the proportion of entries classified as a certain status that were correct.
- **F1-Score** is the harmonic average of the Recall and the Precision.

Since the F1-Score takes into account both Precision and Recall, and it does not have the disadvantages that the Accuracy has, we will use the F1 Score as the reference in order

to evaluate our results.

Finally, results regarding all models can be seen at Sections 5.3.1, 5.3.2, 5.3.3 and 5.3.4.

5.3.1 Model 1: Attribute-Level Model

As we can see in Table 5.1, the first model performs well in the diagnosis of most of the faults. However, we can see that it has difficulties when diagnosing a fault-free status of the network, since we obtain a value of F1-Score for the S0 status of 0.69, lower than the 0.95-1.00 range that we obtain for most of the remaining statuses.

Fault Type	**S0**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**
F1-Score	0.69	0.75	1.00	0.96	0.95	1.00	0.95	0.94	0.93	0.87
Recall	0.59	1.00	1.00	1.00	1.00	1.00	0.90	1.00	1.00	0.87
Precision	0.83	0.6	1.00	0.92	0.90	1.00	1.00	0.89	0.88	0.87
Accuracy	0.90	0.97	1.00	0.99	0.98	1.00	0.99	0.99	0.98	0.98

Table 5.1: Metrics for Model 1

5.3.2 Model 2: Fast-Learning Model

As we can see in Table 5.2, the results obtained are similar to the ones presented in Section 5.3.1, but with a faster learning method. Also, we have not provided any background knowledge.

Fault Type	**S0**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**
F1-Score	0.72	0.89	1.00	0.96	0.95	0.97	0.90	0.94	0.93	0.86
Recall	0.68	0.89	1.00	1.00	1.00	0.94	0.82	1.00	1.00	0.86
Precision	0.77	0.89	1.00	0.92	0.90	1.00	1.00	1.00	0.88	0.86
Accuracy	0.90	0.99	1.00	0.99	0.98	0.99	0.98	0.99	0.98	0.98

Table 5.2: Metrics for Model 2

5.3.3 Model 3: Flow-Level Model

In Table 5.3, we can see the evaluation of the model obtained from the model described in Section 5.2.3. As we can see, due to the fact that we are skipping some steps in the processing of the dataset, we are losing the ability to detect some faults in the network.

Specifically, we can see that S3, S4, S5 and S9 faults are probably being diagnosed as S3 (due to the fact that S3, S4, S5 and S9 have a recall of 0.00, and S3 has a very low precision).

Fault Type	**S0**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**
F1-Score	0.77	0.89	1.00	0.31	0.00	0.97	0.90	0.00	0.86	0.00
Recall	0.69	0.89	1.00	1.00	0.00	0.94	0.83	0.00	1.00	0.00
Precision	0.87	0.89	1.00	0.18	0.00	1.00	1.00	0.00	0.76	0.00
Accuracy	0.90	0.99	1.00	0.72	0.86	0.99	0.98	0.96	0.95	0.92

Table 5.3: Metrics for Model 3

5.3.4 Model 4: Pure Naive Bayes Model

As we can see in Table 5.4, the results obtained for this model are similar to the obtained for the second model, whose results can be seen in Table 5.2. Specifically, the results obtained are the same except for the statuses S0 and S5, where this model is outperformed.

Fault Type	**S0**	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**
F1-Score	0.69	0.89	1.00	0.96	0.95	0.87	0.90	0.94	0.93	0.86
Recall	0.59	0.89	1.00	1.00	1.00	1.00	0.82	1.00	1.00	0.86
Precision	0.83	0.89	1.00	0.92	0.90	0.77	1.00	1.00	0.88	0.86
Accuracy	0.90	0.99	1.00	0.99	0.98	0.98	0.98	0.99	0.98	0.98

Table 5.4: Metrics for Model 4

5.3.5 Summary

After we analysed the results by class of every model, we obtain the average considering all the classes for every model. Such results can be seen at Table 5.5. As can be seen, the accuracy of all models are very similar. However, M1, M2 and M4 significantly outperforms M3. Therefore, if we had used the accuracy as the value to measure the performance, we would have wrongly reached the conclusion that they all have a similar performance.

But since we use the F1-Score as the metric to determine the performance, we can see that M1 significantly outperforms M3, and M2 slightly outperforms both M1 and M4.

Model	**M1**	**M2**	**M3**	**M4**
F1-Score	0.904	0.912	0.570	0.899
Recall	0.936	0.919	0.635	0.916
Precision	0.889	0.922	0.573	0.905
Accuracy	0.978	0.978	0.929	0.977

Table 5.5: Comparison of all models

5.4 Discussion

As mentioned previously, M1, M2 and M4 have a slight tendency to find errors even when the network is functioning correctly, as noted by the F1-Score of the S0 status (S0 status: no errors); however, this trend is not worrisome, since the F1-Score is still high. In fact, we have some statuses, such as status S2, which we are able to diagnose completely. On the other hand, we can see in M3 a trade-off between processing time and ability to diagnose some statuses (specifically, ability to diagnose statuses S4, S7 and S9, and to distinguish S3 from other statuses).

Finally, we can see in the comparison between M4 and the rest of the models that we are able to obtain a simple, yet well-performing model using pure Naive Bayes as the learning method.

CHAPTER 6

Conclusions and Future Work

In this project, we have proposed and prototyped a solution for the automation of faults management and diagnosis in SDN, by making use of Machine Learning techniques, Big Data tools and Bayesian reasoning techniques. In this chapter, we present the conclusions obtained from such work and propose possible research lines for the future.

6.1 Conclusions

New technologies such as SDN provide a number of benefits in the virtualisation and management of network services. Nevertheless, some research is needed on the application of techniques for enabling its autonomic management. For this purpose, Big Data technologies provide the foundation for collecting and processing huge amounts of raw data from the telecommunication network.

By finding causal relationships within the data, we are able to perform learning and reasoning for fault diagnosis. However, in order to do this, first we have to select the adequate data sources according to the possibility of holding causal relationships by the data collected from them. We also need to perform the appropriate analytic tasks in order

to mitigate any statistical noise that could be distorting the learning of causal relationships.

Then, by finding a way of implementing the causal relationships learned from the data, we can create a system that autonomously diagnose the status of a network (or a network element). This can be done applying machine learning techniques to generate Bayesian networks and reason with them.

In this project, we have proposed a Big Data based architecture that takes advantage of Big Data technologies and explores the use of analysis techniques in order to perform reasoning based on Bayesian networks. Main effort has been dedicated to the creation of the testing environment, since there are not yet available benchmarks for diagnosis purposes. We have implemented a prototype which has been deployed using virtualisation techniques. Finally, multiple models have been generated following different criteria regarding the processing of the data, background knowledge provided and configuration of the Bayesian search algorithm. Their evaluation showed that some of these models reach a F1-Score higher than 90%.

6.2 Future Work

As we have seen, the proposed architecture shows potential for fault diagnosis in SDN. However, this work is still in progress. Next steps in this research line include combining diagnosis models to cope more complex cases, in order to broad the coverage of possible faults within the monitored network.

This can be done by further processing additional data sources, such as final user applications, including probes in the network and/or servers or implementing testing agents which could execute specific tests when symptoms or anomalies are detected in the network.

We would also like to implement a more realistic network simulation, by using agents that mimic real behaviours of users in a telecommunications network. Then, we could consider the behaviour of such agents in the learning and reasoning process. We would also want to introduce new services in the simulation, such as point-to-multipoint streaming.

Moreover, the designing and implementation of a self-healing service is another important aspect that will be addressed as future work. Our current system diagnoses faults, but it does not manage them. In order to do this, we would like to implement a self-healing module which could use machine learning techniques in order to design new traffic policies according to the diagnosis. We could also store back-ups of traffic policies, and push them into the network each time a fault is diagnosed.

Finally, another possible path for future work would be to create a module which takes on the task of coordinating the rest of the modules of our architecture. This "orchestrator" defines a semantic model of the diagnosing process in order to coordinate the steps of fault diagnosing. Then, according to this semantic model, it automates and coordinates the sequence of diagnosing the status of the network using the data stored in the Data Lake.

APPENDIX A

Impact of this Project

Computer networks are present in almost every aspect of our everyday life. Now more than ever our work, our social relationships and our entertainment depend on computer networks. In fact, it can be argued that a big part of our life is based on our "connection" with the rest of the world through computer networks. Even further, computer networks have a vital role in the development and management of companies and infrastructures. Therefore, a system that could have a noticeable impact in computer networks could also have an impact both in the managing of business and in our everyday life. In this appendix, we explain the social, economic, environmental and ethical implications of such system.

A.1 Introduction

In this appendix, we are going to consider the possible social, economic and environmental impact that this project could have in Sections A.2, A.3 and A.4, respectively. We will also reflect on the possible ethical and professional implications of such project in Section A.5.

A.2 Social Impact

Faulty computer networks could have a serious impact in our everyday life. If we are relaxing while making use of a computer network (for example, consuming any type of content through the Internet), a faulty computer network could affect our stress levels. Even more, if we are working and faults in a computer network do not allow us to continue working. Therefore, we could expect a social impact of our system in the avoidance of stress due to faulty computer networks.

We could also predict an impact on the privacy of the end users in the network which is being monitored. In the prototype implemented, we used the network controller as the data source. However, as we mentioned in Section3.3.1, we could use many other data sources. If we collect data from the end users' devices, we would have to comply with current laws on personal data protection.

Security-wise, our system is vulnerable to some attacks. Since it depends on the data sources used for diagnosing, an attack that disables one or more sources could hamper the diagnosis of the network. Also, by attacking the data sources during the learning process, we could force the system to learn a faulty Bayesian network, therefore damaging the diagnosis process. However, these risks are mitigated by the fact that Bayesian networks are very resilient to the absence of data.

A.3 Economic Impact

One of the fields where the impact is most notable is in the economic field. Legacy networking is very costly: since there are many vendor-specific protocols and solutions, it is very complex (and, therefore, expensive) to develop and deploy autonomous managing solutions that work in any network environment. Therefore, a common approach is to hire a group of networks engineers in order to maintain the network. This is obviously very costly.

Thanks to SDN, and specifically thanks to the standardisation of OpenFlow as the communications protocol between nodes and controller, we can now develop applications that will work in any computer network. By developing applications that takes on the task of autonomously assuring the network resilience in any computer network, we can dramatically cut costs on network management.

A.4 Environmental Impact

We have mentioned situations where faults in computer networks could cause stress to users. However, this seems trivial when comparing with faults in computer networks in environments managing key infrastructures, such as train lines, airports, or power stations. An unexpected, uncontrolled fault in a computer network in such environments could potentially cause a catastrophe.

Therefore, a faulty computer network could provoke environmental damage. For example, leaks of waste to the environment could be provoked due to a miscommunication between systems. Since the goal of this project is to achieve a fault-free computer network, this project helps in reducing the risk of any eventuality provoked by a faulty computer network.

A.5 Ethical and Professional Implications

The ethical implications of this project are basically the ethical implications of using machine learning techniques. Obviously, the first ethical issue that arises is the fact that if we design a machine-learning-based system that takes on the issue of fault management in computer networks, we are putting any person that works on the same field out of a job. However, it is also true that while the use of machine learning techniques is automating jobs, it is also creating new jobs in the designing of such systems.

The second ethical implication of our system has to do with the data collecting. There is no ethical controversy in collecting data from the network controller about the status of each node. It is anonymous, highly technical data. However, there are ethical implications in the collecting of user data from apps, as proposed in Section 3.3.1. However, since this data is anonymous and technical (i.e. traffic rates, types of packets received...), we do not think that there is an ethical controversy.

Finally, there is also a professional implication in using machine learning algorithms. Depending on the data used in the learning of the Bayesian model (for example, if it is balanced or not), we could end up creating a Bayesian network (and thus, a diagnosis service) that tends to predict one scenario most of the time just because it was the most common in the training dataset. This is an example of automating human tasks wrongly. However, this risk can be mitigated by employing talented data scientists.

APPENDIX B

Cost of the System

In the designing and development of this project, we have incurred in some costs, most of them derived from the salaries of the developers, although some costs have been provoked by hardware needs. If we wanted to commercialise our software, we would need to spend money on licences, and consider taxes on software products in Spain.

B.1 Introduction

In this appendix, we are going to evaluate the possible expenses that this system could cause. First, we are going to describe the costs in physical resources needed for our system to run in Section B.2. Then, we are going to estimate the costs regarding personnel needed to design and maintain this system in Section B.3. Next, in Section B.4 we are going to specify the budget needed for software licences. Finally, we consider the possible taxation involved in the selling of this software in Section B.5.

B.2 Physical Resources

In order to run our system, obviously we need a computing machine powerful enough to run all the modules. In our experience while developing the system, we have come to realise that the recommended requirements for a computer that would run it without any kind of issue are approximately the following:

- **Hard Disk**: 30 GB
- **RAM**: 8 GB
- **CPU**: Intel i7 processor, $\sim$ 2.80 GHz

On average, a machine with such capabilities costs around 1,000 € as of 2018. However, if we intend to deploy our system, we would need to invest in a cluster of computers with similar capabilities. Therefore, the costs of such infrastructure could reach 6,000 €.

B.3 Human Resources

In this section, we will take into account the hours employed by us in the designing and developing of this system. We will make a prediction on the average salary of a Software Engineer, in order to find the cost of developing the project.

We estimate the cost in hours in developing this project to be around 368 hours. We have come up with this number by considering 4 months of work (considering also 23 working days in a month) in a part-time schedule (4 hours per working day). We estimate the salary of an engineer developing the system to be around 1500 € per month (gross). Therefore, we predict the cost of developing the prototype to be around 368 hours and 6,000 €.

Once this system is created and deployed, we also expect maintenance costs. We would also have to adapt our system to a real-life network, which is more complex than the network simulation used in our prototype. In order to maintain and develop the system, we anticipate that we will need a Software Engineer working full-time, so he can address any possible issue that may occurs and keep the system modules updated. If we consider the salary of a Software Engineer working at full-time to be around 1,500 € per month (gross), we expect the cost of maintaining the system to be around 18,000 € per month.

B.4 Licences

Most of the software used in the developing of this project is open-source software, so we do not expect high expenses in licences. However, the engine and GUI used for the Bayesian network learning and validation, GeNIe (and its engine, SMILE), have been employed using an academic license free of charge. If we were to use this system professionally, we would have to buy a license to the company that owns the rights of such engine.

The price is not publicly exposed by BayesFusion LLC., the company that owns the licensing rights to GeNIe. However, by comparing to other Bayesian learning systems currently available in the market, we expect the cost of such software to be around 10,000 € per year.

B.5 Taxes

One of the possible actions we could take once we have implemented our system is selling the entire software to another company. In that case, we would have to consider the taxes involved in the selling of software products in Spain.

According to [33], there is a tax of 15% over the final price of the product, as regulated by the Statue 4/2008 of the Spanish law. Furthermore, if we want to sell the product to a foreign company, we would need to consider possible cases of double taxation. However, we consider that this is beyond the scope of this project.

APPENDIX C

Software Tools and Libraries

In order to develop the project described in this document, a wide variety of tools have been used. We have used tools for network and traffic generation and network simulation, such as Mininet or Socat. We have also used tools such as PGMPy or Genie/SMILE, which allowed us to learn and implement the reasoning techniques used in this project. Finally, we made use of libraries that eased the processing of the data, like Pandas.

C.1 Introduction

In this appendix, we are going to enumerate and describe the libraries and tools used in this project. First, in Section C.2, we are going to describe the library used for the implementation of simulated SDN network. Then, the tools used for traffic simulation is explained in Section C.3 and Section C.4. Next, in Section C.5, we describe the software used in the process of learning and validation of Bayesian networks. Section C.6 depicts the network controller used in the prototype. Then, we are going to explain the library used for the designing of the scale-free network topology in Section C.7. Next, we detail the library used for data processing in Section C.8. Finally, we are going to illustrate the libraries used

in the reasoning process in Section C.9 and Section C.10.

C.2 Mininet

Mininet [34] is an open-source software used to create a virtual SDN. Mininet takes on the task of creating all the necessary elements of a network (nodes, hosts and links), and allowing communications between them and the SDN controller, without worrying on how this environment is deployed. Thanks to Mininet, we are able to emulate an entire network on a single computer. In order to create the network, we have to provide the following information to Mininet:

1. The SDN controller: when no controller is specified, Mininet provides POX [35], a Python-based SDN controller that uses OpenFlow to coordinate traffic policies with nodes in the network. However, a remote controller can be also used, by specifying an IP and a port where the remote controller listens for (and sends) OpenFlow messages.
2. The topology: if no topology is provided, we can create a network only through the Command Line Interface (CLI): in that case, a simple default topology with two hosts connected by one node is created. However, to create more complex topologies, the topology must be provided either through options in a command (using the CLI) or specifying it while creating the simulation using the Python API.

Once this information is given, Mininet simulates the network, composed by a number of switches (OpenvSwitch switches), hosts, links and a network controller. When hosts are created, Mininet creates a shell process (for example, *bash*). Then, it moves the process into its own network namespace, with the *unshare(CLONE_NEWNET)* system call. The network namespace provides processes with exclusive ownership of interfaces, ports and routing tables.

However, the network namespace only provides isolation regarding network resources. It does not provide an isolated file system (such as, for example, the one that mount namespaces provides), or an isolated process environment (such as the one provided by process namespaces). Therefore, all hosts within the network simulation share the same file directory and are able to see other hosts' processes. Nevertheless, it is enough for the purposes of our simulation.

Then, links are created. Just as hosts are network namespaces, links in Mininet are just pairs of virtual Ethernet devices. A virtual Ethernet device, or vEth (as we will call it from now on), is particularly useful when connecting different namespaces, since vEth can

only be created in pairs. Therefore, we use them to create links between hosts (which are defined each one in its own network namespace) and switches (which are virtualised in the root namespace).

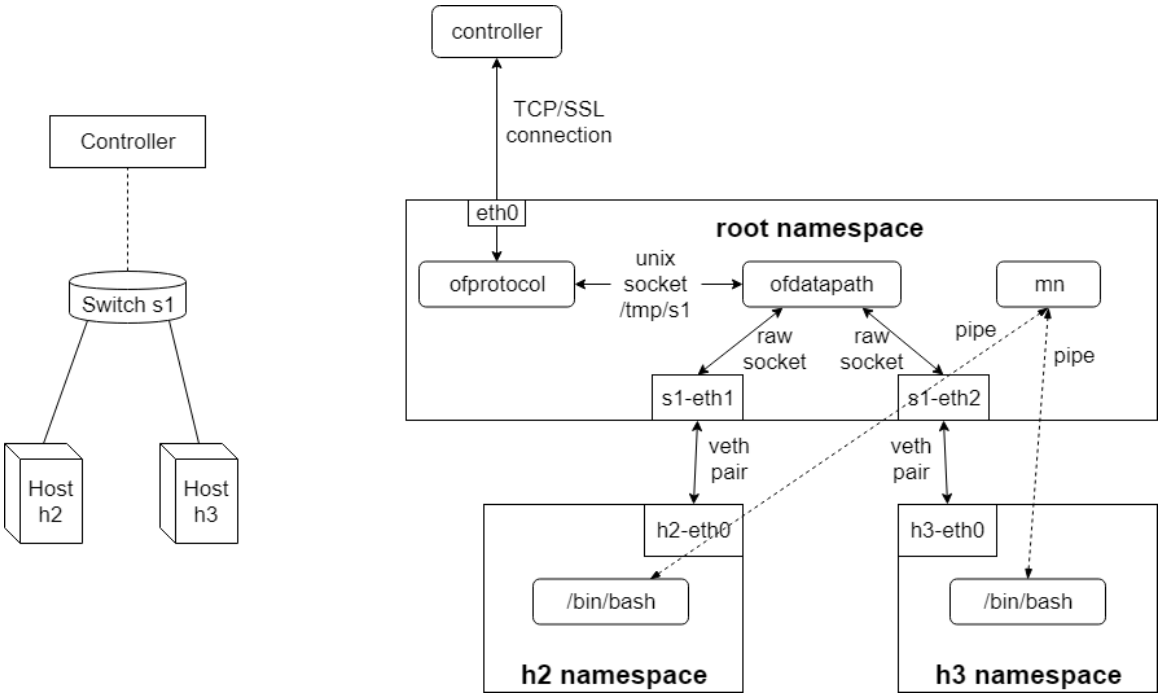

Figure C.1: Low-level implementation (right) of a simple network (left).

However, vEths can also connect virtual switches created within the root namespace. Therefore, vEths are also used in switch-to-switch links. Finally, communications between the SDN controller and network switches do not run over links defined by pairs of vEths. Instead, in the case of remote controllers (as in our case) OpenFlow messages are directed through the eth0 interface of the root namespace.

An example of how elements are interconnected through virtual and physical interfaces can be seen at Fig. C.1. As we can see, connections between network devices are set with vEth pairs. However, connections between the switch and the controller are directed through the eth0 interface. Sockets provide connectivity between internal elements of a switch.

C.3 Socat

We use Socat [36] for the creation of network traffic. Socat is an utility that allows us to establish byte streams between two locations. The power of this utility resides in its

flexibility: a myriad of data sources, destinations and communication protocols can be used to implement such streams. Therefore, when using other hosts as destinations, a wide range of network traffic can be created implementing socket communications between hosts.

Socat has a life cycle composed by four phases. First, the command is parsed and the logging is initialised. Then, Socat opens the source and destination addresses. Then, during the transfer phase, the data is transferred when available. Lastly, when Socat reaches a closing condition, it closes the stream and the addresses. In this project, Socat is used to implement the traffic mentioned in Section 2.5.1.

C.4 VLC

We use VideoLAN Client (VLC) as the tool used for developing the streaming service mentioned in Section 4.3.2. VLC is an open-source tool used as a media player and streaming server which allows for a flexible management of video and audio streams, since it implements a wide variety of transcoding and streaming methods, and supports a broad scope of data sources. Thanks to this feature, we do not have to worry about fully developing an implementation of any streaming protocol. We only need a file containing a video to be streamed, a streaming method, and a path between the source and destinations of the streaming service.

C.5 GeNIe-SMILE

The GeNIe Modeler [37] is a reasoning and learning/causal discovery system used to learn probabilistic graphical models (such as Bayesian networks) from data or to create custom ones. This system also provides a variety of exact and approximate reasoning algorithms for inference. The software is closed-source; however, free versions are provided for academic use only.

This software is composed of two pieces. Firstly, the Structural Modeling, Inference and Learning Engine (SMILE), the engine that holds the learning and inference logic. It is written in C++; however, multiple wrappers are offered in order to allow the use in several development languages. It supports most probabilistic graphical models and learning methods (structural, parametric and incremental learning, model refinement, causal discovery). We are particularly interested in its ability to learn Bayesian networks from data using structural learning algorithms. And secondly, a GUI that allows us to intuitively create or learn our own networks on training data and test their quality through inference on

test data. This GUI is only available for Windows operating systems; however, it can be executed in any OS using a Windows emulating layer (such as Wine [38] in Ubuntu).

C.6 Opendaylight

Opendaylight [39] is a SDN controller supported by companies such as Cisco, Ericsson, Huawei or ZTE. Opendaylight is designed around a Model-Driven Service Abstraction Layer (MD-SAL) that describes network devices and applications as objects. These models are defined using YANG modelling language. Since a generalised description of a device or application is given, the specific implementation does not need to be known. Interactions with these objects (or models) are processed within the MD-SAL. Opendaylight communicates with the network through "southbound" interfaces anchored to the MD-SAL. These interfaces implement the protocol used for communication with devices in the network. Opendaylight is provided inside a Karaf [40] container.

Inside the controller platform, multiple service functions are defined and contained inside their own modules. Access to many of those modules is provided by the Opendaylight controller through its "northbound" REST API. This API allows us not only to obtain data, but also to introduce changes in configuration parameters stored by those modules. Therefore, the REST API allows us to interact with the network. When communicating with modules tasked with managing traffic routing, this API can be used to access and change traffic policies.

Furthermore, the Opendaylight controller provides a GUI accessible via HTTP connection to a port of the machine where Opendayligt is running. In such GUI we can see a graphical representation of the network topology. We can also access much information, such as a guide to the REST API (where we can not only access information on how to communicate with the controller modules, but also implement our own queries intuitively) or the YANG models that define the data provided through the API.

C.7 NetworkX

The Python package NetworkX [27] is a library that allows us to create networks in Python. Even though this can be done without the need of any library, NetworkX is designed to ease the creation and managing of complex networks. One of its most interesting features is the fact that it allows the implementing of a wide variety of network types for both undirected and directed graphs. In fact, it also allows the development of graphs with self-loops and

parallel edges.

In the case of undirected graphs (which is the type of graphs we need as basis for the computer network topology), we can create a wide variety of networks. For the purpose of network simulation, we are specifically interested in scale-free networks, as justified in Section 4.3.1.

C.8 Pandas

The Python package Pandas [41] is a library designed for data managing and analysis in Python. It defines three data structures: Series and DataFrames. A series is a one-dimensional labelled array which can hold any type of data. We can create a Series from a dictionary, a ndarray or a scalar value; Pandas defines multiple attributes and functions to work with such data structures.

On the other hand, a DataFrame is not one-dimensional, but two-dimensional. While a Series resembles an array, a DataFrame is designed to mimic a spreadsheet. It is the most commonly used Pandas data structure, and it is the one that we used in the processing of the data. Pandas provide methods for iterating over and accessing the data stored in the DataFrame efficiently. It also provides methods for performing operations in (and between) DataFrames without accessing the data, and perform statistical analysis in numeric attributes.

Finally, Pandas also provides methods for reading and writing CSV. This is particularly interesting for us, since we download data from the Data Lake as CSV files, and we send the processed data in CSV files to the reasoning module.

C.9 pyAgrum

The Python package pyAgrum [32] is a library designed for building graphical models. Specifically, pyAgrum is a wrapper in Python for the library aGrUM, written in C++. The aGrUM library has many functionalities: it can be used for graphical model's learning, implementation and inference. However, since we use GeNIe for the learning and validating of Bayesian networks, and PGMPy for the implementation and inference, we are interested in pyAgrum because of its ability to read .net files provided by GeNIe.

Thanks to pyAgrum, we can convert .net files in Bayesian Interchange Format (BIF) files, which can be read by PGMPy.

C.10 PGMPy

The Python package PGMPy [31] allows for the implementation of Probabilistic Graphical Model (PGM). A PGM allows us to represent conditional dependencies between random variables. This is particularly useful in our case, since a Bayesian network is just a PGM; therefore, we will use PGMPy as the library to implement Bayesian networks in Python.

Specifically, PGMPy allows us to implement two different types of PGM: Bayesian networks and Markov networks. Bayesian networks were explained at Section 2.4.1. Markov networks are similar to Bayesian networks, but they are represented by an undirected graph. Since the graph is undirected, the concept of "conditional probability" does not make sense. Therefore, in Markov networks we use the concept of "affinity" to represent the likelihood of two random variables having certain values.

In the case of Bayesian networks, it allows for the creation of a PGM Python object from a BIF file. This is useful, since we learn the Bayesian network from GeNIe. Once we learn such network we just have to export it in .net format, and use pyAgrum (described in Section C.9) to convert it to BIF format. Then, we can use PGMPy to implement it.

Once we have implemented such network, we can use multiple methods for Bayesian inference. Specifically, PGMPy offers the following methods:

1. Variable Elimination
2. Belief Propagation
3. MPLP
4. Elimination Ordering

Bibliography

[1] Cisco. The zettabyte era: Trends and analysis. *Cisco White Papers*, 2016.

[2] Nokia. 7950 xrs-xc: Paving the path to petabit routing. `https://networks.nokia.com/solutions/7950-xrs-xc-core-router`, 2017. Accessed: 2018-07-03.

[3] Orange Business Services. Enterprises want sdn to build flexible networks. `https://www.orange-business.com/en/blogs/connecting-technology/networks/enterprises-want-sdn-to-build-flexible-networks`, 2015. Accessed: 2018-09-03.

[4] Cisco. A successful digital business needs an agile network. `https://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/unified-fabric/agile-network.pdf`. Accessed: 2018-09-03.

[5] Deloitte. Disrupting telco business through sdn / nfv. `https://www2.deloitte.com/es/es/pages/technology-media-and-telecommunications/articles/disrupting-telco-business-through-SDN-NFV.html`. Accessed: 2018-09-03.

[6] James Kempf, Elisa Bellagamba, András Kern, David Jocha, Attila Takács, and Pontus Sköldström. Scalable fault management for openflow. In *Communications (ICC), 2012 IEEE international conference on*, pages 6606–6610. IEEE, 2012.

[7] Hyojoon Kim, Mike Schlansker, Jose Renato Santos, Jean Tourrilhes, Yoshio Turner, and Nick Feamster. Coronet: Fault tolerance for software defined networks. In *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pages 1–2. IEEE, 2012.

[8] Carmelo Cascone, Luca Pollini, Davide Sanvito, Antonio Capone, and Brunilde Sanso. Spider: Fault resilient sdn pipeline with recovery delay guarantees. In *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*, pages 296–302. IEEE, 2016.

[9] Antonio Capone, Carmelo Cascone, Alessandro QT Nguyen, and Brunilde Sanso. Detour planning for fast and reliable failure recovery in sdn with openstate. In *Design of Reliable Communication Networks (DRCN), 2015 11th International Conference on the*, pages 25–32. IEEE, 2015.

[10] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester. Fast failure recovery for in-band openflow networks. In *Design of reliable communication networks (drcn), 2013 9th international conference on the*, pages 52–59. IEEE, 2013.

[11] Gabriela Gheorghe, Tigran Avanesov, Maria-Rita Palattella, Thomas Engel, and Ciprian Popoviciu. Sdn-radar: Network troubleshooting combining user experience and sdn capabilities. In *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, pages 1–5. IEEE, 2015.

[12] Yongning Tang, Guang Cheng, Zhiwei Xu, Feng Chen, Khalid Elmansor, and Yangxuan Wu. Automatic belief network modeling via policy inference for sdn fault localization. *Journal of Internet Services and Applications*, 7(1):1, 2016.

[13] Neda Beheshti and Ying Zhang. Fast failover for control traffic in software-defined networks. In *Global Communications Conference (GLOBECOM), 2012 IEEE*, pages 2665–2670. IEEE, 2012.

[14] Balakrishnan Chandrasekaran and Theophilus Benson. Tolerating sdn application failures with legosdn. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 22. ACM, 2014.

[15] Fábio Botelho, Alysson Bessani, Fernando MV Ramos, and Paulo Ferreira. On the design of practical fault-tolerant sdn controllers. In *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, pages 73–78. IEEE, 2014.

[16] He Li, Peng Li, Song Guo, and Amiya Nayak. Byzantine-resilient secure software-defined networks with multiple controllers in cloud. *IEEE Transactions on Cloud Computing*, 2(4):436–447, 2014.

[17] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 4. ACM, 2015.

[18] Nader Bouacida, Amer Alghadhban, Shiyam Alalmaei, Haneen Mohammed, and Basem Shihada. Failure mitigation in software defined networking employing load type prediction. In *Communications (ICC), 2017 IEEE International Conference on*, pages 1–7. IEEE, 2017.

[19] José Manuel Sánchez, Imen Grida Ben Yahia, and Noël Crespi. Thesard: On the road to resilience in software-defined networking through self-diagnosis. In *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*, pages 351–352. IEEE, 2016.

[20] Xuemei Ding, Yi Cao, Jia Zhai, Liam Maguire, Yuhua Li, Hongqin Yang, Yuhua Wang, Jinshu Zeng, and Shuo Liu. Bayesian network modelling on data from fine needle aspiration cytology examination for breast cancer diagnosis. In *Proceedings of the 2017 5th International Conference on Frontiers of Manufacturing Science and Measuring Technology (FMSMT 2017)*. Atlantis Press, 2017.

[21] Shamshad Lakho, Akhtar Hussain Jalbani, Muhammad Saleem Vighio, Imran Ali Memon, Saima Siraj Soomro, et al. Decision support system for hepatitis disease diagnosis using bayesian network. *Sukkur IBA Journal of Computing and Mathematical Sciences*, 1(2):11–19, 2017.

[22] Sun Jin, Changhui Liu, Xinmin Lai, Fei Li, and Bo He. Bayesian network approach for ceramic shell deformation fault diagnosis in the investment casting process. *The International Journal of Advanced Manufacturing Technology*, 88(1-4):663–674, 2017.

[23] Salma Ktari, Stefano Secci, and Damien Lavaux. Bayesian diagnosis and reliability analysis of private mobile radio networks. In *Computers and Communications (ISCC), 2017 IEEE Symposium on*, pages 1245–1250. IEEE, 2017.

[24] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *NSDI*, pages 117–130, 2015.

[25] VR Benjamins et al. Problem-solving methods for diagnosis and their role in knowledge acquisition. *International Journal of Expert Systems: Research and Applications*, 8, 1996.

[26] Albert-László Barabási, Réka Albert, and Hawoong Jeong. Mean-field theory for scale-free random networks. *Physica A: Statistical Mechanics and its Applications*, 272(1-2):173–187, 1999.

[27] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[28] Filebeat. `https://www.elastic.co/products/beats/filebeat`. Accesed: 2017-11-28.

[29] Logstash. `https://www.elastic.co/products/logstash`. Accessed: 2018-05-09.

[30] Elasticsearch. `https://www.elastic.co/products/elasticsearch`. Accessed: 2017-11-09.

[31] Pgmpy. `http://pgmpy.org/`. Accessed: 2018-23-04.

[32] Christophe Gonzales, Lionel Torti, and Pierre-Henri Wuillemin. aGrUM: a Graphical Universal Model framework. In *International Conference on Industrial Engineering, Other Applications of Applied Intelligent Systems*, Proceedings of the 30th International Conference on Industrial Engineering, Other Applications of Applied Intelligent Systems, Arras, France, June 2017.

[33] Francisco de la Torre Díaz. La tributación del software en el IRNR. Algunos aspectos conflictivos. *Cuadernos de Formación*, 10/2010:239–249, 2010.

[34] Mininet. `http://mininet.org/`. Accessed: 2017-11-09.

[35] Pox controller. `https://github.com/noxrepo/pox`. Accessed: 2018-13-03.

[36] Socat. `http://www.dest-unreach.org/socat/doc/socat.html`. Accessed: 2017-11-09.

[37] BayesFusion. Genie modeler. `https://www.bayesfusion.com/`, 2017.

[38] Wine. `https://wiki.winehq.org/Main_Page`. Accessed: 2018-14-03.

[39] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a*, pages 1–6. IEEE, 2014.

[40] Karaf. `https://karaf.apache.org/`. Accessed: 2018-14-03.

[41] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[42] PwC. The software-defined carrier: How extending network virtualisation architecture into IT BSS/OSS architectures opens up transformational opportunities for telecom and cable operators. https://www.pwc.com/gx/en/industries/communications/publications/communications-review/assets/communications-review-the-software-defined-carrier.pdf, 2016. Accesed: 2017-11-27.

[43] Florian Groene, Christopher Isaac, Harish Nalinakshan, and Joseph Tagliaferro. The software-defined carrier: How extending network virtualisation architecture into IT BSS/OSS architectures opens up transformational opportunities for telecom and cable operators. *Communications Review*, December 2016.

[44] Dilpreet Singh and Chandan K Reddy. A survey on platforms for big data analytics. *Journal of Big Data*, 2(1):8, 2015.

[45] Hadoop. http://hadoop.apache.org/. Accessed: 2017-11-09.

[46] The OpenDaylight Project, Inc. Opendaylight. https://www.opendaylight.org/. Accessed: 2017-11-09.

[47] José Sánchez, Imen Grida Ben Yahia, and Noël Crespi. Poster: Self-healing mechanisms for software-defined networks. *arXiv preprint arXiv:1507.02952*, 2015.

[48] Cisco VNI. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021. *Cisco White Papers*, 2016.

[49] Vishal Sharma. *Beginning Elastic Stack*. Springer, 2016.

[50] Karamjeet Kaur, Japinder Singh, and Navtej Singh Ghumman. Mininet as software defined networking testing platform. In *International Conference on Communication, Computing & Systems (ICCCS)*, pages 139–42, 2014.

[51] Markets and Markets. Software-Defined Networking and Network Function Virtualization Market by Component (Solution (Software (Controller, and Application Software), Physical Appliances), and Service), End-User, and Region - Global forecast to 2022, 2017.

[52] Accenture. Network transformation survey 2015, final results. https://www.accenture.com/t20170416T224033Z__w__/us-en/_acnmedia/Accenture/Conversion-Assets/DotCom/Documents/Global/PDF/Indurties_17/Accenture-Network-Transformation-Survey-2015.pdf, 2015. Accessed: 2017-11-27.

[53] Accenture CMT Digital Consumer Survey 2015. Network transformation survey 2015, final results. Technical report, Accenture, 2015.

[54] Stewart Baines. Enterprises want sdn to build flexible networks. https://www.orange-business.com/en/blogs/connecting-technology/networks/enterprises-want-sdn-to-build-flexible-networks, 2015. Accessed: 2017-11-27.

[55] Stewart Baines. Enterprises want SDN to build flexible networks. Technical report, Orange, June 2015.

[56] Sean Vig. Network congestion as an emergent phenomena in internet traffic. 2011.

[57] Steve Alexander. When networks hit the wall. `https://www.networkworld.com/article/3221333/lan-wan/when-networks-hit-the-wall.html`, 2017. Accesed: 2017-11-28.

[58] Steve Alexander. When networks hit the wall. *NetworkWorld*, September 2017.

[59] Cisco. Cisco Global Cloud Index: Forecast and Methodology, 2015–2020. *Cisco White Papers*, 2015.

[60] Allan Vidal. Flexible networking hot trends at SIGCOMM 2016. *Ericsson Research Blog*, 2016.

[61] BayesFussion, LLC. Genie. `https://www.bayesfusion.com`. Accessed: 2017-12-01.

[62] László Gyarmati and Tuan Anh Trinh. Scafida: A scale-free network inspired data center architecture. *ACM SIGCOMM Computer Communication Review*, 40(5):4–12, 2010.

[63] Fernando Perez, Brian E Granger, and John D Hunter. Python: an ecosystem for scientific computing. *Computing in Science & Engineering*, 13(2):13–21, 2011.

[64] Bruno Lecoutre and Jacques Poitevineau. The significance test controversy revisited. In *The Significance Test Controversy Revisited*, pages 49–62. Springer, 2014.