UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN
INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MASTER

Development of a Fault Diagnosis System of Software Defined Networks based on Linked Data Technologies.

Manuel García-Amado Sancho

2018

TRABAJO DE FIN DE MÁSTER

Título:	Desarrollo de un Sistema de Diagnóstico de Redes Definidas por Software basado en Tecnologías de Datos Enlazados.
Título (inglés):	Development of a Fault Diagnosis System of Software Defined Networks based on Linked Data Technologies.
Autor:	Manuel García-Amado Sancho
Tutor:	Álvaro Carrera Barroso
Ponente:	Carlos Ángel Iglesias Fernández
Departamento:	Departamento de 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 DE FIN DE MÁSTER

Development of a Fault Diagnosis System of Software Defined Networks based on Linked Data Technologies.

Julio 2018

Resumen

Esta memoria es el resultado de un proyecto cuyo objetivo ha sido desarrollar y desplegar un sistema de diagnostico autónomo de redes definidas por software (SDN) mediante herramientas de análisis de Big Data y enfoques de la web semántica como el conocido Linked Data, varias de ellas desarrolladas en el Grupo de Sistemas Inteligentes.

Para hacer esto, se ha desarrollado un sistema de recolección y procesamiento de datos de SDN, así como un sistema de orquestación de diagnóstico y visualización tanto de los datos de red SDN como de los diagnósticos llevados a cabo.

Hemos elegido un caso de uso concreto donde usar nuestro sistema, un entorno de red SDN emulado y controlado por un controlador SDN como Opendaylight. El prototipo desarrollado recolectará datos de este entorno, realizará un procesamiento de estos datos, de tal manera que se puedan detectar ciertos síntomas en la red basándonos en unos parámetros definidos para el caso de uso y más tarde, introducirlos en un módulo de diagnóstico por fases, la primera de ellas que generaría una hipótesis inicial al recibir un síntoma, y la segunda de ellas que recopilaría más información del entorno para dar una conclusión final.

Además, el prototipo contará con un sistema de visualización basado en tecnologías web como Polymer o D3.js que permitirá a un operador de red una mejor gestión de la red y su diagnóstico.

Como resultado, tendremos un sistema completo capaz de monitorizar y diagnosticar una red de telecomunicaciones basada en SDN que podrá ser interoperable con otros aplicaciones o módulos gracias a el enfoque semántico que se le ha dado al proyecto.

Palabras clave: SDN, Big Data Analytics, Linked Data, SPARQL, Elastic. Opendaylight

Abstract

This thesis is the result of a project whose objective has been to develop and deploy an autonomous fault diagnosis system of software-defined networking (SDN) through Big Data analytics tools and semantic web approaches such as Linked Data.

To do so, a system that recollects and process SDN data, as well as a diagnosis orchestration and visualization system both of the SDN data and the performed diagnoses.

The use case where our system has been deployed is based on a simulated simulated SDN network environment and controlled by a SDN controller as Opendaylight. The developed prototype will collect data from this environment, it will perform a processing of that network data, in such way we can detect symptoms of the network based on defined parameters for the case study. Then, we will introduce symptoms in a diagnosis module by phases, the first of them, to generate a initial set of hypothesis when a symptom is detected and the second, to give a final conclusion collectin more information from the environment.

Furthermore, the prototype has a visualization system based on web technologies such as Polymer or D3.js, that allows a network operator or a user a better network management and its diagnosis.

As a result, we have a complete system be able to monitor and diagnose a telecommunications network based on SDN that can be interoperable with other applications or modules thanks to the semantic approach given.

Keywords: SDN, Big Data Analytics, Linked Data, SPARQL, Elastic, Opendaylight

Agradecimientos

Dar las gracias a Álvaro, por el gran trabajo y apoyo que ha dedicado para que yo pueda presentar este proyecto. También a Carlos por darme la oportunidad de realizarlo en el grupo, así como a todas las personas que trabajan en él, en especial a mis compañeros Alberto, Enrique y Rodrigo.

También dar las gracias a mi familia, a mi padre por inspirarme la vocación. También en especial a mi madre por su apoyo. A Bea por su paciencia y a todos mis amigos, especialmente Fran y Patricia que han estado estos años ahí apoyándome.

Contents

Resumen **VII**

Abstract **IX**

Agradecimientos **XI**

Contents **XIII**

List of Figures **XVII**

1 Introduction **1**

1.1 Context 2

1.2 Motivation 3

1.3 Project goals 4

1.4 Structure of this document 4

2 Enabling Technologies **7**

2.1 Introduction 8

2.2 Software Defined Networking Technologies 8

2.2.1 Openflow 8

2.2.2 Opendaylight 9

2.3 Data Analytics Technologies 11

2.3.1 ElasticSearch 12

2.3.1.1 Searching 12

2.3.1.2 Adding data 13

2.3.2 Logstash and Beats . . . 13
2.4 Semantic Technologies . . . 14
2.4.1 Linked Data . . . 14
2.4.2 SPARQL . . . 15
2.5 Data Visualization Technologies . . . 16
2.5.1 D3.js and other Javascript libraries . . . 16
2.5.2 Sefarad . . . 17

3 Semantic Models **19**
3.1 Introduction . . . 20
3.2 Data Modeling for Networking . . . 20
3.2.1 Background . . . 21
3.2.1.1 Network Description Language . . . 21
3.2.1.2 Infrastructure and Network Description Language . . . 21
3.2.1.3 Network Markup Language . . . 23
3.2.2 SDL: Software-defined networking Description Language . . . 24
3.3 Data Modeling for Fault Diagnosis . . . 27
3.3.1 Background . . . 28
3.3.1.1 B2D2 Knowledge Model . . . 28
3.3.2 B2D2-SDN: B2D2 Model Extension for Software Defined Networking 30

4 Architecture **33**
4.1 Introduction . . . 34
4.2 General Architecture . . . 34
4.3 Data Ingestion layer . . . 35
4.4 Data Processing Module . . . 37
4.4.1 Semantic Converter . . . 37
4.4.2 Symptoms Detector . . . 38
4.5 Diagnosis Semantic Orchestrator . . . 40

4.5.1 Symptom Detection Phase 42
4.5.2 Hypothesis Generation Phase 42
4.5.3 Hypothesis Discrimination Phase 42
4.6 Semantic Visualization Module 44

5 Case study **47**
5.1 Introduction 48
5.2 Network Environment 49
5.2.1 Network Simulation 49
5.2.2 SDN controller: Opendaylight 50
5.3 Data Ingestion 51
5.3.1 Data Sources 51
5.3.2 Data Collection and Storing 52
5.4 Data Processing 55
5.4.1 Preprocessing 55
5.4.2 Data Enrichment 56
5.4.3 Symptoms Detection 59
5.4.4 Storing Semantic Triplets 61
5.5 Diagnosing data 62
5.5.1 Starting Diagnoses 63
5.5.2 Using external reasoning API and diagnosis final report 64
5.6 Visualizing Data 66
5.6.1 Network Diagnosis and Symptoms 67
5.6.2 Network Status and Statistics 71

6 Conclusions and Future Work **83**
6.1 Conclusion 84
6.2 Future Work 84

A Impact of the project **i**

A.1 Social Impact ii

A.2 Economic Impact ii

A.3 Environmental Impact iii

A.4 Ethical and Professional Implications iii

B Cost of the System **v**

B.1 Physical Resources vi

B.2 Human Resources vi

B.3 Taxes vii

C Ontologies **ix**

C.1 SDL Ontology x

C.2 B2D2-SDN Ontology x

Bibliography **xi**

List of Figures

2.1 Opendaylight architecture 10

2.2 Elastic Stack Pipeline 12

3.1 INDL Defined Classes Hierarchy 22

3.2 NML Base Schema 24

3.3 Data Plane Hierarchy 25

3.4 Flow Concept Hierarchy 26

3.5 Control Plane Hierarchy 27

3.6 Main classes of B2D2 Diagnosis Model [11] 29

3.7 Symptoms for a diagnosis model in SDN 30

3.8 A Faults hierarchy in SDN 31

4.1 General Architecture 35

4.2 Activity diagram of the data connector 36

4.3 Semantic Converter structure 38

4.4 Symptoms Detection Model 39

4.5 Activity diagram of the Symptoms Detector 40

4.6 Model of Bayesian Network [11] 41

4.7 Diagnosis Phases 41

4.8 Symptom Detection Phase 42

4.9 Hypothesis Generation Phase 43

4.10 Hypothesis Discrimination Phase 44

4.11 Visualization Module Architecture [24] 45

4.12 Sequence of how visualization module retrieves data 46

5.1 Prototype Architecture 49

5.2 Opendaylight Topology UI 51

5.3 Opendaylight Connector structure 53

5.4 SDL Hierarchy used for inventory nodes 57

5.5 SDL Hierarchy used for topology data 58

5.6 Symptoms detector structure 60

5.7 Storing Semantic Triplets Pipeline Example 62

5.8 Visualization system operation schema 66

5.9 Symptoms Detected 67

5.10 Diagnoses number and its location 68

5.11 Diagnoses Table 69

5.12 Symptoms timeline 69

5.13 Diagnoses timeline 70

5.14 Possible faults timeline 70

5.15 Finding affected paths 72

5.16 Diagnoses for paths 72

5.17 Dashboard Filters and search 73

5.18 Current Topology chart 73

5.19 Chart with the number of network elements 74

5.20 total traffic of the network 75

5.21 Handled traffic for a specific node 76

5.22 Active flows in the nodes 77

5.23 Packets handled/matched comparison 77

5.24 Number of interfaces anomalies monitorized 79

5.25 Timeline for the anomalies detected 79

5.26 State of the interfaces of a node 80

5.27 Interface bitrate 81

CHAPTER 1

Introduction

This chapter introduces the context of the project, including a brief overview of all the different parts that will be discussed in the project. Among them, the SDN technology panorama, but also the trends and possibilities in others influential areas for this project. After this, we introduce the motivation for the development of this project in a few lines.

It will also break down a series of objectives to be carried out during the realization of the project. Moreover, it will introduce the structure of the document with an overview of each chapter.

1.1 Context

The industry in Software-defined Networking (SDN) and OpenFlow has grown in recent years. That is because network management operators can manage devices and resources and suit them to the changing needs of the network, i.e, SDN provides a centralized method to configure more conveniently and flexibly the settings of a network. SDN has its roots anchored in education, where technologies like OpenFlow have allowed researchers to open up networking operating systems for those looking to program the network or alter forwarding behavior. To do that, SDN provides a central point of control. This allow to develop a lot of applications within it, such as traffic engineering, load balancing or the concerning use developed in this project, fault diagnosis applications.

Following a modular architecture, we will not develop a solution totally integrated with a particular SDN controller. This is due to two reasons: not having a system totally dependent on a technology and taking advantage of current powerful technologies, such as Big Data analytics and Linked Data.

Big Data Analytics contributes with very valuable information about the field we are focusing on. Nowadays, the evolution in techniques, methodologies and technologies associated with Big Data is very considerable. So we can take advantage of this fact to boost our systems allowing the processing, search, indexing and visualization of complex network data.

Semantic Web [6] seeks manners to build linked interoperable data that can be automatically processed by software systems. Furthermore, Linked Data initiative are favouring the publication of annotated data in web resources, so that automatic processes can actually consume this data and perform other operations. The publication of ontologies can help integration and standardization of different software data, including data extracted from a SDN controller to describe network resources or control directives.

Now, in this Master Thesis, uniting all these technologies, it has been possible to develop an autonomous diagnosis system of Software-defined Networking.

1.2 Motivation

The motivation of this project comes firstly by the increasing importance of SDN technology, which allows the decoupling of software and hardware levels in networking; it also allows centralized control of the network, simplifying the process of managing the network and allowing the implementation of much more flexible traffic policies. Moreover, for having overall picture of the network, a central controller in SDN is a better way to handle localized problems in networks such as congestion or switch faults.

Secondly, comes by the use of Big Data techniques, which will facilitate processes like monitoring and processing the status of SDN, tasks that implied a certain complexity in legacy networks environments. In the case of monitoring , using Big Data techniques eases us the collection, indexing and storing of the network status data. Having a Data Lake[1] based architecture will allow us to store all structured and unstructured data. Thus we can run different types of big data analytics (real-time analytics, big data processing, visualizations...) in a faster and dynamic way.

SDN permits user create applications or services to be used through SDN controller. This applications are in many cases web APIs that represent data in different formats and ways. Thus in this project we try to achieve a common vocabulary for representation of SDN controllers data improving the interoperability of this applications. Furthermore, diagnosis of telecommunications networks is a particular domain that we will trait using semantic models extended for diagnosis of SDN that allows the prototype to carry out an autonomic fault diagnosis process.

So the aim of this project is to develop a fault diagnosis system using semantic technologies in a SDN, specifically, a environment controlled by a Opendaylight controller. For that purpose, a network will be emulated and some services will be simulated over it. A system to collect data for this emulated network will be created. The data collected of this network, especially the one extracted from the SDN controller, in our case Opendaylight Controller, will be processed, enriched and stored to create an advanced SDN monitoring and fault diagnosis system. This system will have an interactive visualization system that provide the network operator with the status of a SDN, as well as stats and automated fault diagnoses.

[1]More information about this term can be found here: `https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/`

1.3 Project goals

The main goals of this project are:

- Define and develop a module that collects data from an OpenDaylight-controlled SDN simulator with a Python-based framework,
- Extend the recent work in network ontologies to include SDN,
- Process and enrich the collected network data,
- Using semantic techniques, create inference models to allow diagnosing the cause of the fault in a network and
- Create an interactive dashboard for the network administrator to show all this information

1.4 Structure of this document

In this section we provide a brief overview of the chapters included in this document. The structure is the following:

Chapter 1 explains the context in which this project is developed, mainly the SDN environment. Moreover, it describes the main goals to achieve in this project, as well as the motivation and the structure of this document that is being read.

Chapter 2 provides a description of the main technologies on which this project relies, among them, we can find Software defined networking technologies, big data analytics technologies, linked data technologies and frontend visualization technologies.

Chapter 3 describes the semantic models defined for in this project, both for a structural definition of networks and for the domain of diagnosis in telecommunications. First, the background models that we can find in related works are reviewed, second the extensions done for both mentioned fields.

Chapter 4 describes the architecture of the project, including the design phase and implementation details. A general picture of the prototype architecture, as well as the project in which this Master Thesis is framed. Then, all the modules developed in the prototype are widely detailed.

Chapter 5 describes the system evaluated in a case study. In this chapter is explained how the whole prototype is tested for different network simulations and how the developed

visualization system works.

Chapter 6 discuss the conclusions drawn from this project, as well as the problems faced in its development. Finally we will focus in the possible next step to be done for a future work.

CHAPTER 2

Enabling Technologies

This chapter shows a review of the main technologies used in this project, as well as some of the related published works to them. There is a great of variety in implemented control technologies for SDN, in this work we make use of the most extended one, the Opendaylight project, that uses mainly Openflow to send instructions to the switches in data plane. This two technologies are viewed in this chapter.

The new paradigm of Software-defined Networking is not independent of the evolution that is taking place nowadays. Technologies such as Big Data analytics, the great progress in artificial intelligent, with the develop and establishment of advanced machine learning techniques may be incorporated to SDN environments. Also the technologies that make possible the easy exchange of information in the web, the technologies for a semantic web, that aims to get all existing resources in the web linked. In this Master thesis we have use some of them to implement or get an improvement in tasks or applications for SDN such as fault diagnosis.

2.1 Introduction

In this chapter we introduce the technologies that have made possible the development of this Master Thesis. Among them, the SDN technologies, which are the main context of this work, but also the Data Analytics and Linked Data technologies, key technologies to achieve the processing of SDN data and finally a fault diagnosis in SDN environments. In this list of technologies we can not forget visualization technologies, that allow to show the features from our developed application.

In Section 2.2, we describe the SDN technologies used in this project, a brief introduction to the Openflow protocol and the description of the Opendaylight project. Section 2.3 make an overview of the technologies used for data analytics, specifically, the Elastic Stack, a compendium of technologies used in this work for data search and storage purposes. In Section 2.4 we explain the technologies concerning the semantic web, such as RDF or the SPARQL query engine. Finally, Section 2.5 shows the technologies used for data representation, Sefarad, a technology based on Polymer Web Components and D3.js, a Javascript library for rendering advanced charts.

2.2 Software Defined Networking Technologies

In this section the technologies concerning SDN networks used in this project will be briefly introduced. One of these is Openflow, a capital technology to understand SDN networks. Other technology reviewed here will be a SDN controller technology, which allows to create Openflow environments.

The Openflow protocol is briefly viewed in Section 2.2.1. Then, we present the Opendaylight project in Section 2.2.2.

2.2.1 Openflow

Openflow is the communication protocol in SDN environments that enables the SDN Controller to directly interact with the forwarding plane of network devices such as switches and routers, both physical and virtual. Openflow started in 2008 in Stanford University in the wake of the research "Openflow: Enabling Innovation in Campus Networks" [23]. By December 2009, Version 1.0 of the OpenFlow switch specification was released. Since its inception, OpenFlow has been managed by the Open Networking Foundation (ONF), organization dedicated to open standards and SDN adoption.

The ONF defines some Openflow specifications, one of these concerns the description of a Openflow Switch[1] and its requirements. To access and modify configuration data on an Openflow Switch, the ONF defines OF-Config, a special set of rules that defines a mechanism for Openflow controllers communicate instructions to an Openflow switch.

So OpenFlow is a protocol between SDN controllers and network devices, as well as several specifications of the logical structure of the network switch functions and the controller management functions.

The idea of Openflow is based on the fact that all Ethernet switches have flow tables that implements firewalls, NAT, QoS or stats collector. The difference is that each maker has its own type of flow tables in its products, but Openflow bases its behaviour in standard flow tables for all the switches. Through Openflow interface, the SDN Controller pushes down changes to the switch flow table allowing network administrators to partition traffic, control flows for optimal performance, or start testing new configurations and applications.

In February 2011, the first version of Openflow (1.1) is released. Later, upgraded to the 1.2 version, with the supervision of ONF. Nowadays, the last version released is 1.5, but there is not a lot of implementations for it. Because of this, in the project, older versions of the protocol are used.

To sum up, Openflow allows to simplify provisioning, optimizes performance and decouples control plane and data plane, which can accelerate new features and services introduction.

2.2.2 Opendaylight

Opendaylight[2] is an open source collaborative project hosted by the Linux Foundation. Its aim is to enhance software-defined networking (SDN) by providing a shared platform called Opendaylight opened to anyone, including end users, developers and customers.

Opendaylight includes support for the Openflow protocol, specifically, version 1.0, 1.3 and 1.4 until now. But it can also support other protocols, such as NETCONF, YANG, OVSDB or SNMP.

Using YANG models [29] in the Service Abstraction Layer of the Opendaylight controller, it generates APIs for several purposes, as Northbound APIs. YANG models are used to render REST APIs according to the RESTCONF specification [7]. This North-

[1]The Openflow 1.5.1 specification and other versions can be found on the ONF web page: `https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf`

[2]Official Site of OpenDayLight Project: `https://www.opendaylight.org`

bound interface based on REST APIs is common, flexible an open.

Related works to SDN have made analysis of the current panorama of SDN controllers [28]. In most of them, we find that Opendaylight is a powerful SDN controller that allows deploying in several network environments. Also it supplies a wide range of possibilities for traffic engineering tasks.

Specifically, these analysis points out that the main advantages of this controller are the very good documentation published[3], the possibility of integration with Openstack[4] and the large number of use cases to use with it.

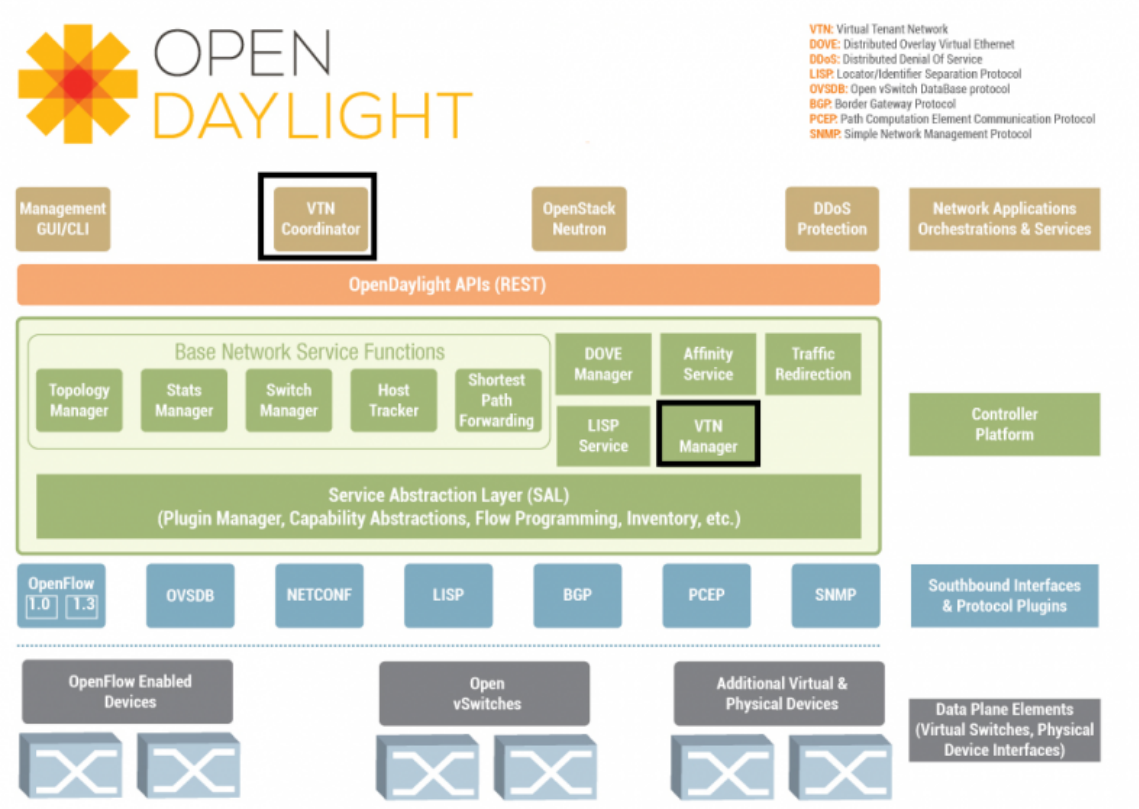


Figure 2.1: Opendaylight architecture

Figure 2.1 depicts the architecture, where it can be seen how southbound interfaces are defined by several plugins for each protocol (including Openflow). This interfaces have an abstraction layer mentioned earlier called MD-SAL. Using Yang models Northbound APIs can be defined, (in the figure in orange).

In conclusion, Opendaylight provides a lot of useful features and can be used as a cross-

[3]Official OpenDayLight documentation: `http://docs.opendaylight.org/en/stable-oxygen/`
[4]Official Site of Openstack: `https://www.openstack.org/`

platform controller, besides being the most widespread open technology.

2.3 Data Analytics Technologies

Data Analytics is nowadays a trending technology that is being used in all kind of systems, no matter the scope. In this section we present the technologies used in this project related to the data management and analysis.

First we introduce the Elastic stack. The Elastic Stack is a compendium of technologies among those that we find Elasticsearch, described extensively in Section 2.3.1, Logstash and Beats, two of them detailed in Section 2.3.2.

Elastic Stack

Elastic Stack is a compendium of open source products from Elastic[5] designed to take data from any source and search, analyze and visualize that data in real time. The product group was known as ELK Stack. "ELK" is the acronym for three open source projects: Elasticsearch, Logstash and Kibana. A fourth project named Beats was subsequently added to the stack. Elastic Stack can be deployed on premises or made available as Software as a Service. Now a brief glance to this different projects is done:

- Elasticsearch is a Java-based RESTful distributed search engine built on top of Apache Lucene[7]
- Logstash is a data collection engine that unifies data from different sources, normalizes and distributes it. The product is optimized for log data.
- Beats are "data shippers" installed on servers as agents used to send different types of operational data or logs to Elasticsearch either directly or through Logstash.
- Kibana is an data visualization and exploration tool for large volumes of streaming and real-time data. The software makes easier the management of huge and complex data streams through graphic representation.

Together, these different open source products are most commonly used for centralized logging in different environments. Logstash collects and parses logs, and then Elasticsearch indexes and stores the information. Finally Kibana presents the data in visualizations that

[5]For more information, try to enter the main web site of Elastic: [6]

[7]The official website of Lucene project: `https://lucene.apache.org/core/`

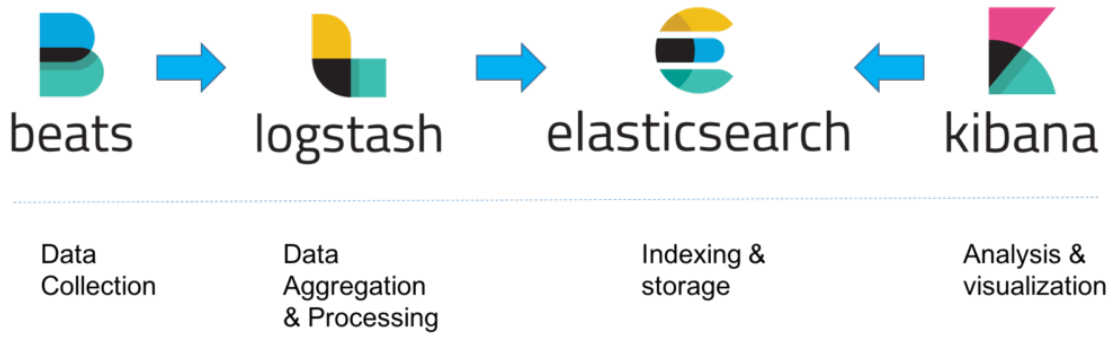


Figure 2.2: Elastic Stack Pipeline

provide actionable insights into one's environment. Figure 2.2 shows how this technologies work together.

In the following, we will see technologies from the Elastic Stack used in this project in detail.

2.3.1 ElasticSearch

Elasticsearch provides a distributed, full-text search engine with an HTTP web interface and schema-free JSON documents. Elasticsearch is distributed, which means that indices can be divided into shards and each shard can have zero or more replicas. Each node hosts one or more shards, and acts as a coordinator to delegate operations to the correct shard(s) [18].

2.3.1.1 Searching

Search API allows you to execute a query and retrieve searched hits that match the query. It can either be composed by a simple string as a parameter, or using a request body.

As a parameter:

```
http://localhost:9200/simulation/node/_search?q=describesNode:openflow11
```

As a request body:

```
http://localhost:9200/simulation/node/_search' -d '{
"query" : {
    "term" : { "describesNode" : "openflow11" }
  }
}
```

2.3.1.2 Adding data

To ingest data easily to Elasticsearch, the fastest method consists in adding a whole index using the Elasticsearch's bulk API. This can greatly increase the indexing speed. Elastic provides API clients for the most common programming languages, such as Python, Java or Nodejs. This example shows how to do that with the Python API.

```
from datetime import datetime

from elasticsearch import Elasticsearch
from elasticsearch import helpers

es = Elasticsearch()

actions = [
  {
    "_index": "tickets-index",
    "_type": "tickets",
    "_id": j,
    "_source": {
        "any":"data" + str(j),
        "timestamp": datetime.now()}
  }
  for j in range(0, 10)
]

helpers.bulk(es, actions)
```

2.3.2 Logstash and Beats

Logstash allows to capture real-time data. With Logstash, it is easy to create ingest pipelines, supporting a variety of inputs that pull in events from a multitude of common sources, all at the same time. Logstash needs a configuration file to works. In this file is described where data is going to be stored and the query terms that are going to be indexed.
Logstash example for catching real-time logs:

```
input {
  beats {
    port => 5044
  }
}

output {
```

```
    elasticsearch {
      hosts => ["${ELASTICSEARCH}"]
      index => "simulation%{[simulation_id]}"
      document_type => "sim_state"
    }
  }
}
```

As we can see, the input data is given by Beats. Beats simplify collecting, parsing common log formats such as, NGINX and Apache and system metrics such as Redis and Docker. In the previous example, beats is configured creating a pipeline to direct logs to Logstash, and Logstash ingest the data in Elasticsearch.

2.4 Semantic Technologies

When we refer to semantic technologies, we are talking about the technologies created in the context of the evolution towards a semantic web that W3C[8] is helping to build. Among this technologies, the most important and extended one is clearly RDF. In this section we introduce the concept and linked data, deepening into RDF standard. Also we detail SPARQL, a language for querying RDF graphs.

So we introduce Linked Data concepts in Section 2.4.1, describing the RDF standard and its kindness. In section 2.4.2 we will talk about the SPARQL query language.

2.4.1 Linked Data

Linked Data[9] is the term used to describe a method of exposing and connecting data on the Web from different sources following the four principles of Linked Data:

- Use URIs as names for things
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL)
- Include links to other URIs. so that they can discover more things.

[8]The semantic web according to the W3C: https://www.w3.org/standards/semanticweb/
[9]Linked Data in W3C website: https://www.w3.org/DesignIssues/LinkedData.html

RDF (Resource Description Language) is a method specified by the W3C for representing information about resources on the Web. It provides a framework for expressing metadata to be exchanged between applications without loss of meaning. The principle of RDF is that every information is expressed in triplets, with a subject, a predicate and an object. A set of such triplets is called a graph. Taking in account that an object of a triplet can also be the subject of another triplet, the possibilities are bigger and complex graphs can be created.

There are several ways of serializing RDF graphs. The most common one is RDF/XML [3], encoding the graph in an XML format. But there are more, such as Notation3 [5], Turtle or JSON-LD [31]. JSON-LD is mainly the one used in this project, which allows us to take advantage of the JSON data structure.

RDF supports definition of new vocabularies through RDF Schema (RDFs). RDFs provides tools to define classes, types, domains, ranges. RDFs follows the subject, predicate and object criteria of RDF as expected. Popular vocabularies as Schema[10] or some vocabularies that defines network structures as NDL [34] or NML [32] are RDFs defined vocabularies. This vocabularies will be used in this project.

Also W3C provides Ontology Web Language (OWL) [2]. OWL adds additional semantics based on Description Logics and complex class construction. OWL is used in this project to extend some existing vocabularies and ontologies with the purpose of covering the field that concerns us.

Data defined with RDF framework are stored in a triple store. Some of the most common Triple stores are Apache Jena Fuseki[11] or Openlink Virtuoso[12].

W3C also has standardized language to query and manipulate RDF data contained in a triple store. This language is an extension of SQL, SPARQL[13]. SPARQL is also a tool that will be used in this project for query reasoning and inference. Now let see this tool in detail.

2.4.2 SPARQL

SPARQL is an SQL-like query language for RDF. It uses a simple syntax to specify variables and triplet templates for retrieving information from an RDF store [25].

Throughout this project we have used Apache Jena Fuseki [21], a triplets store and a

[10]The schema language URI with its entire vocabulary: `http://schema.org/`
[11]Official Site of Apache Jena project: `https://jena.apache.org/index.html`
[12]Official Site of Virtuoso: `https://virtuoso.openlinksw.com/`
[13]W3C recommendation for SPARQL: `https://www.w3.org/TR/rdf-sparql-query/`

SPARQL server. Fuseki provides REST-style SPARQL HTTP Update, SPARQL Query, and SPARQL Update using the SPARQL protocol over HTTP [14].

It was made official W3C Recommendation on January 15, 2008. SPARQL queries allow conjunctions, disjunctions, and optional patterns.

The result can be represented in many formats, such as HTML, XML, JSON-LD ...

An example of SPARQL query is given in listing 2.1.

Listing 2.1: ***SPARQL query example that retrieves two interconnected hosts [35]***

```
PREFIX ndl: <http://cinegrid.uvalight.nl/owl/ndl-topology.owl#>

SELECT ?host1 ?host2
WHERE {
    ?if1 ndl:connectedTo ?if2 .
    ?if2 ndl:connectedTo ?if1 .
    ?host1 ndl:hasInterface ?if1 .
    ?host2 ndl:hasInterface ?if2
}
```

2.5 Data Visualization Technologies

2.5.1 D3.js and other Javascript libraries

D3.js is a JavaScript library for manipulating documents based on data. D3 helps you show data using standard web technologies HTML, SVG, and CSS. D3's emphasis on web standards gives you the full capabilities of browsers without tying yourself to a concrete framework, combining visualization components and a data-driven approach to DOM manipulation.

D3 allows you to bind arbitrary data to a Document Object Model (DOM), and then apply data-driven transformations to the document. For example, you can use D3 to generate an HTML table from an array of numbers. Or, use the same data to create an interactive SVG bar chart with smooth transitions and interaction.

D3 is not a framework that exploits the full capabilities of web standards such as HTML, SVG, and CSS. With minimal overhead, D3 is enough to support large datasets and dynamic behaviors for interaction and animation. D3's functional style allows code reuse through

[14] Official Apache Jena Fuseki Documentation: `https://jena.apache.org/documentation/serving_data/`

a diverse collection of components and plugins. This feature allows us to draw all kind of graphs, such as network graph[15]. The aim of using this technology is integrating it within 2.5.2, a technology for developing dashboard visualizations.

Besides D3, in this Master Thesis we will use "sgvizler" library for visualization [30]. This is a Javascript library that allows to represent graphically information extracted from SPARQL queries. To achieve this, sgvizler[16] use the Google Visualization APIs to retrieve google charts. Other framework that use google charts and webcomponents for represents data is Sefarad, detailed in the next section.

2.5.2 Sefarad

Sefarad is a web-based data visualization and browsing application implemented by Intelligent Systems Group (GSI) at Universidad Politécnica de Madrid (UPM) for consultation and visualization of semantic data. It is developed to explore linked data by making SPARQL queries to the chosen endpoint without writing more code. In this way, it provides a semantic front-end to Linked Open Data.

It allows the user to configure his own dashboard with many widgets to visualize, explore and analyze graphically the different characteristics, attributes and relationships of the queried data.

Sefarad is based on Web components. Web Components[17] are a set of standards currently being produced by Google engineers as a W3C specification that enables the creation of reusable widgets or components in web documents and web applications. The intention behind them is to bring component-based software engineering to the World Wide Web. The component model enables encapsulation and interoperability of individual HTML elements.

For more information about this technology, Sefarad has a demo online[18] and a complete documentation repository[19].

[15]More information in the gallery of D3 charts: `https://github.com/d3/d3/wiki/Gallery`
[16]Sgvziler Documentation, largely detailed: `http://mgskjaeveland.github.io/sgvizler/`
[17]The official site of Web Components: `http://webcomponents.org/`
[18]Demo online of Sefarad: `http://sefarad.cluster.gsi.dit.upm.es/`
[19]Sefarad Online Documentation: `http://sefarad.readthedocs.io/`

CHAPTER 3

Semantic Models

Software-defined Networking is a recent technology. There are many current implementations of this kind of networks and the panorama is still heterogeneous. Making use of the linked data technologies, we have proposed to extend the RDF models present in the literature in order to generalize SDN concepts and its data representation. To do so, a review of the existing models to define networks is made, and then, we introduce an extension done for a network data model to introduce SDN concepts.

It has also been seen that the RDF domain models referred to diagnosis in telecommunications networks need more information to be useful for diagnosis in SDN systems. Thus, first, we will resume a domain model from previous researches that describes the fault diagnosis of telecommunication networks. Second, we present an extension for this model, to adapt it to the technology that we are using in this project, SDN

3.1 Introduction

The aim of this chapter is to present two extensions made in order to represent SDN data in a semantic way. First of this two extensions is referred to the infrastructural model of a network based on SDN. We did a retrospective of the existent semantic models based on RDF that try to define classic telecommunication networks and then, we try to achieve an extension for these models taking into account which were the most important concepts to add. The second of this extensions refers to the fault diagnosis domain for telecommunication networks. We have chosen an already defined model and we have made an extension of it in order to adapt it to a SDN environment.

SDN environment is heterogeneous today. The amount of SDN and Openflow based technologies has grown in recent years. There are different controller technologies. In most of them, the common point is that they use Openflow as standard to communicate with the data plane, i.e., the network elements such as switches or routers [28].

There are two approaches in SDN controllers [26]. The non-proprietary strategy, and a lot of proprietary solutions. In these two cases, it is undeniable that Openflow has had a capital impact, being instrumental to commoditize the switches and routers.

The non-proprietary strategy sustains that the main purpose of SDN survives by being available to all as an open source provision with no hidden proprietary components. They defend the openness at the cost of isolated overlay network from the physical network at the bottom. From this approach, it is worth stressing an important project launched by the Linux Foundation, Opendaylight, but there are more, as ONOS, POX, etc.

To extend this models, we focus on this non-proprietary approach.

This chapter is divided in two main sections. In Section 3.2 we present the overview of the semantic models present in the literature for modeling network infrastructures and expose the extension done based on these semantic models to introduce concepts from SDN within them. Once again, we deep in the models defined for Fault Diagnosis in Section 3.3 and also the extension done to cover the diagnosis in SDN environments.

3.2 Data Modeling for Networking

Building a model for networks is not a new thing in the literature, so we dive into the RDF defined models for networks that has been developed. This is the basis our model was built upon.

The aim of this section is to present a extension done for modeling SDN infrastructures. To do that, we review the previous defined models in the literature in Section 3.2.1. Then, the extension itself is presented in Section 3.2.2.

3.2.1 Background

There are many ways to model network infrastructures. In this section, we take a glance at existing work that models computer networks using Semantic Web technology, specifically, the main RDF-based models created for networks.

Section 3.2.1.1 describes the *Network Description Language* and Section 3.2.1.2 an extension of it, *Infrastructure Network Description Language*. Section 3.2.1.3 presents a standard model for networks domain that is the evolution of the preceding models, *Network Markup Language*.

3.2.1.1 Network Description Language

The first model adopting the Semantic Web for its schema, and in particular RDF, is found in the Network Description Language (NDL) [35]. This model was developed in the mid 2000s at the University of Amsterdam. This language was a simple ontology based on the distributed descriptions and network properties of the FOAF project [10]. NDL defines three main classes: **Location**, **Device** and **Interface**, and six properties: **locatedAt**, **hasInterface**, **connectedTo**, **description**, **name** and **switchedTo**.

NDL vocabulary is created with the aim of embracing a particular use case [34]. NDL solved many of the issues that operators and users face in the domain of the hybrid optical networks.

Except the previous use case, the Network Description Language did not have more repercussion on the research community, but it serves as the basis for other languages and standards. In the next section, we will clarify this point presenting languages and standard which were developed based on NDL.

We can see an example in Listing 2.1 where the NDL ontology is used to retrieve specific information of the network, such as two connected hosts.

3.2.1.2 Infrastructure and Network Description Language

An extension of Network Description Language is defined by the University of Amsterdam in 2012, the Infrastructure and Network Description Language (INDL) [17]. The aim of

INDL is to provide technology independent descriptions of computing infrastructures.

As well as NDL introduces a model for network infrastructures, INDL focus on the computing infrastructures, but including the concepts introduced in NDL, such as physical resources and the network infrastructure that connects these resources. Also, INDL provides the necessary terms to describe virtualization of resources and the services offered by these resources.

The INDL ontology includes two main classes, **Resource** and **Service**. The three subclasses of **Resource** are **Node**, **Network Element** and **NodeComponent**. The class **Service** represents the Service that are provided by a node, it depends on the domain in which the vocabulary is applied, some defined are **StorageService**, **StreamService** or **DisplayService**. In the case of virtualization, INDL defines the **VirtualNode**, which is modeled as a subclass of **Node**. Figure 3.1 shows this concepts as a hierarchy tree.

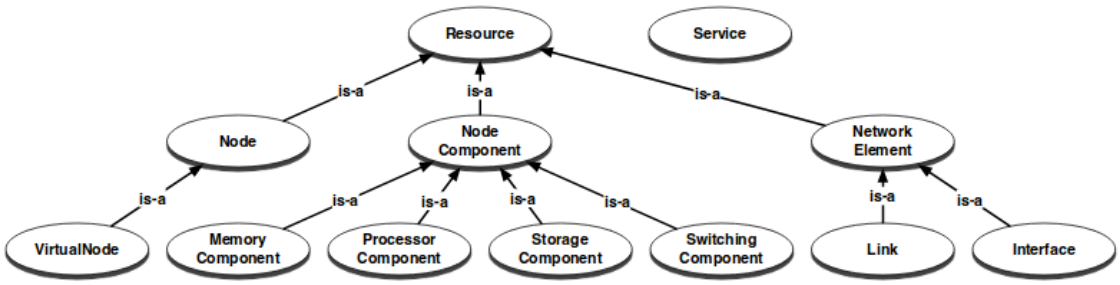


Figure 3.1: INDL Defined Classes Hierarchy

So INDL models the internal components of a **Node** with the abstract class **NodeComponent**, which describes the essential components of a computing node: **Memory Component**, **Processor Component**, **Storage Component** and **Switching Component**. Otherwise, it groups the classes defined in NDL in an abstract super class **Network Element**, to separate the connectivity from the functionality.

In fact, the key feature of this information model which makes it reusable and easy to extend is that virtualization, functionality and connectivity are decoupled pieces of a set. So, we can add new connectivity items, such as Network Elements without impacting how we model its functionality or virtualization.

3.2.1.3 Network Markup Language

The Network Markup Language Working Group - NML-WG - within the Open Grid Forum[1] worked towards a first standard where the modeling effort done in NDL and INDL were incorporated. This standard arrived in 2013, the Network Markup Language (NML) [32]. NML was defined in the context of research and education networks to describe network topologies. As the NML standard states:

> *The Network Markup Language is designed to create a functional description of multi- layer networks and multi-domain networks [...] NML can not only describe a primarily static network topology, but also its potential capabilities (services) and its configuration [...]*

That is, the sum of efforts done in the confection of NDL to define heterogeneous network infrastructures and INDL to separate the plane of functionality and connectivity.

Fruit of this, the standard [33] describes in its section number two the NML Base Schema, a complete information model for computer networks. NML schema is represented with two syntax. XML syntax and OWL RDF/XML syntax. We focus in the RDF-based syntax for our project.

Figure 3.2 the UML class diagram of the classes in the NML schema and their hierarchy. The schema consists of several classes, attributes, relations and parameters. The main classes of NML schema, as in INDL, are the **Network Object** with subclasses like **Port** and **Node** and the **Service**. NML adds an abstract class **Group**, with subclasses as **Topology**, an important concept introduced in this standard. The schema counts with a lot of relations and attributes to link this classes and enrich the vocabulary, allowing an extended description of a network infrastructure.

It is important to say that the schema omits a definition for the term *Network*, due to the wide uses it has. Instead of this, the standard proposes using **Topology** for the concept of a network domain and a **Link** with multiples sources and sinks for the concept of a local are network.

As its predecessors, this schema may be extended, or sub-classed as pointed in the document specification, to represent technology specific classes. This has motivated us to take NML schema as one of our basis to define or model for SDN.

> *NML only attempts to describe the data plane of a computer network, not the*

[1] Official Site of The OGF: `https://www.ogf.org/ogf/doku.php` (Standards released can be found here)

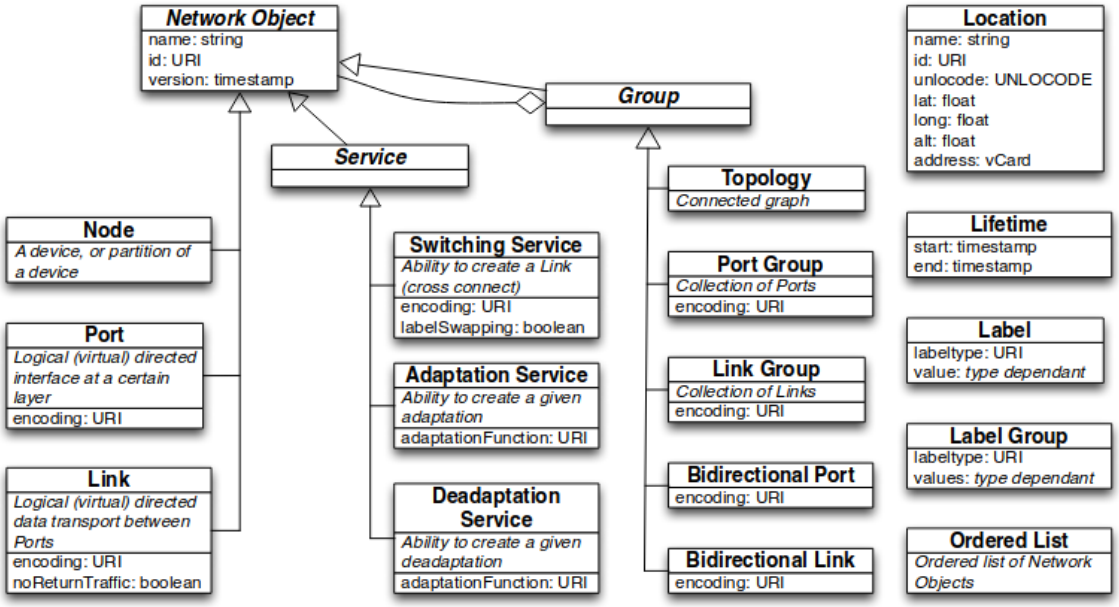


Figure 3.2: NML Base Schema

> *control plane. It does contain extension mechanism to easily tie it with network provisioning standards and with network monitoring standards.*

We can see that, as it is argued in the NML recommendation, the NML schema is not prepared to describe the control plane of a network infrastructure, but this is a point we will argue in the next section.

Finally, it should be noted that all the terms defined in the NML schema are related to the ones in the ITU-T G.800 recommendation [G.800] [27].

3.2.2 SDL: Software-defined networking Description Language

In this section we will define a semantic model to achieve a common description for the environment of SDN and Openflow. The aim of creating this model is to provide technology independent descriptions of the different technologies mentioned previously. In this section, we give a vision of the model defined for this purpose.

We have defined the Software-defined networking Description Language (SDL). The aim of SDL is to extend with new concepts, extracted from the current SDN panorama, the existing semantic models defined for telecommunications networks. Because of NML is the most recent model and contains all the wisdom from INDL and NDL, we mainly have extended our model directly from the NML base schema classes [33].

The SDL schema defined consists of classes and properties. In this section, the main classes of SDL are described. Firstly, we introduce the concepts introduced for defining the data plane of an SDN network infrastructure. Secondly, we present the ideas introduced in SDL, trying to approach the control plane of a SDN network.

NML is mainly a schema for defining the data plane of a network, as mentioned in section 3.2.1.3, so we have inherited most of the classes from its schema. In it, it was defined the class **Node**. We have grounded this concept for SDN-based environments in which Openflow 2.2.1 is the main protocol used. Thereby, figure 3.3 shows the worked done extending the NML hierarchy.

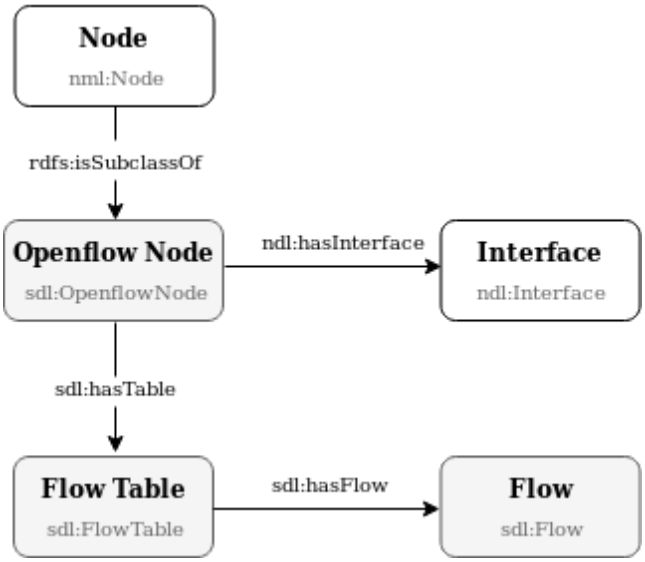


Figure 3.3: Data Plane Hierarchy

The concept **Openflow Node** corresponds to a node with Openflow capabilities. The essential feature of an Openflow switch is having a **Flow Table**, so we had to create that concept and a property that links the Openflow node with its flow table. An Openflow switch is still a switch, therefore it has interfaces, modeled in this case with the NDL ontology. Lastly, flow tables contains flows, an important concept also introduced in SDL in the way shown in figure 3.4.

As it can be shown in figure 3.4, the class **Flow** can have an **Instruction** which in turn can have one **Action**, a **Match** and other fields as a **Timeout** or **Priority**. Since 1.3 version of Openflow, nodes can support also statistics by flow, so we reflect this fact including the class **Flow Stats**. It has to be emphasized the importance of this part of the information model.

When it comes to covering concepts concerning Openflow approach, we should consider

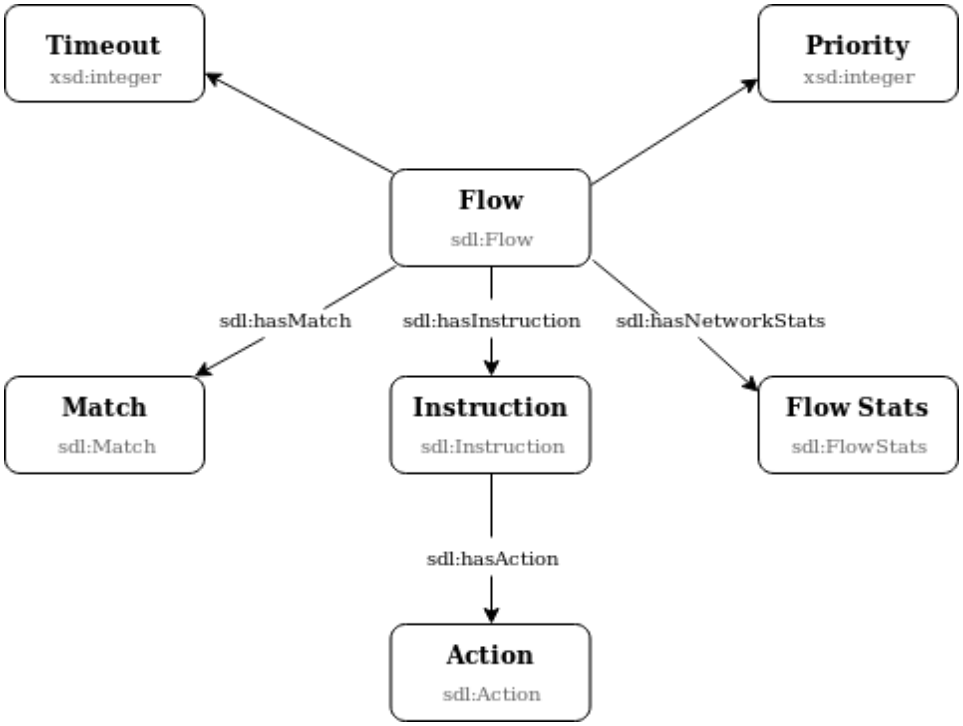


Figure 3.4: Flow Concept Hierarchy

that the more versions of Openflow are released, the more complex an Openflow switch is. Through the definitions in figure 3.4, we are including only some initial concepts concerning the instructions and matches that a controller establishes in an Openflow switch, in favor of the generality of the model. Despite this, this part of the model is of great importance when performing tasks of diagnosis or detecting anomalies, as we can see in recent research studies [1].

On the other hand, the control plane of an SDN network should be also defined. As it was foregrounded in section 3.2.1.3, NML does not take into account the control plane concepts, so the extension of this part of the model is done with the INDL ontology. INDL embraced abstract concepts to define functionality aspects in network infrastructures, so we have followed a similar approach to define this kind of concepts.

The idea of approaching the control plane in SDN networks had to be based on the class **Controller**. Initially, the idea was to define the interfaces a SDN Controller has, the Northbound and Southbound[2]. This idea was dropped due to we are not interested in the internal controller behavior, but we focus on reflecting how it provides instructions to its controlled network in SDN.

[2]For more information about Northbound and Southbound Interfaces in SDN: `https://whatis.techtarget.com/definition/northbound-interface-southbound-interface`

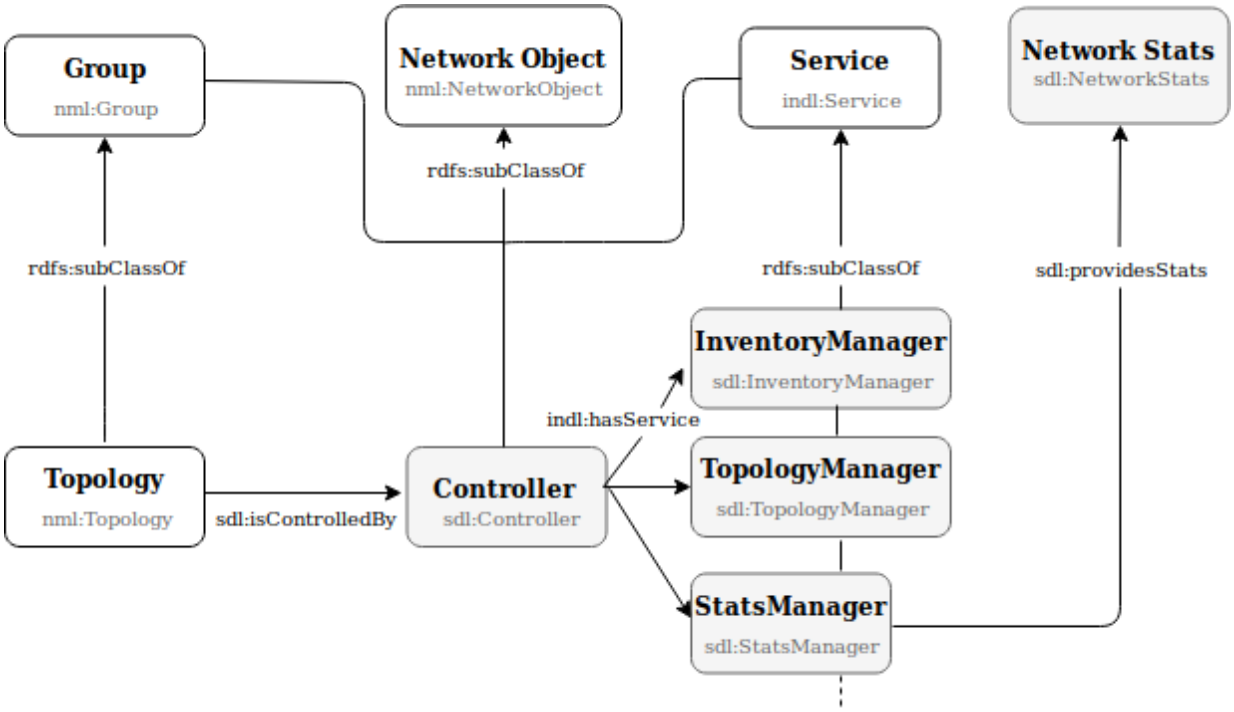


Figure 3.5: Control Plane Hierarchy

Figure 3.5 presents this concepts. It is remarkable the definition of the class **Network Stats**. This stats are provided by the controller using a **StatsManager** service. In figure 3.4 we can see an inherited child of Network Stats, Flow Stats, but there are more in the model, because stats is a key feature in SDN that could be defined.

We have described the main classes and properties in the model, but the model is much more rich in terms, in order to embrace more concepts of software-defined networking. The complete ontology in RDF/Turtle format can be found in the appendix C.1.

3.3 Data Modeling for Fault Diagnosis

Fault diagnosis tasks in dynamic networks scenarios are often quite complex tasks. Furthermore, the constant increase in the size and complexity of the network makes fault diagnosis a critical task that should be handled quickly and in a reliable way. So the automation of these tasks allows to obtain better results in its execution. To achieve this automation, we need to design our automatic fault diagnosis systems defining or following a diagnosis model. An automated agent will follow this model to carry out a diagnosis in the system.

The aim of this section is to extend a current diagnosis model for agents to include possible faults given in SDN environments. Also we extend the possible symptoms that an

agent or system can detect in the network to add more that this agent could observe in a dynamic network environment based on SDN. To do that, first we review this model, the *B2D2* model in Section 3.3.1, then, we define the extension, called *B2D2-SDN*.

3.3.1 Background

This section reviews a knowledge model, the B2D2 knowledge model [11]. We based our work in this model, extending it (in the next section) to bring it closer to SDN technology. So first, in Section 3.3.1.1 we give a brief overview of B2D2, specially, to its Diagnosis Model.

3.3.1.1 B2D2 Knowledge Model

A knowledge model is composed of different models, describing different aspects of a specific problem. In this case, B2D2 model is divided in two main models: a domain model and an inference model. The domain model describes the main static information and knowledge in an application domain, in our case, fault diagnosis for telecommunication networks. The inference model is used to define the tasks required to carry out a diagnosis process.

B2D2 presents in its domain model an Structural Model that contains knowledge about the network itself using Infrastructure and Network Description Language (INDL). For us, this structural model does not contains enough knowledge for the task of defining an SDN network, so we use the model defined in section 3.2.2.

Also, B2D2 offers complementary views of the domain knowledge including a Causal Model that relates the symptoms with the possible fault root causes while handling uncertainty using Bayesian networks, as specified in this work [12]. Causal models uses evidences collected from the environment (in our case, symptoms from the telecommunication network) to discriminate between the possible fault root causes.

As a Structural Model, a Causal Model is built based on the domain knowledge extracted from the network modeled as evidence variables. In the same way, the possible fault root causes or any context information are included in the causal model as variables completing the variables set. So we need to extend the model with the symptoms and possible fault root causes concerning a SDN network. This extension will be explained in section 3.3.2

Regarding the inference model, B2D2 takes a generic Task Model [20] composed by tasks. For this scope, the Fault Diagnosis task is decomposed into three subtasks: *Symptom Detection*, finding out whether complaints are indeed symptoms, *Hypothesis Generation*, generating hypotheses of possible causes based on the symptoms, and *Hypothesis*

Discrimination, discriminating among the hypotheses based on additional observations.

This three models found in B2D2, the Structural Model, the Causal Model and the Task Model lead to a model for an autonomic fault diagnosis process. This Diagnosis Model is formalized as an ontology[3].

The main concept on this model is the **Diagnosis** which is performed by **actors** that execute **actions** to collect **observations** from the monitored network. From this **observations**, a set of **hypothesis** is generated and discriminated until a **conclusion** with enough confidence is reached. This is a simplified overview of the model, the most important concepts in the model are shown in Figure 3.6.

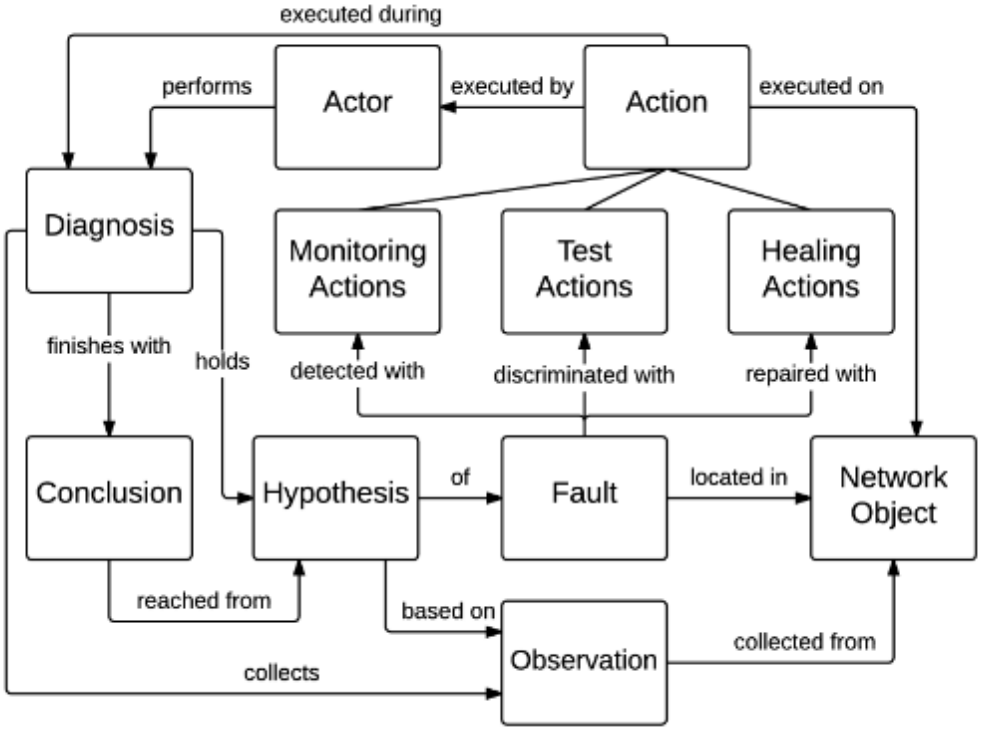


Figure 3.6: Main classes of B2D2 Diagnosis Model [11]

In the next section we will describe the extension done for this Diagnosis Model to adapt it to SDN technology.

[3]B2D2 Diagnosis Model ontology URI: `http://www.gsi.dit.upm.es/ontologies/b2d2/diagnosis`

3.3.2 B2D2-SDN: B2D2 Model Extension for Software Defined Networking

Once we have reviewed the B2D2 knowledge model and the Diagnosis model defined within it, we are going to present in this section an extension done for this model in order to define possible symptoms in a SDN network an its possible fault root causes.

In this work [4], a retrospective of the research in fault diagnosis in SDN scenarios is done. In it, we can see that most of the research focuses on links/switches fail-over due mainly to the decentralizing tasks that is given in SDN environments. Nevertheless, the focus of [4] is different, extending failure diagnosis to faulty traffic configurations which could interference with provision of specific services within the network.

In this project, as part of the investigation reflected in this work [4], we have followed the same approach regarding to fault diagnosis extending the model presented in 3.3.1.1. According to this model and its tasks hierarchy, the objective of the task Fault Diagnosis is finding a conclusion through a Hypothesis Discrimination. But first, you need an Hypothesis Generation from a Symptom Detection inferred with data provided by the SDN Controller services, as the stats or manager modules.

So first, we define the symptoms in a SDN network that could interference with provision of specific services. Figure 3.7 shows the concept Symptom, defined in the diagnosis model of B2D2, with its corresponding defined sub-classes. That extension classes are shown in the figure with the b2d2-sdl prefix.

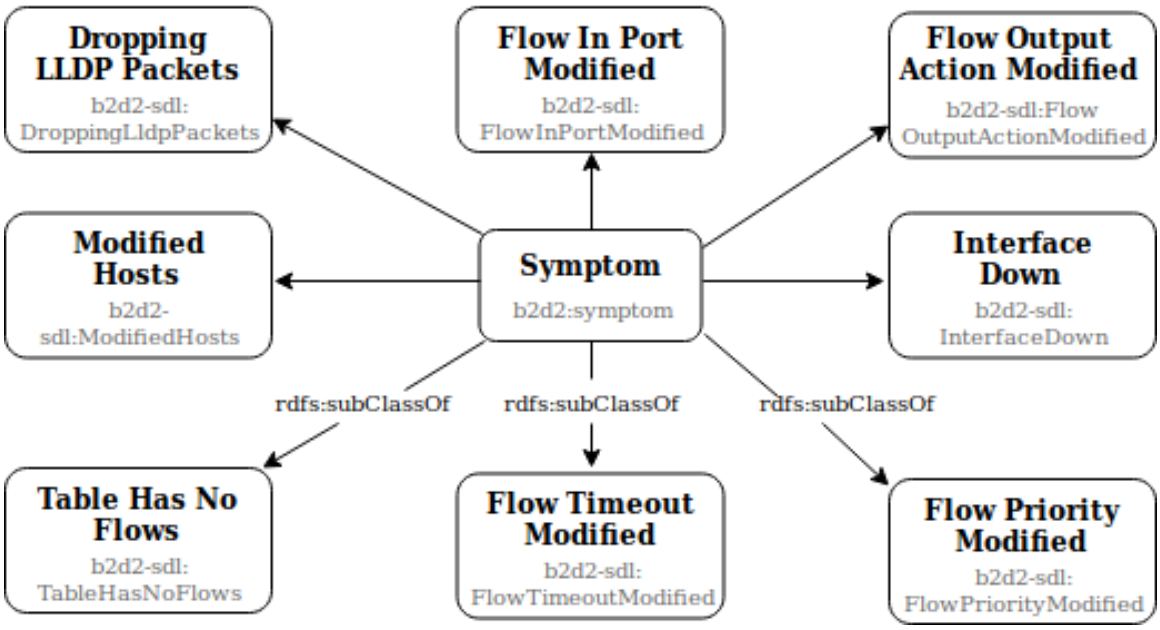


Figure 3.7: Symptoms for a diagnosis model in SDN

Second, we define the other need to enrich the causal model, the possible faults. Figure 3.8 shows a faults hierarchy introduced inheriting from the class Fault in the diagnosis model. The hierarchy define some general faults that could be also for a legacy common network.

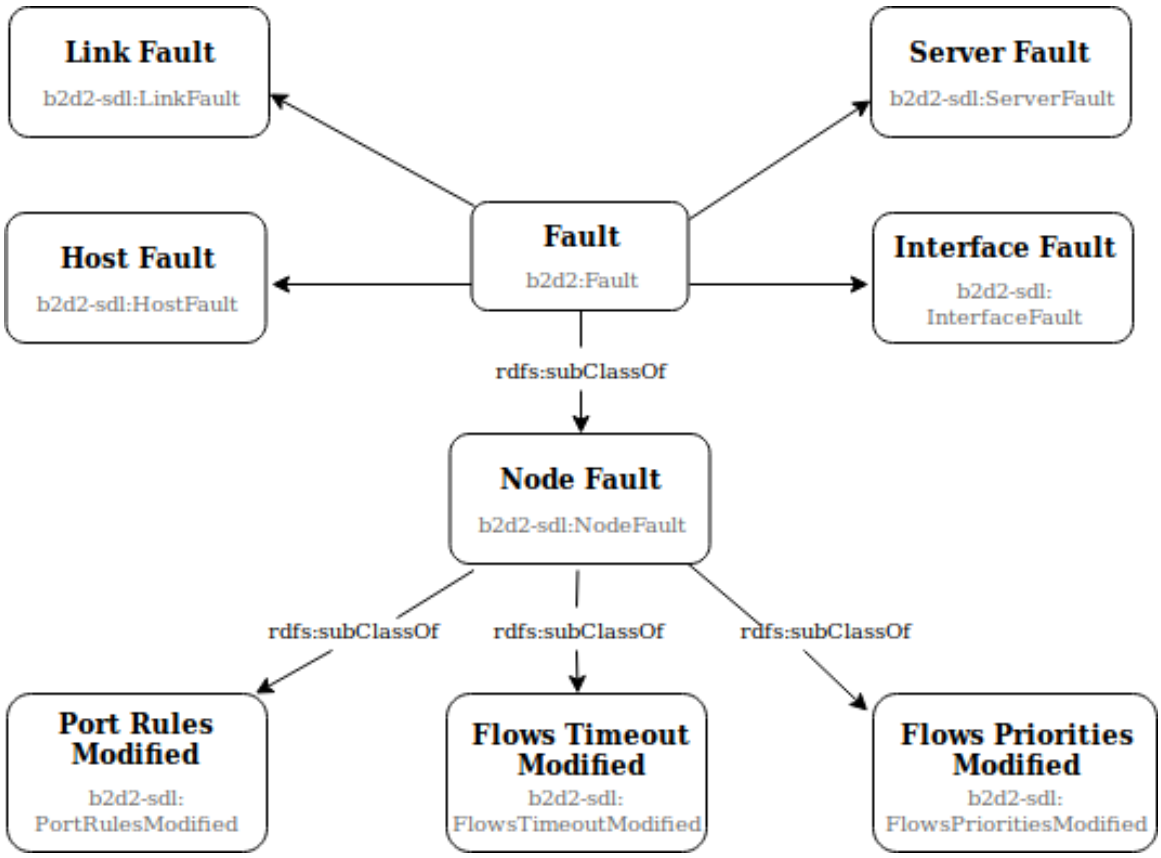


Figure 3.8: A Faults hierarchy in SDN

This concepts embraces among others, the fact that a data center server from the network is disconnected, or that a node is shut down. But the main objective of this hierarchy is defining the possible misconfiguration that a controller is sending to a node, or the node itself is generating. Among others, we could find that somehow timeouts are added in a node, or the priorities or rules are been changed in a node.

We have defined the symptoms as evidence variables in our SDN environment and the possible fault causes. Because we are based the extension of the diagnosis model in the approach given in this work [4], the causal model to inference a certain fault given some symptoms, can be done with he model developed based on Bayesian reasoning. It makes an heuristic model that relates symptoms and cause of fault, inferring a hypothesis of the most probable status of the network. This status will be predicted according to a Conditional Probability Table (CPT), where the conditional relations between evidences

and the possible faults are stored. To model this probability, we will use Probabilistic OWL (PR-OWL), that extends the OWL language to represent probabilistic ontologies, in this case, for describing the Bayesian networks of the causal model. For further information, please refer to the complete PR-OWL ontology[4] or to the work of Bayesian causal model complete development [4].

[4]Pr-owl ontology URI: http://www.pr-owl.org/

CHAPTER 4

Architecture

In this Master Thesis we have created a system capable of carrying out a fault diagnosis in a telecommunication network based on SDN. This system will support technologies based on Linked Data techniques and Big Data Analytics that will allow to endow the system with a great capacity. It will also have a visualization module to make the most of all its features.

This chapter describes the architecture of this prototype, including the design phase and formal implementation. Firstly, we will present a global vision about the architecture, identifying the different modules that integrates the system. Secondly, we will describe the modules which compose the prototype, showing its functionality in depth and how they have been put together.

4.1 Introduction

The development of this project comes under BayesianSDN [1], a larger Spanish project co-financed by MINETAD [2] and FEDER [3], where GSI-UPM is a project partner. This project aims to generate different monitoring and diagnosis techniques based on intelligent systems to build a system capable of learn and adapt to the changes and evolution in a SDN network.

The general architecture of this master thesis is based on the one presented for the BayesianSDN project. That architecture is based on a Data Lake Architecture which ingests data from different sources, such as application monitoring systems or SDN controllers which are used to monitor the network behaviour. Data collected from these sources is ingested to the Data Lake and analyzed applying Big Data techniques to provide refined and useful data and models.

So this chapter is divided in five main sections. The general architecture of the system developed is presented in Section 4.2. Section 4.3 describes the data ingestion layer of the system. Section 4.4 describes the module that includes all tools to manage the data processing tasks. Section 4.5 presents a coordination module that process semantic data, and the last section, Section 4.6 describes the visualization module developed.

4.2 General Architecture

In this Master Thesis, some of the modules from the Bayesian SDN project have been developed or extended in order to achieve a fault diagnosis system of SDN based on semantic technologies. We present the modular architecture of the system in Figure 4.1.

As it can be seen, the most striking is that the architecture is based on a Data Lake. The data collected from the data ingestion layer are ingested to the Data lake and analyzed applying Big Data techniques. The system collects data from SDN environments and store this data in the Data Lake. The rest of the modules will use the Data Lake to retrieve or ingest information in it, depending on the situation and the operation regime. In the next section we describe the operation of each module of this architecture.

[1] The description of the project in the GSI Main Page: `http://www.gsi.dit.upm.es/index.php/es/investigacion/proyectos`

[2] The Offical Site of Minetad in spanish: `www.minetad.gob.es/es-ES`

[3] More information in the funding page of the FEDER: `http://ec.europa.eu/regional_policy/es/funding/erdf/`

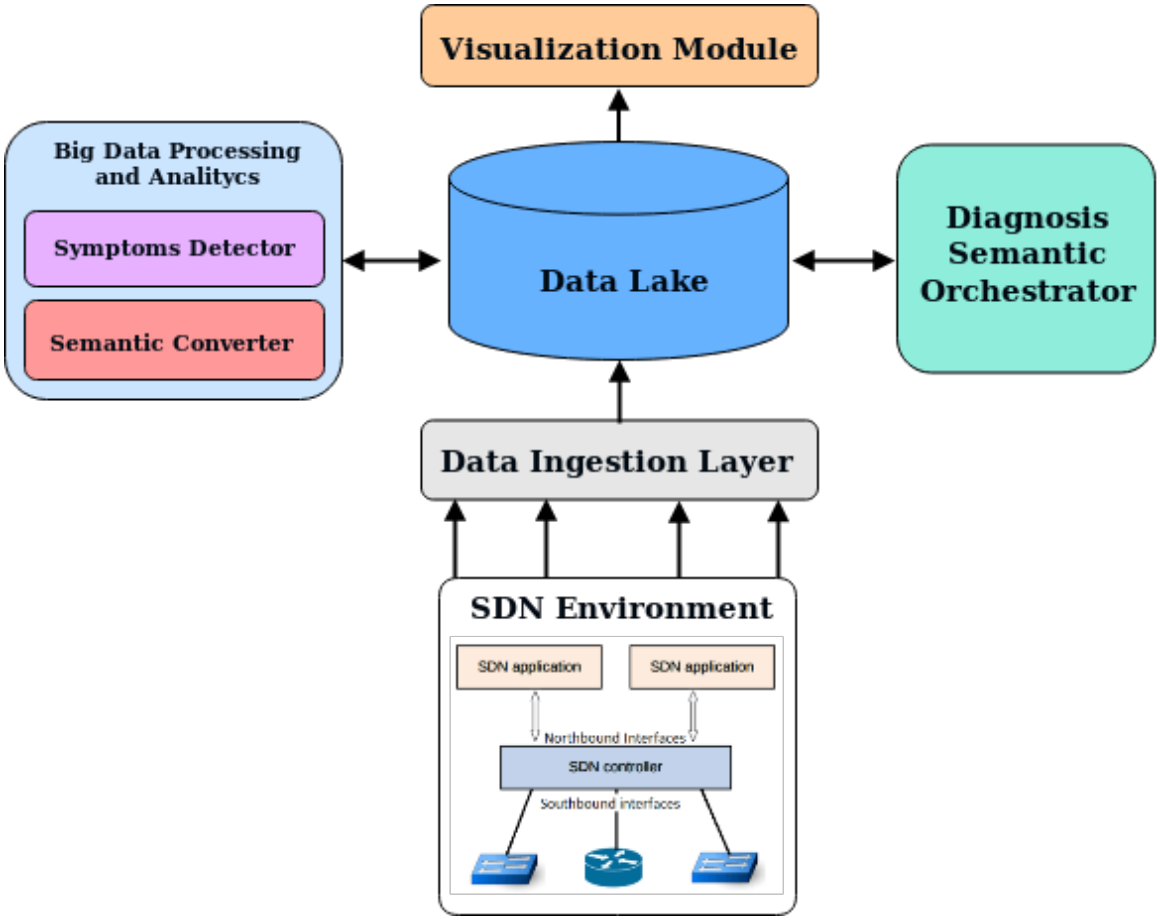


Figure 4.1: General Architecture

4.3 Data Ingestion layer

Classic Internet-based monitoring systems defined for integrated network management used SNMP[4] protocol for this task. Having the control plane centralized in an SDN network eases this process reducing the complexity of the problem significantly.

In general, in monitoring tasks, choosing polling as a way of get the state of the monitored machines can entail problems of network overload or excessive data noise. First of this problems is not happening in our system thanks to the SDN approach. The second of this problems could emerge in our prototype but we trait it later in the data processing module (Section 4.4).

The aim of a data ingestion layer is to collect data from data sources and to store it into the Data Lake. This task will be performed by its data connectors. Since each data source presents different ways of accessing its data, the design for each data connector will

[4]Official Site of SNMP protocol: http://www.snmp.com/protocol/

have different approaches, so we need to define a different connector for each data source. For example, in this Master Thesis, we will develop a connector for the SDN controller Opendaylight (The so-called ODL Connector[5] in Fig. 5.1)

This connectors will collect data periodically from the source, if the source is available. Depending on the system needs or the use case, this recollection could be in batch mode or it could also be in streaming, in order to keep a constant flow of data from the data source.

The connector need a database for storage. In our architecture, raw data will be stored in the Data Lake. The Data Lake consists in having a place where we can store both structured and unstructured data, regardless of its origin or purpose.

The Data Ingestion Layer controls the operation regime of this data connectors. Figure 4.2 describes this operation regime in a schematic way. The connector receives the order from the Data Ingestion layer to start collecting data. If the data collected does not contain information, the system reports an error on the data source selected, but also store that, since will be useful for the diagnosis module. Once it knows the composition of the batch of data, the Data Lake connection is tested. If there is no connection with the Data Lake, the system wait until a previously configured timeout, if the Data Lake is perfectly reached, the data connector send the data to it and then, waits a preset time until the querying of more information.

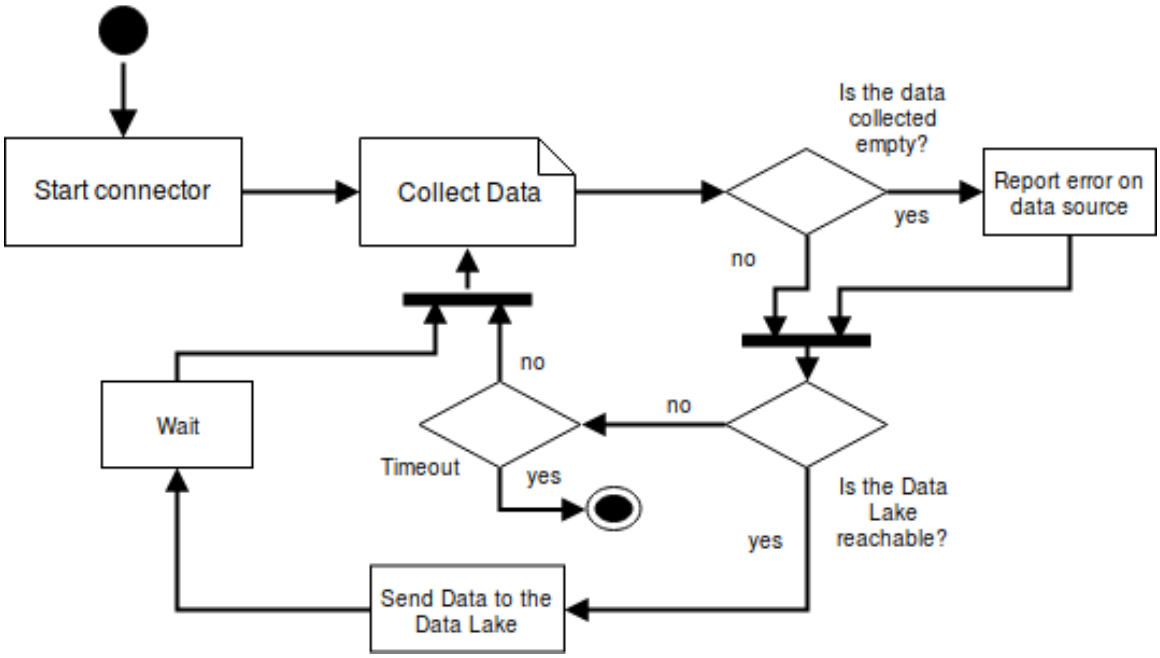


Figure 4.2: Activity diagram of the data connector

[5]For more information, refer to Chapter 5

4.4 Data Processing Module

In this section we describe the data processing module that can be seen in the system architecture (Fig. 5.1). This module is composed by two sub-modules or layers. First of all, a semantic converter that enrich the data present in the data lake collected by the data connectors will be described in Section 4.4.1. Second, a processing task that detects symptoms from the data semantically converted is detailed in Section 4.4.2.

4.4.1 Semantic Converter

Semantic web and RDF [6] provides powerful features to work with graphs and linked resources.To create an application based on semantic technologies it is necessary to structure data following the semantic web canons. Also we will need a site to store this semantic data, so we introduce a new component to store RDF triplets. Having data structured and stored in a RDF store would enhance collaboration with external data units and diagnosis modules, improve general flexibility, and ease the development of a visualization module.

This module enrichs and represents semantically structured the data collected by the data connectors (Section 4.3) using semantic models. The semantic converter will be fed with different ontologies. In this Master Thesis, we have used mainly the *SDN Network Language* but we will focusing on this in the Chapter 5.

The mapping of the unstructured data from the data sources to the ontologies information contained in the semantic converter is a dependent task from each data source. So, as in the case of the data connectors, a different "metadata tagger" will be developed in a different way depending on the data source.

Fig. 4.3 shows the structure of the semantic converter. As it can be seen, it works with the raw data from any source stored in the Data Lake. The module will be composed by this different metadata taggers. For the mapping task, this taggers will count with a series of ontologies extracted from the knowledge base models as the ones defined in chapter 3. The annotated data will be store in a RDF store, provided with a SPARQL endpoint. Semantic applications could benefit of this SPARQL endpoint retrieving structured information.

Raw data contains much non-relevant information for diagnosis and similar tasks. Semantic Converter take knowledge from ontologies and formats this data, what supposes filtering the data and reducing the noise of it. Once we have structured and filtered data, we can introduce the data in semantic modules that will be based on this ontologies, which

[6] We explain the kindness of the semantic web technologies in section 2.4

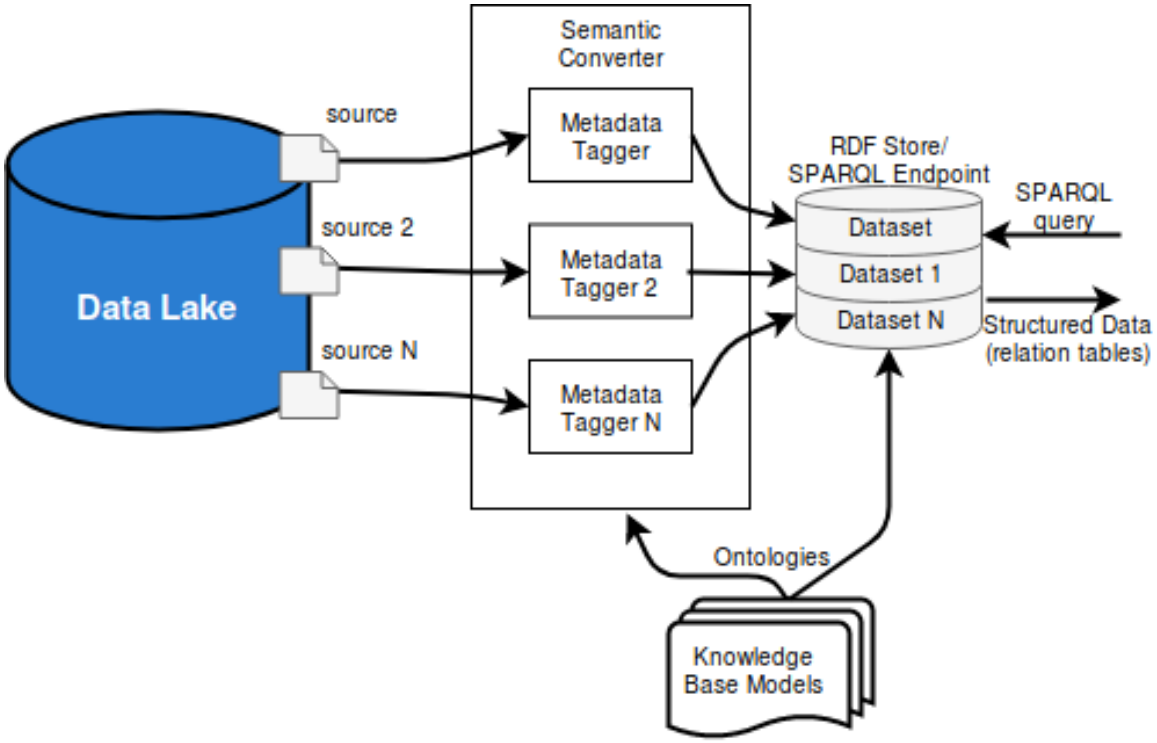


Figure 4.3: Semantic Converter structure

eases the automation of such modules. This will be explained in the following sections.

4.4.2 Symptoms Detector

An autonomous fault diagnosis system is fed with symptoms, possible evidences collected from the environment. This section describes the module that performs a detection of the symptoms, a fundamental part of a monitoring system. Therefore, independently of the data source and the type of data, collected data must be analyzed to find symptoms.

In diagnosis, the symptoms detection is the main needed monitoring task. We need to find anomalies on the environment, and then, start a diagnosis. This is a short overview of the complete B2D2 Diagnosis Model, that is described in section 3.3.1.1. This module will be an adaptation done in this master thesis for the part of *Symptoms Detection* of such model in a complex time-based environment. More concretely, we use it to detect symptoms in an SDN environment, but this will be explained in Section 5.4.3.

Figure 4.4 shows the UML diagram of the adaptation done for that symptoms detection. As it can be seen, the main class is the *Monitoring Action*, that will have one or more *Observations*. The *Monitoring Action* analyze this *Observation*. If the value of the observation/s is different from an expected value extracted from a model or historical data,

the monitoring action will have detected a symptom.

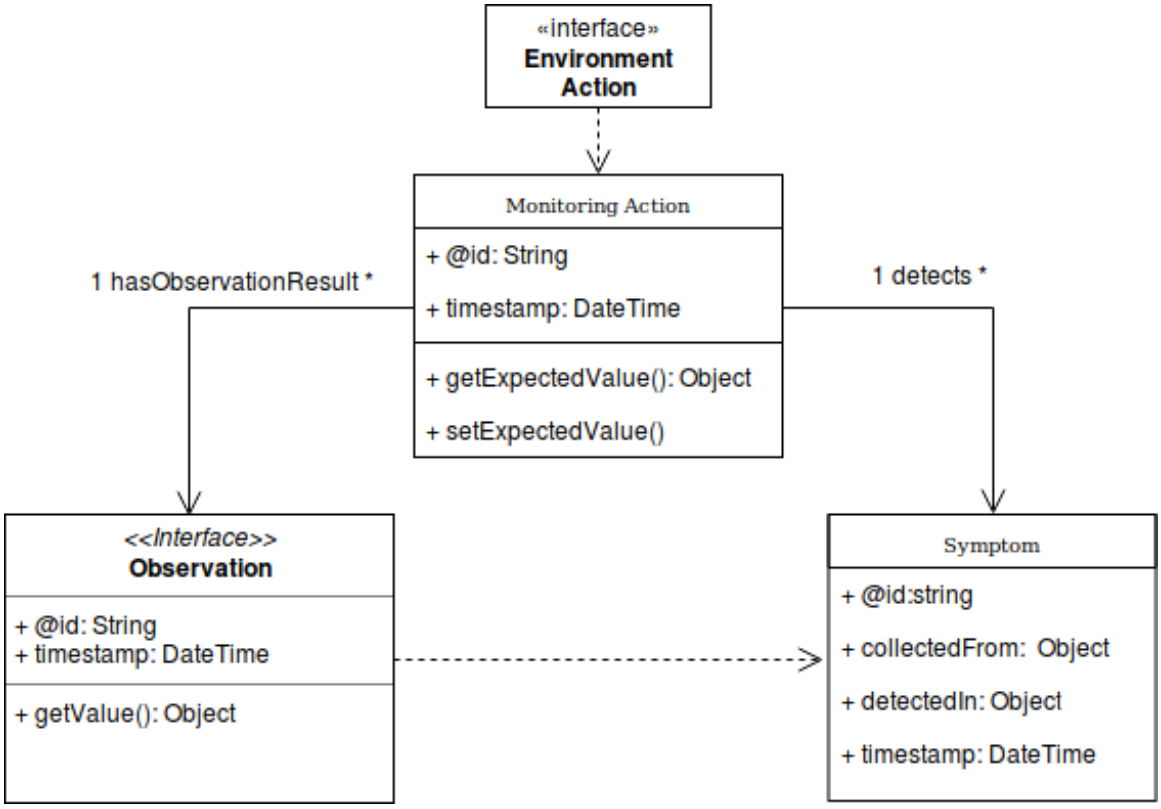


Figure 4.4: Symptoms Detection Model

Specifically, Fig. 4.5 shows in detail how the symptoms detector operates if the expected values are not extracted from models, but rather based on historical data. So, when the module starts finding symptoms, first the module try to find previous information, if not, save the observation done. If this allegation is affirmative, it launches a monitoring action. If there are unexpected changes in observations, the monitoring action will append a symptom based on this observation. The process will repeat except in the case there are not more saved observations.

Such operational mode will consist mainly in time-based analysis, and will be supported in Big Data Software Platforms, such as Elasticsearch, Hadoop or Spark. This technologies can be used to index and classify high volume of collected data in order to simplify further processing, as the pretended time-based processing. Also it is worth to stress that this processing could be done in streaming or by batches, independently of the way data is collected and stored.

To use this module, we could simply feed it with raw data from the Data Lake, but as we have explained in the previous section, this is a module based in a semantic model, so

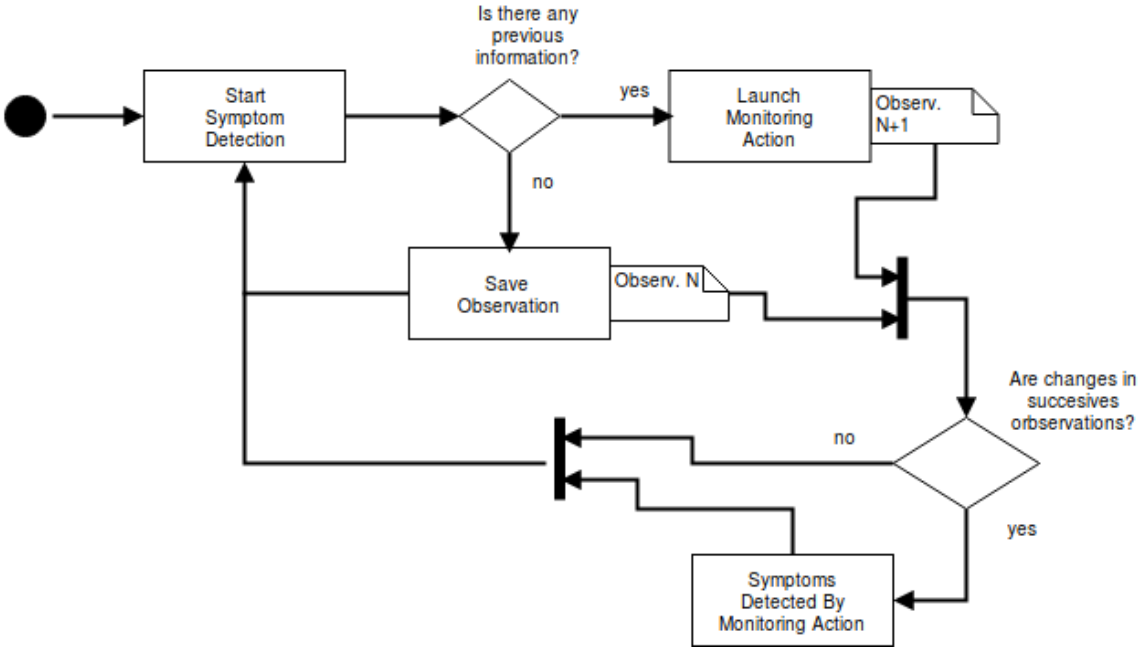


Figure 4.5: Activity diagram of the Symptoms Detector

the input of it must already be semantic triples to ease semantic inferences in it. Therefore, this module will monitor structured data by the Semantic Converter (Section 4.4.1), i.e, these data will be the Observations.

The symptoms detected will be published in a semantic format, in order to this data could be accessible to other applications, specifically, the module that will be in charge of the diagnosis process.

4.5 Diagnosis Semantic Orchestrator

In computing, orchestration is the automated arrangement, coordination, and management of computer systems. This module is in charge of coordinate the diagnosis of the system, following a Diagnosis Model, B2D2 (Section 3.3.1.1). This model follows a task model that divides the diagnosis task in three sub-tasks: *Symptom Detection* (Sec. 4.5.1), *Hypothesis Generation* (Sec. 4.5.2), *Hypothesis Discrimination* (Sec. 4.5.3)

This diagnosis semantic orchestrator generates this diagnosis reports concerning a certain element of an environment, for example, a network object in a network environment. This diagnosis reports will be expressed as RDF data, making use of the Diagnosis Model described in Section 3.3.1.1.

To determine this reports, an easy solution to implement could be generating rules

expressing cause-effect relationships to determine diagnosis depending on the symptoms detected in the data processing module. However, it is known that this kind of solutions are incapable of dealing with situations uncovered by those rules.

We make use of a better solution using causal models based on Bayesian reasoning. This kind of causal models are interesting for an uncertain environment. They make a heuristic model that relates symptoms and cause of fault, obtaining the probability of a failure. Figure 4.6 shows shows how the causal model based on Bayesian reasoning would behave.

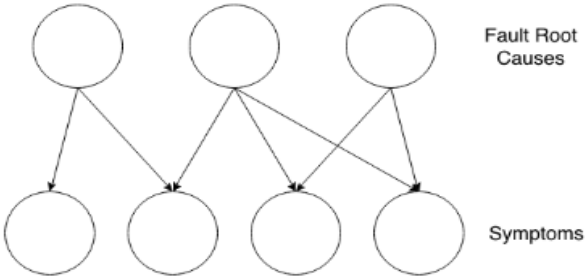


Figure 4.6: Model of Bayesian Network [11]

Thus, we apply Bayesian networks as causal model, once we had detected symptoms from the environment in the data processing module and established a set of possible faults to be predicted by the Bayesian model. To do that reasoning we use a stand-alone reasoning module through an API that has not been developed in this Master Thesis. Even so, this technology has been developed in the context of the Bayesian SDN project [4].

So this module follows the three phases mentioned at the beginning of this section. Figure 4.7 shows the general operation of this module based on this phases. Note that the reasoning module is used in both hypothesis phases.

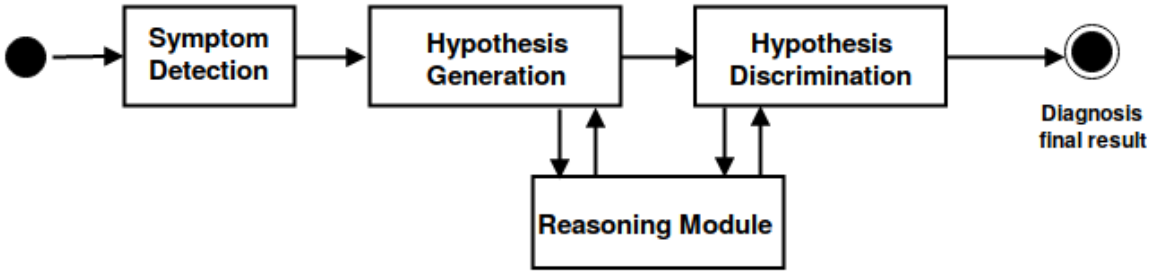


Figure 4.7: Diagnosis Phases

4.5.1 Symptom Detection Phase

First, as previously mentioned, the orchestrator waits for a symptom to be detected by the data processing module. This occurs when a *Monitoring Action* has an object property *b2d2-diag:detects* that relates it to objects of the class *Symptom*, as explained in Section 4.4.2. This event will start a *Diagnosis*[7] in the Hypothesis Generation Phase. The orchestrator will have communication with the data processing module to known if a symptom has been detected in the environment. Figure 4.8 shows how this phase works.

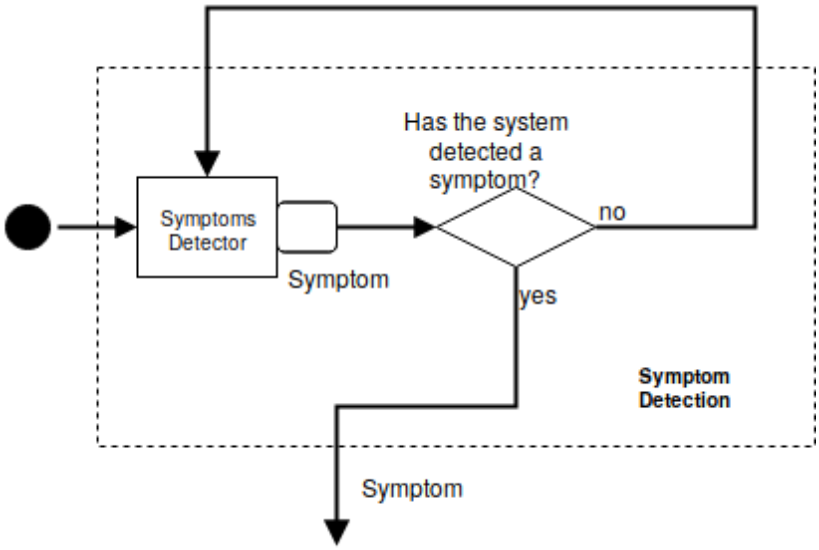


Figure 4.8: Symptom Detection Phase

4.5.2 Hypothesis Generation Phase

With a *Diagnosis* started by a symptom detected in the previous phase, begins the *Hypothesis Generation* phase. To achieve hypothesis for the current state of the system, the orchestrator queries the reasoning module giving the symptom extracted that has started the *Diagnosis* process. The reasoning module retrieves a state probability for each possible error. This process can be observer in Fig. 4.9.

4.5.3 Hypothesis Discrimination Phase

Once we have set a first hypothesis, we need to retrieve more information to achieve a conclusion. To do this, we get all symptoms detected in the data processing network from

[7]Diagnosis class is the center of the diagnosis model defined in b2d2 diagnosis model, section 3.3.1.1

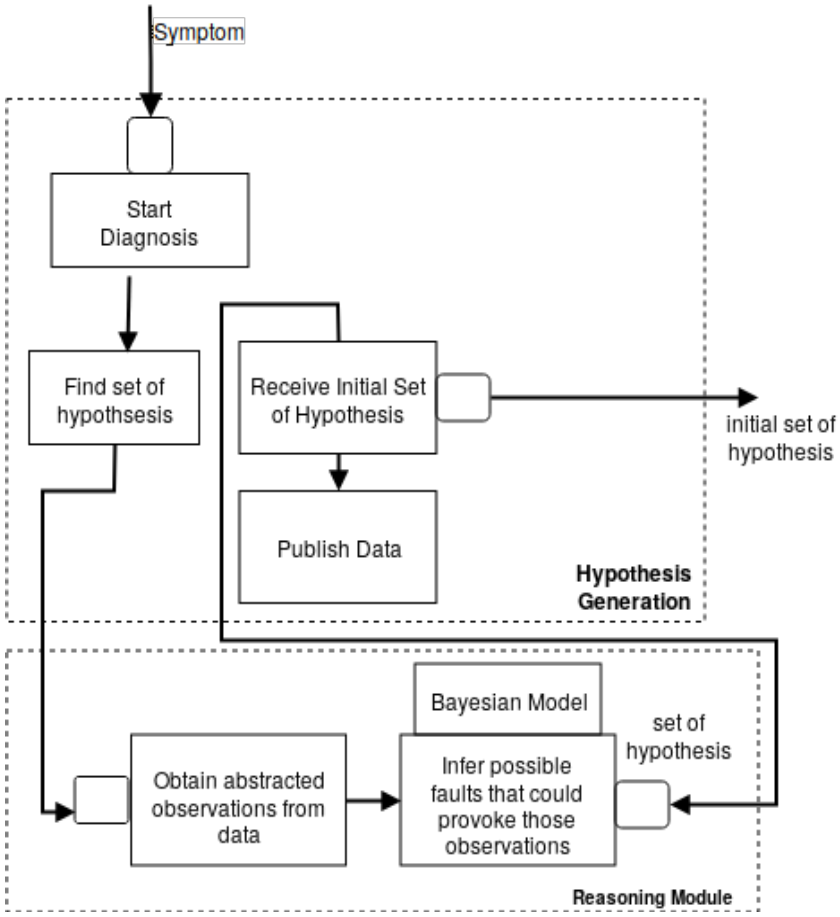


Figure 4.9: Hypothesis Generation Phase

the regarding node and the topology in a closer time window. The diagnosis steps to the *Hypothesis Discrimination* phase. In this phase, we take the most probable error predicted by the reasoning module, this will be the *Conclusion* and the *Diagnosis* will be finished. This process is shown in Fig. 4.10.

In the next section we introduce the visualization module developed, that among its multiple purposes, one of them is to show the diagnosis results.

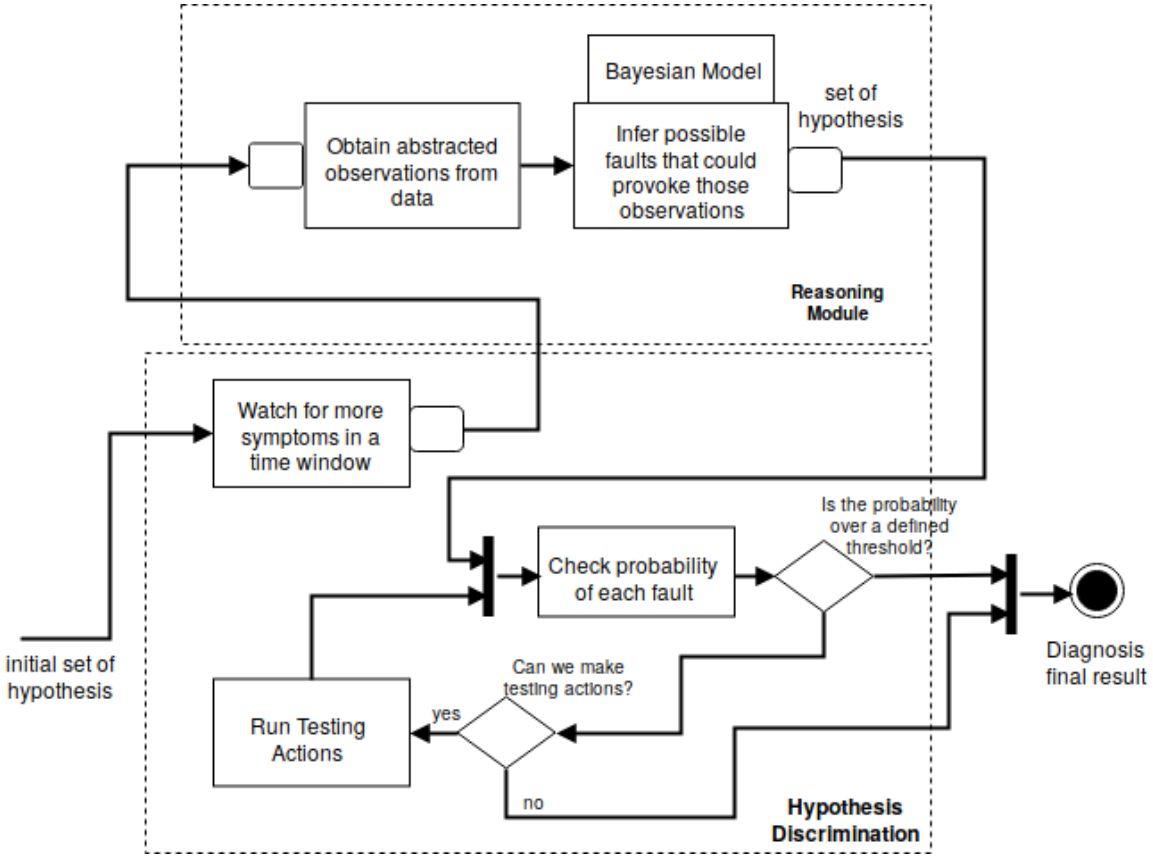


Figure 4.10: Hypothesis Discrimination Phase

4.6 Semantic Visualization Module

In this section we introduce the visualization module designed for this Master Thesis. The objective of this module is to provide a network operator with summarized and easy to handle information about the fault diagnosis management and collected information from the monitoring and processing modules. Figure 4.11 shows the architecture of this system, that is the Sefarad architecture, introduced in Section 2.5.2. We have adapted it for this project. As it can be seen, it retrieves data from database systems such as Apache Fuseki Jena or Elasticsearch (Sections 2.4.1 and 2.3.1 respectively). Furthermore, the module has the necessary flexibility to add more if necessary.

Several data sources could be connected to this visualization module. These data sources will provide it with batches of data in RDF format. The main data to be shown will be the one provided by the Diagnosis Semantic Orchestrator, but also the processed data from the data processing module, or the one directly collected from the environment by the data connector. Therefore, the visualization module, as it can be seen, only need to connect with

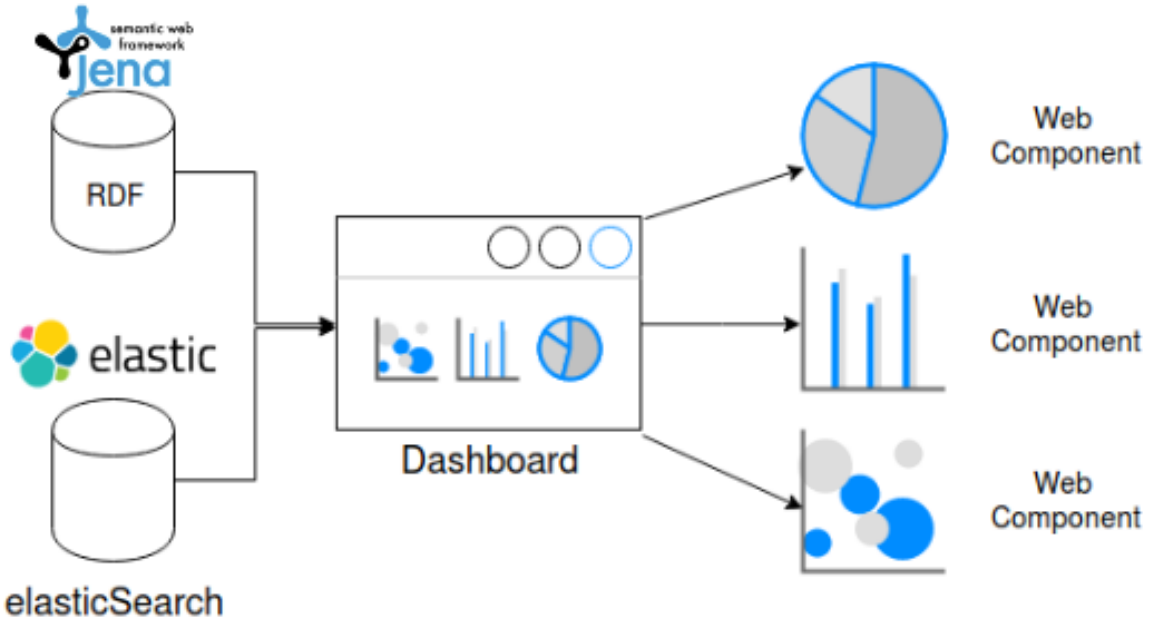


Figure 4.11: Visualization Module Architecture [24]

the storing system that are showed in the Figure 4.11, where all that information will be stored. Also several web components can be used for the dashboard. In the Chapter 5 we will do a complete review of the ones selected to improve the user experience.

Figure 4.12 shows a summary of the operation of the entire system developed. The aim of the diagram is to show how finally the visualization module gets the data for the dashboard.

As it can be seen, when the module starts collecting data, it queries the data source/s and store this data in the data lake. It will be converted and stored again in the data lake and in a RDF store with a SPARQL endpoint. The symptom detector will process this data in order to detect symptoms. When a symptom is detected, a diagnosis starts and the diagnose report and the symptoms are stored in the data lake. Until this point, the visualization module could be retrieving data directly from the data lake or using SPARQL to query the endpoint configured.

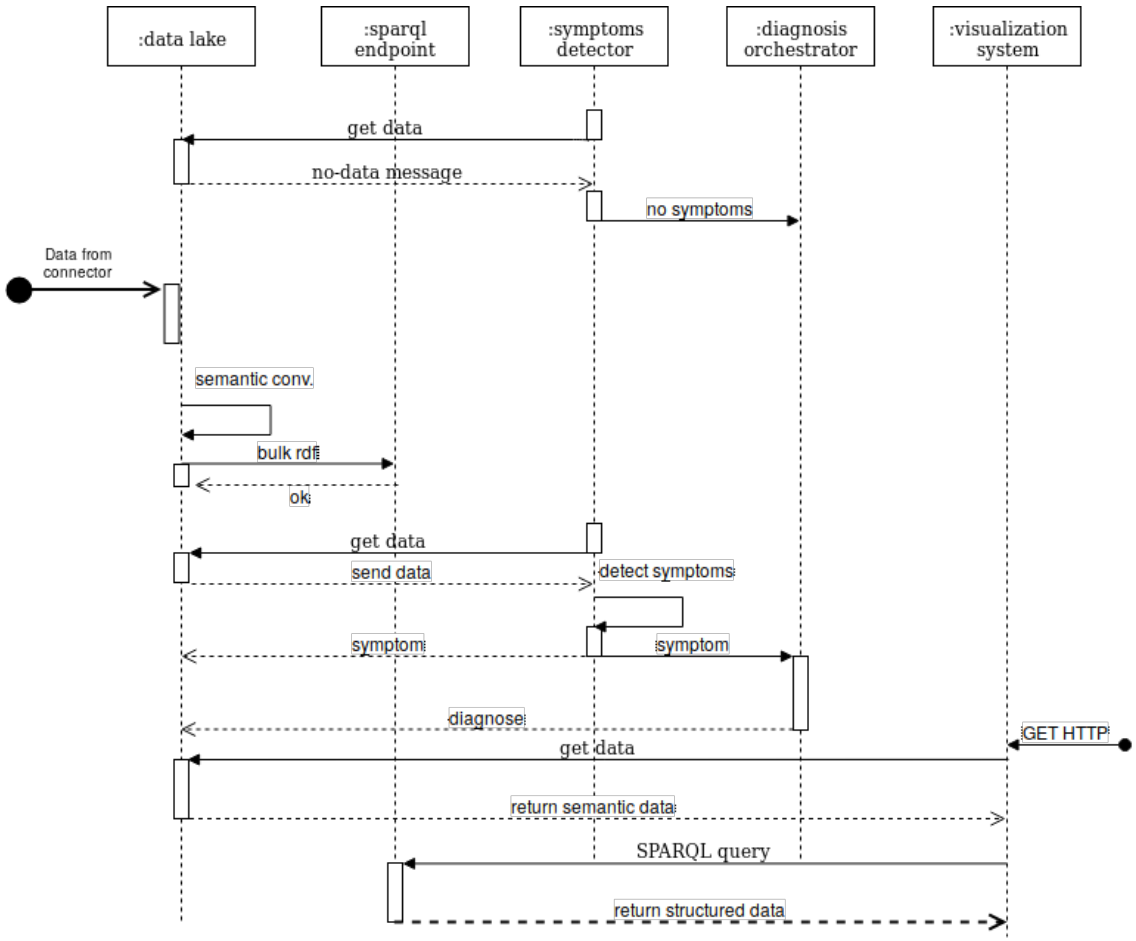


Figure 4.12: Sequence of how visualization module retrieves data

CHAPTER 5

Case study

A SDN controller, as Opendaylight, transforms completely the classic network environments. Giving the whole control role to Opendaylight involves an efficient monitoring and facilitates the diagnosis task for the system developed in this master thesis. This will be the case study developed in this work.

In this chapter we are going to describe this selected use case. The architecture explained in Chapter 4 is put into operation with a simulated network. All modules described make its function in the prototype developed and works as an automatic fault diagnosis system.

The process for the task performed for each module, as well as the information treatment in each step of the pipeline of the prototype is presented in this chapter. All the implementations for a concrete case where a Opendaylight platform is controlling that simulated network are detailed. How we monitor this network environment, what faults we will be looking for, what specifically errors defined semantically we will diagnose and how we store, publish and visualize all this data.

5.1 Introduction

The case study carried out in this master thesis is detailed in this chapter. The prototype based on semantic approaches described in Chapter 4 is implemented in a network environment with a simulated network with Mininet framework that will be controlled by Opendaylight. This is the initial premise where our system will be performing.

The way it will do with a concrete and specific data, extracted from that environment, as well as the implementation details for this concrete case are fully explained. As a result of this case of study certain amount of data flows with knowledge about the network environment is achieved. This information will be put together in a visualization system for its use by a network operator, that will be the unique actor of the case study.

Figure 5.1 shows the prototype developed for this case study. As a brief abstract, the prototype works monitoring the network through a connector, colored blue in Figure 5.1. This data is introduced in the Data Lake, implemented with an Elasticsearch node cluster, where will be the raw data from the network. This data is converted to a semantic format an stored in a RDF tuple store [1] and Elasticsearch again [2]. Finally, a *Semantic Orchestrator* use this data and the Reasoning API Module to represent in a semantic format live diagnoses from the network. These and information from the it will be shown in a visualization system with useful information for network operators. This simplified overview is exposed to facilitate the understanding of the prototype. Nevertheless, explanations for each task done in the prototype are included in the following sections.

This chapter is divided in sections as follows. First section, Section 5.2, describes the network environment used to test the prototype developed in this master thesis for fault management. Section 5.3 relates how the data ingestion from Opendaylight controller is done. Section 5.4 on the other hand describes the process done for transform raw data to useless data. Last, Section 5.5 will explain us how the data after the previous process can be diagnosed. Finally, Section 5.6 will illustrate how we show all this data in a complete visualization system.

[1] Apache Jena Fuseki

[2] Please, refer to 2.3

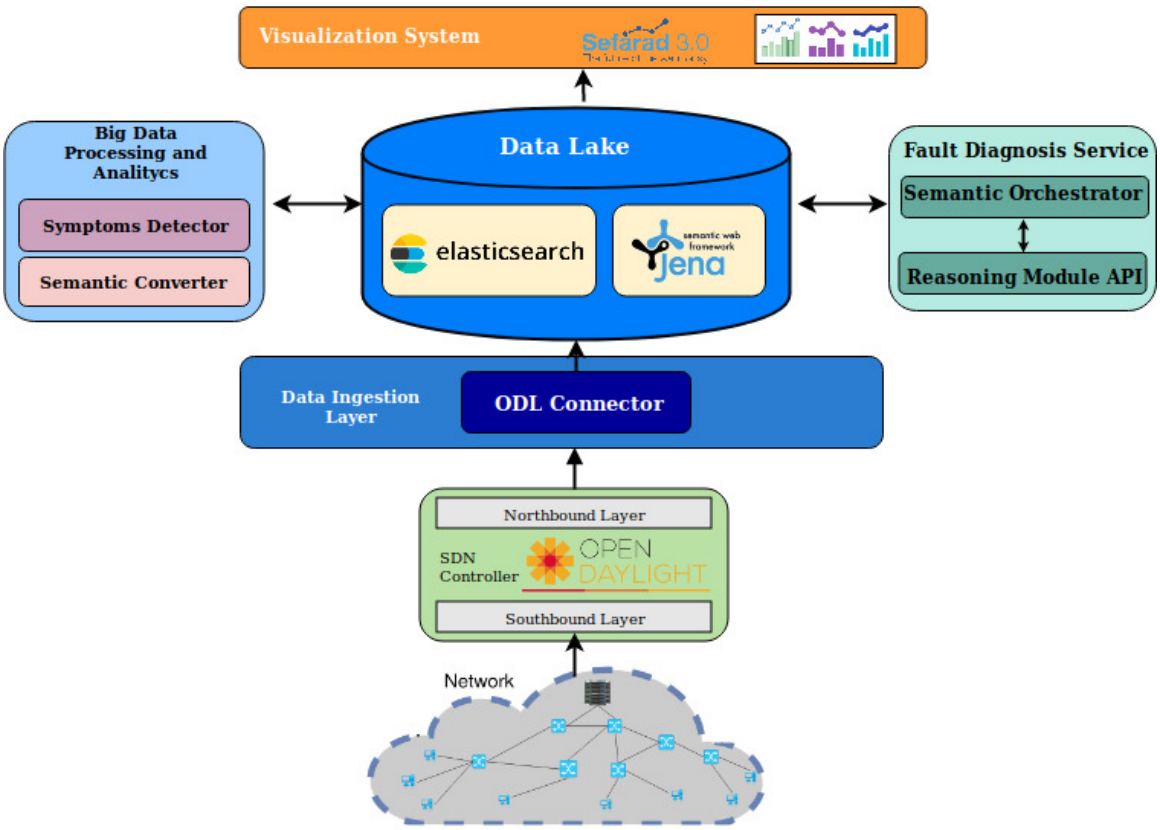


Figure 5.1: Prototype Architecture

5.2 Network Environment

The network environment is not mentioned in Chapter 4, but it is worth detailing it since it is necessary to develop and test the fault diagnosis prototype with different networks, data and faults.

This section is divided in two. The section that follows talks about the network simulator framework and Section 5.2.2 the SDN controller connected to this simulator.

5.2.1 Network Simulation

To test the prototype described in the previous chapter[3] that carries out a monitoring and diagnosis of the network, we need a network to monitor. We have selected the network virtualization tool Mininet [15]. With Mininet we obtain synthetic data that will feed our prototype. Mininet provides a virtual test bed and development environment for software-

[3] Architecture is detailed in Chapter 4.

defined networks (SDN).

This SDN simulation environment supports OpenvSwitch[4] switchs and external controllers. An OpenvSwitch switch is a virtual switch that implements the OpenFlow protocol among others. The external controller to which we will connect this simulated network is Opendaylight[5].

To diagnose faults in this simulated network, we need the network to fail. To generate synthetic faulty data, some failures will be purposely injected and fixed in the network periodically. Those faults will be executed periodically either through the SDN controller or directly at network level through Mininet API.

5.2.2 SDN controller: Opendaylight

The network simulation with Mininet framework allows using external controllers. For this, we use a complete software controller called Opendaylight. Opendaylight is based on a Model-Driven Service Abstraction Layer (MD-SAL) that describes network devices and applications as objects. The idea is that the interactions with these objects (or models) are processed within the MD-SAL. Opendaylight communicates with the network through "south-bound" interfaces anchored to the MD-SAL. Thereby, inside the controller platform, multiple service functions are defined, some of them can be useful to oversee the network status.

Some of this services are installed in Opendaylight to give basic forwarding functions to the network, as well as basic monitoring functions. Specifically, Topology Manager, Switch Manager and Host Tracker functions will allow us to monitor the network and collect data. Access to the data granted by these functions is provided by the Opendaylight controller through its "Northbound" API. This API allows not only obtaining data, but also make changes in the network to update configuration and policies if required, in our case, we only use it for retrieving data.

In Figure 5.2 we can show how Opendaylight is already managing a network simulated and through the topology manager shows the topology of the network.

With this two elements working, the network environment set up is complete. The system is prepared to be monitored and diagnosed. In the following sections, the performance of the prototype in a particular case is tested.

[4]The Official Site in the Linux Foundation for OpenvSwitch can be found here: "https://www.openvswitch.org/".

[5]Widely viewed in Section 2.2.2.

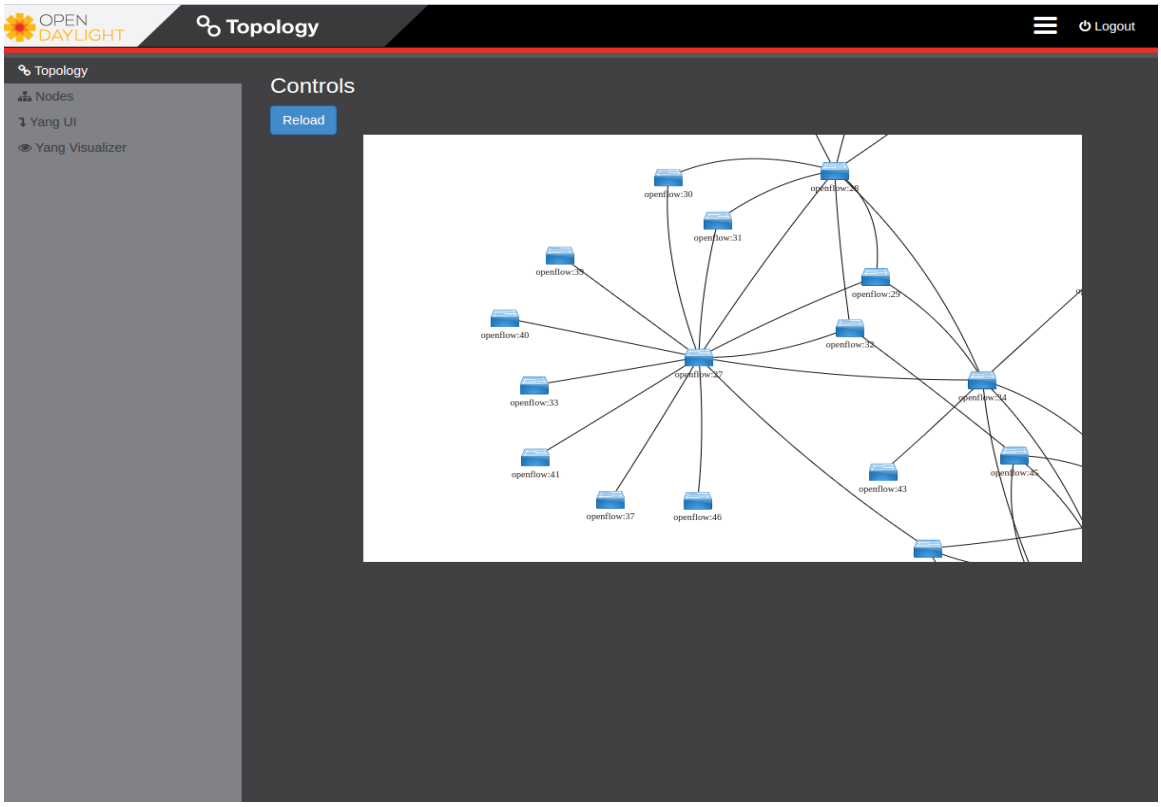


Figure 5.2: Opendaylight Topology UI

5.3 Data Ingestion

In this section the data ingestion performed with a module specifically developed is detailed. The methodology followed to do this and the problems that this process entails are explained.

This sections is organized as follows. Section 5.3.1 give and explanation of what are the data sources to be monitored. Section 5.3.2 explains how to collect data from this data sources and the solution done for that purpose.

5.3.1 Data Sources

The selection of the data sources is one of the most important steps of the designing process. It is important for the correct functioning of the system that we collect data which holds causal relationships between its values. We will be collecting data from certain modules within Opendaylight, selecting the ones that would show changes about the status of the network, giving stats, current status of specific nodes, or giving information that can be very useful to detect when certain faults happen within the network.

Therefore, we choose data from the “network-topology” and “opendaylight-inventory” data models of the Opendaylight network controller, since these contains the necessary information to get a snapshot of the network status.

The network-topology data model from Opendaylight includes information on the elements that are part the network its interconnections and some basic informations on their physical status.

The opendaylight-inventory model holds information on the configuration of each network node, as well as statistics of its components,. Specifically, all the information related to the configuration of node flow tables and its statistics, the information of the interfaces configured in that node and its statistics and more.

While many other data models and sources could be used in order to provide more information in the searching of causal relationships, the data obtained from those sources would not add too much extra information, and could actually generate noise so we decide to only poll this two models from Opendaylight.

5.3.2 Data Collection and Storing

Once we have selected the data sources, we proceed to the ingestion of the data extracted from them. Data ingestion is a function performed by a data connector. As all the data sources comes from Opendaylight, we have called this the "ODL connector"(ODL from Opendaylight). The ODL connector will be sending the data from the data sources described in the previous section to an indexing engine provided by ElasticSearch. Elasticsearch is a document-driven engine[6] where this raw data extracted from the network controller will be stored.

When simulation starts, the ODL connector starts sending these two essential data requests about topology and nodes status reports from Opendaylight by sending HTTP queries to its operational REST API. Opendaylight replies by sending back data models in JSON format. These requests are made periodically. Figure 5.3 will show this pipeline that the ODL connector follows for the mining of information from the Opendaylight controller.The pipeline illustrates how this mentioned process works, in such a way that the data connector polls the controller through a Python-based client for bulking the data requested to Elasticsearch engine.

This data collection is possible thanks to the "northbound" REST API that Opendaylight provides. Through this API, we obtain access to the data structures that have been

[6]For more information, please refer to 2.3 for more information

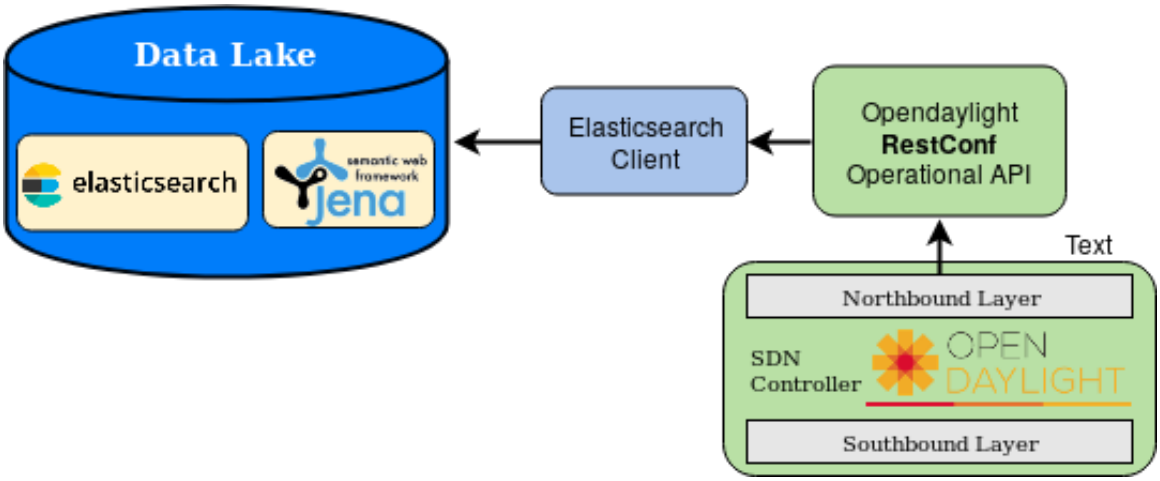


Figure 5.3: Opendaylight Connector structure

defined in the controller by Yet Another Next Generation (YANG) models [8], 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. On the other hand, the "config" database contains data that we want to push into the network. Thus we are focusing in the operational database, that retrieves the information about network-topology and opendaylight-inventory models.

To illustrates this, in listings below (5.1 and 5.2), the necessary queries to retrieve information from Opendaylight controller is shown. First of them, gives information about the nodes through the *Inventory Manager*, for example, the state of its flow tables or its interfaces. Query to the *Topology Manager* gives information about the current of the network topology.

Listing 5.1: Query to the *Inventory Manager*

```
http://{Opendaylight Endpoint}/restconf/operational/opendaylight-inventory:nodes
```

Listing 5.2: Query to the *Topology Manager*

```
http://{Opendaylight Endpoint}/restconf/operational/network-topology:network-topology
```

So we query periodically Opendaylight like the way showed in Listing 5.1, and then, we store this data in the Data Lake. The Data Lake implementation is done with Elasticsearch. We have chosen it due to its flexibility, the variety of integration tools that eases the process

of ingesting data and the easy to use web API[7], that allows retrieve and ingest data over HTTP.

Listing 5.3 shows the summary structure of the raw data collected from Opendaylight. There is information about the tables of a node and its flows and its interfaces (node-connectors).

Listing 5.3: Example of Opendaylight Inventory Raw Data

```
{
    "id": "openflow37",
    "flow-node-inventory:table": [
        "id": 0,
        "flow": [
          {
            "priority": 0,
            "id": "L2-switch1",
            "table_id": 0,
            "opendaylight-flow-statistics:flow-statistics": {
              "byte-count": 0,
              "packet-count": 0
            },
            "idle-timeout": 0,
            "cookie_mask": 0,
            "hard-timeout": 0
          }, ... {}],
            "opendaylight-flow-table-statistics:flow-table-statistics": {
              "packets-looked-up": 5,
              "active-flows": 0,
              "packets-matched": 0
            } }]
    "node-connector"[{
        "id": "openflow:37:1",
        "opendaylight-port-statistics:flow-capable-node-connector-statistics": {...},
        "flow-node-inventory:state": {
          "live": false,
          "link-down": false,
          "blocked": false
        }
      }
    ],
```

We will be indexing the data from the two data sources into two types of elements, node documents, that will store information from the opendaylight-inventory and topology documents, that will hold the network-topology information. Once we have defined the mapping of the data, we run Elasticsearch. While Elasticsearch is running, we can push

[7]More information is presented in Section 2.3

data entries which are stored. We store one data entry information from the network-topology and a number of data entries from the inventory-manager equal to the N-nodes that compose the network. For this second bunch of queries, we use the bulk API of Elasticsearch[8].

In addition, we add a timestamp mark for each of the messages. It is worth stressing that, although these two messages are in different index of Elasticsearch, they are correlated by this timestamp field, because the two queries to the two modules of Elasticsearch will be done at the same time.

5.4 Data Processing

Raw data by themselves do not provide all the information we want to get from the network. In this section, we explain the processes performed to achieve a really useful system that could be easy to use for a network operator. To do this, we will perform a first preprocessing of the raw data, reduce noisy data; then, we match this preprocessed data with a semantic hierarchy defined in this Master Thesis for SDN data. Last, we offers this data to a symptom detector module who find changes or missing values or unexpected values in these hierarchical data. In order to do all this processing tasks, we implemented a Python software that can carry out these concrete actions for our use case[9].

Section 5.4.1 talks about how we are reduce noise in raw data, while Section 5.4.2 explain the enrichment of the data resulting from the preprocessing process. The detection of symptoms done is illustrated in Section 5.4.3. By last, Section 5.4.4 returns to the storing problem, in this case, for storing data in a triplets store[10].

5.4.1 Preprocessing

We have already extracted data from the data source and stored it in the Data Lake at this point. Raw data does not contribute anything in particular to diagnose faults happened in the simulation. 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, so we always do a process of this data.

Data is stored in Elasticsearch as mentioned in Section 5.3.2 in a JSON format. Data

[8]The bulk API is describe in the Official Elastic site: https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-bulk.html

[9]More information is given in Chapter 4

[10]If you are here without knowing what a triplets store is, please refer to 2.4

from Opendaylight contains a lot of proprietary information or marks that are exclusively useful for Opendaylight environments. We try to unlink from an environment totally embedded in Opendaylight, so we remove this information.

On the other hand, the amount of information that opendaylight-inventory retrieves about flow tables is excessive for our kind of nodes because Openflow nodes only have a flow table and Opendaylight retrieves a long amount of empty not used flow tables. This flow tables are removed from the data.

In addition to this preprocessing tasks, we make a time-based analysis in some statistics values that gives total information (bytes received). When processing this kind of metrics giving it a temporal dimension, we obtain better information about this statistics.

Definitively, this preprocess task is the previous step for matching data with the semantic model create, the SDN Description Language, introduced in Section 3.2.2 of this Master Thesis.

5.4.2 Data Enrichment

One of the objectives of this project is representing network data in a visualization system that allows a network operator to manage the network with human-readable data, other, get and application that gives interoperability to work with other web applications. Raw data does not match the requirements of this objectives, so we need to get structured, hierarchic and linked data.

SDN is a recently booming technology, so there are not RDF models to describe its singularities. For that reason, we have extended RDF models created previously for network descriptions. In section 3.2.2, this extension is presented. This structural model can be used to represent data from a SDN controller.

Depending on the data source, we need to follow a different approach for each case. On one hand, we get periodically data from the *Inventory Manager* of Opendaylight. For this data we need to introduce terms from the control plane as showed in Figure 3.5. On the other hand, the *Topology Manager* gives us data about the current state of the topology. In section 3.2.1.3 is mentioned that *Network Markup Language* its focused in model the data plane. Well, topology data can be modelled only using properties from NML.

To illustrate this conversion easily, Figure 5.4 shows the hierarchy used for the status of nodes retrieved from Opendaylight through the opendaylight-inventory module. The documents from this module are modeled as *OpenflowNode* RDF instances, each of them identified with a unique URI. A *OpenflowNode* instance has *Flow Tables* and *Interfaces*.

The *Flow Tables* in turn contains the *Flows* entries information.

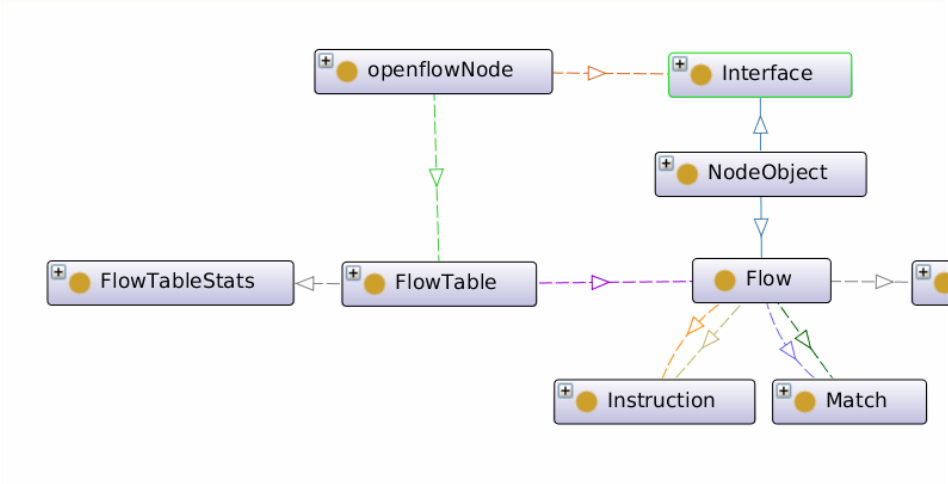


Figure 5.4: SDL Hierarchy used for inventory nodes

Listing 5.4 shows the processed data output from 5.3 after the data enrichment.

Listing 5.4: Example of Annotated Inventory Data

```
{   "@context" : {...},
    "@id":"",
    "@type":"OpenflowNode",
    "hasSnapshot":{},
    "hasTable":[{ "@id":"table0",
                  "@type":"Table",
                  "hasFlow":[{"@id":"L2switch-1","@type":"Flow",
                  "hardTimeout":0,
                  "iddleTimeout":0,
                  "hasNetworkStats":{"@id":""L2switch-1#flowStats",
                                     "@type":"FlowStats",
                                     "byteCount":0,
                                     "packetCount":0},
                  "priority":0,
                  "hasMatch":[...],
                  "hasInstruction":[...]
                }
    "ndl:hasInterface":[{"@id":"openflow37_2",
                         "@type":"Interface",
                         "hasState":{},
                         "hasNetworkStats":{},
                         "stpStatus":"discarding"}]
    }
```

On the other hand, the *Flows* has a lot of interesting data values, some of them showed in this figure, but there are more, that we explained in the model extension done in Chapter

3. That properties will be very useful in the symptoms detection task, since errors in the flow tables can lead to many network faults.

The data extracted from the network-topology module is modeled as shown in Figure 5.5. As it can be seen, we have the concept of *Controller* (Opendaylight in our case) that has *Topology*. This topology is composed by *Nodes* (OpenflowNodes) and *Links*[11]

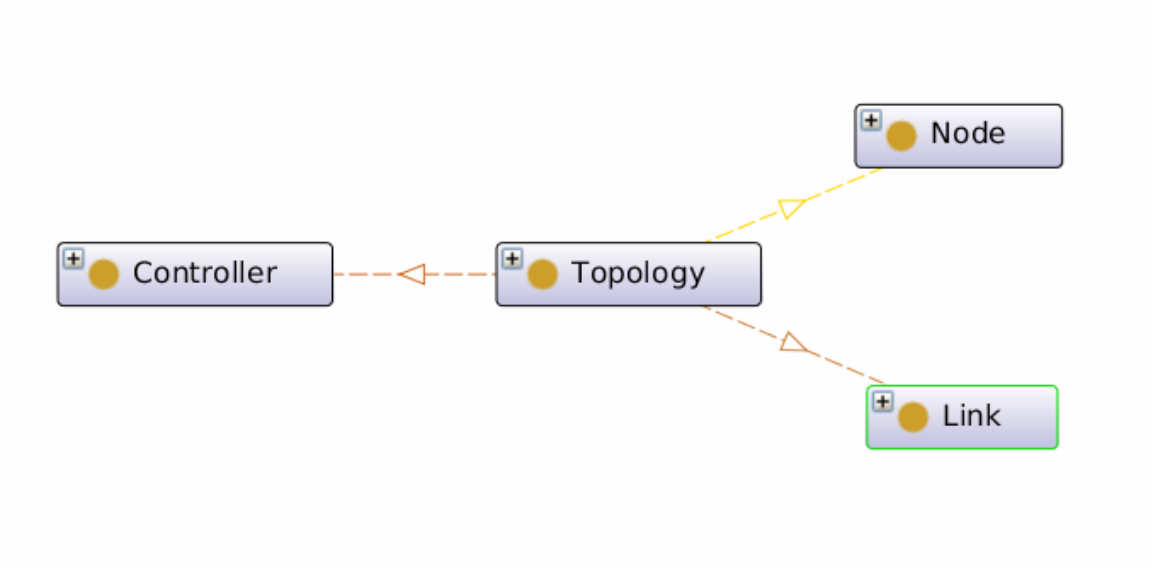


Figure 5.5: SDL Hierarchy used for topology data

Listing 5.5 and Listing 5.6 describes an example of a processed topology message modeled with SDL. For comparison, we give RDF data serialized as JSON-LD[12].

[11].

[12]A brief view of json-ld format: https://json-ld.org/

Listing 5.5: Opendaylight Topology Data

```
{ "topology-id": "flow:1",
  "node": [{"node-id": "openflow:33", "
      termination-point":
      [...]},...{...}]
  "link": [{
  "link-id": "openflow:2:9",
  "destination": {
    "dest-node": "openflow:9",
    "dest-tp": "openflow:9:2"
  },
  "source": {
    "source-node": "openflow:2",
    "source-tp": "openflow:2:9"
  }
 }]
}
```

Listing 5.6: Annotated Topology data

```
{"@context" : {...},
"@id":"1518011749",
"@type":"nml:Topology",
"nml:hasNode":  [ {
        "@id": "openflow33",
        "@type": "OpenflowNode",
        "ndl:hasInterface": [...],
        "ndl:connectedTo": [...] } ...],
"nml:hasLink":  [{
        "@id": "openflow10_2",
        "@type": "nml:Link",
        "nml:isSource": {
          "@type": "Interface",
          "@id": "openflow10_2",
        }, ... {...}],
"hasSnapshot":{...}}
```

The capacity to have the data under RDF standard comes by the relation between two kind of documents. An instance from the snapshot of a node in a certain temporary moment always points to the definition of that node in the topology document. This cross referencing boosts the capacity of the prototype (Faults detected in topology messages could be easily related to inventory messages for example).

5.4.3 Symptoms Detection

At this point, we have RDF instances from the Openflow switches status as well as for the network topology status. In this section, the symptoms detector described in the architecture section is used for its purpose processing data from data sources selected for this case study.

So the symptoms detector detects anomalies in this data performing 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 analyze the data looking for unexpected changes in the values of some variables. Section 3.3.2 describes the extension done in B2D2 diagnosis model that we will apply to this case study, some of the *Symptoms* of the model will be which we will finding in this time-based analysis.

Symptoms detector works with enriched data. For each entry in the Data Lake (Elasticsearch) that have been formatted with the SDL model by the Semantic Converter, this module generates a report following B2D2 model, a *Monitoring Action* that realize Obser-

vations into the Data Lake. If there is a temporal anomaly (or anomalies), will append a detected *Symptom*. Figure 5.6 shows how we implemented this process based on the architecture modules.

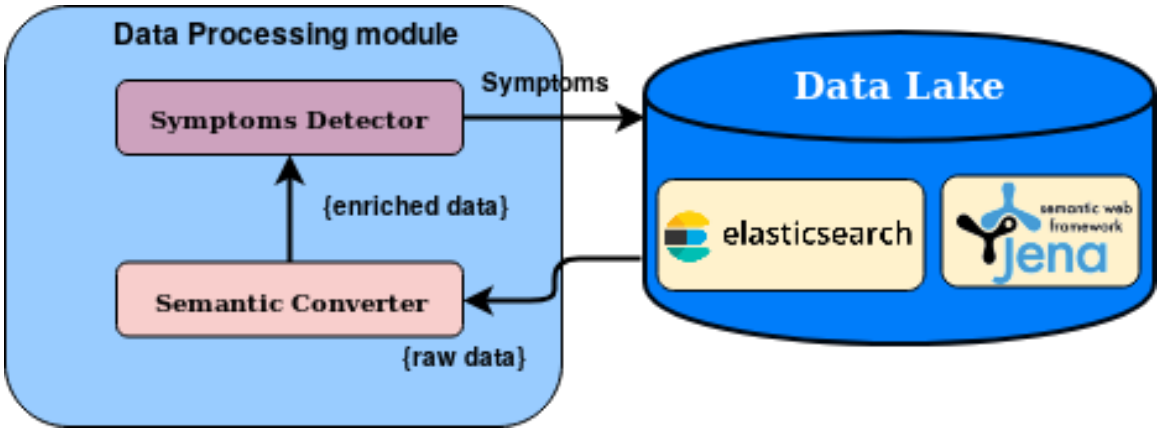


Figure 5.6: Symptoms detector structure

To detect symptoms this module checks every inventory message of each node of the monitored network and look for unexpected changes. It looks at flow tables of each node. The flows configured within must not change in an unexpected way. For this case of study, the symptoms looked are the followings. As mentioned, we can find this symptoms in semantic model for SDN diagnosis among others (Figure 3.7). It is worth to stress that the model is composed by some symptoms, some of them for this case of study, but the idea when creating the model is extending it with more symptoms from SDN and Openflow infrastructures.

- **FlowHardTimeoutModified -** A Hard Timeout is added to a flow.
- **FlowIddleTimeoutModified -** A Iddle Timeout is added to a flow.
- **FlowInPortModified -**An in port match is modified within a flow.
- **FlowOutputActionModified -** An output action is modified within a flow.
- **ModifiedHosts -** Hosts modified in topology.
- **FlowPriorityModified -** Flow action priority modified.
- **DroppingLLDPPackets -** A LLDP packet has been dropped in a node.
- **TableHasNoFlows -** A flow tables does not have flows.

In Listing 5.7 a turtle[13] serialized example of the output of this module is given. As

[13]The W3C recommendation for turtle format: https://www.w3.org/TR/turtle/

it can be seen, it follows *b2d2-diag* knowledge model to describe the *Monitoring Actions* carried out[14].

Listing 5.7: A Monitoring Action Example

```
@prefix b2d2-diag: <http://www.gsi.dit.upm.es/ontologies/b2d2/diagnosis/> .
@prefix sdn-b2d2: <http://www.gsi.dit.upm.es/ontologies/sdn-b2d2/diagnosis/> .
@prefix topology: <http://bayesiansdn/simulation2018271454_0/topology/> .
@prefix sdn-ndl: <http://www.gsi.dit.upm.es/ontologies/sdn-ndl/> .

<http://bayesiansdn/simulation2018271454_0/monitoring/openflow2/1518011846>
  a <http://www.gsi.dit.upm.es/ontologies/b2d2/diagnosis/MonitoringAction> ;
  b2d2-diag:detects <http://bayesiansdn/simulation2018271454_0/monitoring/openflow2
      /1518011846#L2switch-38> ;
  b2d2-diag-diag:whenHasBeenPerformed 1518011846 .

<http://bayesiansdn/simulation2018271454_0/monitoring/openflow2/1518011846#L2switch-38>
  a <http://www.gsi.dit.upm.es/ontologies/sdn-b2d2/diagnosis/FlowInPortModified> ;
  b2d2-diag:collectedFrom <topology:openflow2> ;
  sdn-b2d2:detectedIn <http://bayesiansdn/simulation2018271454_0/monitoring/openflow2/
      L2switch-38> .

<topology:openflow2> a <http://www.gsi.dit.upm.es/ontologies/sdn-ndlOpenflowNode> .
<http://bayesiansdn/simulation2018271454_0/monitoring/openflow2/L2switch-38> a <http://
    bayesiansdn/simulation2018271454_0/monitoring/openflow2/Flow> .
```

This monitoring actions will be also indexed in Elasticsearch, as a new kind of document, "monitoring actions". This data will be the feed for the Semantic Orchestrator (Section 4.5). Also mention that a Python-based implementing classes of the diagnosis model to carry out the symptom detection of this module can be found in Appendix A.

5.4.4 Storing Semantic Triplets

As mentioned in previous subsections of this section, the processed data in the data processing steps are stored in Elasticsearch, but we also need a data storing system for semantic data represented in RDF. We select Apache Jena Fuseki(or simply Fuseki) [22] for storing our semantic triplets. Fuseki serves RDF data over HTTP, moreover, allows to expose our triples as a SPARQL end-point accessible over HTTP. Fuseki provides REST-style interaction with our RDF data.

Fuseki allows to make requests about the data stored in it, i.e, works as a SPARQL

[14]Monitoring Actions are defined as a subclass of action that is executed during a diagnosis in the model. Refers to Figure 3.6 for more information

Server [15]. Figure 5.7 shows this idea and places this sub-module and its functionality inside the data processing module.

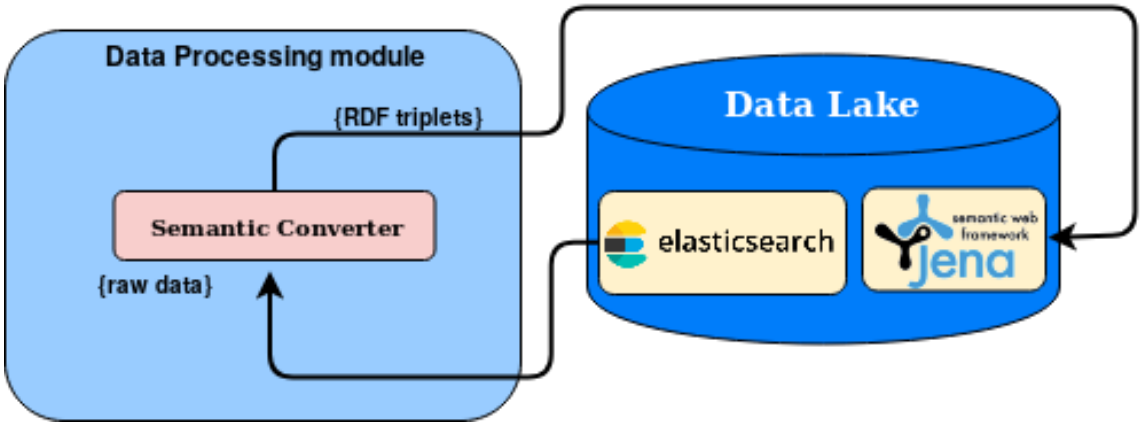


Figure 5.7: Storing Semantic Triplets Pipeline Example

To do that, we ingest the processed and converted RDF data to TDB [16]. TBD is a component of Jena for RDF storage and query. Just like data ingestion in Elasticsearch, data ingestion for Fuseki is also easy, more if possible in this case, because we introduce all semantic triples as a TBD dataset, there is no need of indexing or similar techniques, it is automatic since data is in a semantic format. Hence, at this point, we can retrieve information from the dataset created (an RDF graph) through SPARQL queries. The visualization system make use of this feature to get information quickly from the Data Lake.

5.5 Diagnosing data

Now that the data is processed and stored, the diagnosis process is described. On one hand, We have chosen to carry out an approach of following a semantic diagnosis model, described in B2D2. On the other hand, the reasoning process needed for achieve a diagnosis conclusion is carried out by a external Bayesian reasoning API.

For this, this section is divided in two main parts. Section 5.5.1 will tell information about the coordination done with the semantic orchestrator described in Section 4.5. Section 5.5.2 explains how we will use this external API for reasoning briefly. Also Section 5.5.2 will conclude how this diagnosis reports are published.

[15] Sparql is described in section 2.4.2
[16] TBD documentation available here: https://jena.apache.org/documentation/tdb/index.html

5.5.1 Starting Diagnoses

Let's have this assumption. For example, at a certain time a node identified as "openflow2" have presented an unexpected change in a flow from its table. A in port match is change to other port. If this port number is nonexistent or the interface of that port is down, the node presents a fault an begins to discard packets. This fact will start a diagnosis. This easy example serves as introduction for this section where we explain how the semantic orchestrator module boots diagnoses. The symptoms detected use the B2D2 diagnosis model (Section 3.3.1.1), carrying out the first necessary task for a diagnosis process. This diagnosis process will be regarding to the Task Model included in the same model, with a *Monitoring Action* to detect symptoms. If a symptom is detected, a diagnosis is started. In this case of study this fact is implemented adding a "watcher" to the symptoms detector. For each symptom detected in the network data, a new diagnosis in *Hypothesis Generation* phase starts. It is important to say that these diagnosis are independent for each switch of the network, but we will see in the next phase done of these diagnosis that other environment issues in the moment of the start of these diagnosis will be also incorporated to it. Listing 5.8 shows the diagnosis for the described phase.

Listing 5.8: Diagnosis in Hypothesis Generation Phase

```
@prefix b2d2-diag: <http://www.gsi.dit.upm.es/ontologies/b2d2/diagnosis/> .
@prefix sdn-b2d2: <http://www.gsi.dit.upm.es/ontologies/sdn-b2d2/diagnosis/> .
@prefix sdn-ndl: <http://www.gsi.dit.upm.es/ontologies/sdn-ndl/> .
@prefix topology: <http://bayesiansdn/simulation2018271454_0/topology/> .
@prefix pr-owl: <http://www.pr-owl.org/pr-owl.owl#> .

<http://bayesiansdn/simulation2018271454_0/diagnosis/openflow2/1518011943/started> a
    b2d2-diag:Diagnosis ;
    b2d2-diag:hasDiagnosisPhase [ a b2d2-diag:HypothesisGeneration ] ;
    b2d2-diag:hasHypothesis <http://bayesiansdn/simulation2018271454_0/diagnosis/
        openflow2/1518011943/started#10>

    b2d2-diag:hasPerformedAction [ a b2d2-diag:MonitoringAction ;
             b2d2-diag:detects <http://bayesiansdn/simulation2018271454_0/monitoring/
                 openflow2/1518011943#L2switch-38> ;
             b2d2-diag:whenHasBeenPerformed 1518011943 ] ;
    b2d2-diag:isStartedBySymptom <http://bayesiansdn/simulation2018271454_0/diagnosis/
        openflow2/1518011943#L2switch-38> ;
    b2d2-diag:whenHasStarted 1518011943 .

<http://bayesiansdn/simulation2018271454_0/diagnosis/openflow2/1518011943#L2switch-38> a
     sdn-b2d2:FlowInPortModified ;
    b2d2-diag:collectedFrom <topology:openflow2> ;
    sdn-b2d2:detectedIn <http://bayesiansdn/simulation2018271454_0/diagnosis/openflow2/
        L2switch-38> .
```

```
<http://bayesiansdn/simulation2018271454_0/diagnosis/openflow2/1518011943/started#10> a
    b2d2-diag:Hypothesis ;
    b2d2-diag:hasConfidence [ a pr-owl:ProbAssign ;
            pr-owl:hasStateProb 3.030299e-03 ] ;
    b2d2-diag:representsPossibleFault <http://bayesiansdn/simulation2018271454_0/
        diagnosis/openflow2/1518011943/started#FlowsPrioritiesModified> .
```

It is necessary to contemplate the special situation that occurs when several symptoms are detected at the same time in the same node. For example, if in a flow, a timeout is added and the priority for the actions changes. In this case, the system is intended to choose one of them and starts only one diagnosis for that, the other symptoms will be incorporated as additional evidences in the next phase.

5.5.2 Using external reasoning API and diagnosis final report

A python-based API to import a causal model for Bayesian reasoning is incorporated to the semantic orchestrator [4]. After starting a diagnose as explained in Section 5.5.1 we use the programmed API for Bayesian reasoning giving only the symptom that has started the initial diagnosis instance. This reasoning module will retrieve a set of hypothesis. This initial diagnosis reports is stored as a new type "diagnosis" in Elasticsearch.

When receiving this initial set of hypothesis, the diagnosis commutes to the *Hypothesis Discrimination* phase. In this phase, the processed data is examined in search of more evidences, first in the switch, and then, events from the topology. For example, a diagnosed has been started for "openflow2", the symptom is introduced in the reasoning module and the semantic orchestrator process the response and release the initial report with the set of hypothesis. Then, the module collect more symptoms from the node and the topology in a certain time window, and again, send this data to the reasoning module.

Finally, the semantic orchestration uses this data to generate a complete report of the diagnosis process, including the final set of *hypothesis*, the *conclusion* of the diagnosis, the start of it (semantically *whenHasStarted*) and its end time (semantically *whenHasEnded*). Also this report will include information about all the monitoring actions collected to carry out it (in property *hasCollectedInformation*). Each hypothesis has a possible error associated. Once, some errors for this case of study can be found in the model defined in Section 3.3.2. The reasoning API gives the information and the semantic orchestrator map them with the model.

- **NodeFault -** A node has been collapse or shut down.

- **ServerFault -** A server (host) has not be available.
- **OutPortRulesModified -** Rules from output actions is being changed.
- **InPortRulesModified -** Maths of type in port are being modified.
- **FlowsTimeoutModified -** Unexpected timeouts are being added to a node.
- **FlowsPrioritiesModified -** Flow priorities are changing.
- **DroppingLldpPackets -** Certain node is dropping LLDP packets.
- **PortRulesModified -** In and Out rules in a node are being changed.

So we store this final reports in Elasticsearch in a way that all the process of the diagnosis is stored so the phases that it has followed can could be consulted.

Listing 5.9: Diagnosis in Hypothesis Discrimination Phase

```
@prefix b2d2-diag: <http://www.gsi.dit.upm.es/ontologies/b2d2/diagnosis/> .
@prefix sdn-b2d2: <http://www.gsi.dit.upm.es/ontologies/sdn-b2d2/diagnosis/> .
@prefix sdn-ndl: <http://www.gsi.dit.upm.es/ontologies/sdn-ndl/> .
@prefix topology: <http://bayesiansdn/simulation2018271454_0/topology/> .
@prefix pr-owl: <http://www.pr-owl.org/pr-owl.owl#> .

<http://bayesiansdn/simulation2018271454_0/diagnosis/openflow2/1518011943/started> a
    b2d2-diag:Diagnosis ;
    b2d2-diag:hasDiagnosisPhase [ a b2d2-diag:HypothesisDiscrimination ] ;
    b2d2-diag:hasConclusion <http://bayesiansdn/simulation2018271454_0/diagnosis/
        openflow2/1518011943/finished#7> ;
    b2d2-diag:whenHasFinished 1518011945 ;
    b2d2-diag:hasCollectedInformation http://bayesiansdn/simulation2018271454_0/
        diagnosis/openflow2/1518011943#L2switch-38> ;

<http://bayesiansdn/simulation2018271454_0/diagnosis/openflow2/1518011943#L2switch-38> a
     sdn-b2d2:FlowInPortModified ;
    b2d2-diag:collectedFrom <topology:openflow2> ;
    sdn-b2d2:detectedIn <http://bayesiansdn/simulation2018271454_0/diagnosis/openflow2/
        L2switch-38> .
```

Listing 5.9 shows an example of a *Diagnosis* in its final phase. This is the "report" commented to introduce the section.

With this final achievement of getting an automated diagnosis for a specific node, the case of study is over for this master thesis. This case of study is intentionally prepared for the integration with a visualization system that will be presented in the next section.

5.6 Visualizing Data

Our prototype include a visualization system. All the data collected in the selected case study, as explained in in this chapter, will be easy to access and consult in this visualization system. A series of graphics developed specifically for this master thesis will be part of this visualization system, grouped in a dashboard.

This is possible thanks to technologies such as Polymer and Web components, implemented to ease the task as a framework called Sefarad (Section 2.5.2). Sefarad allows consultation and visualization of semantic data. It makes use of Elasticsearch as the search engine and Google Chart's API to show the information stored in it. Also it explores linked data by making SPARQL queries to the chosen endpoint.

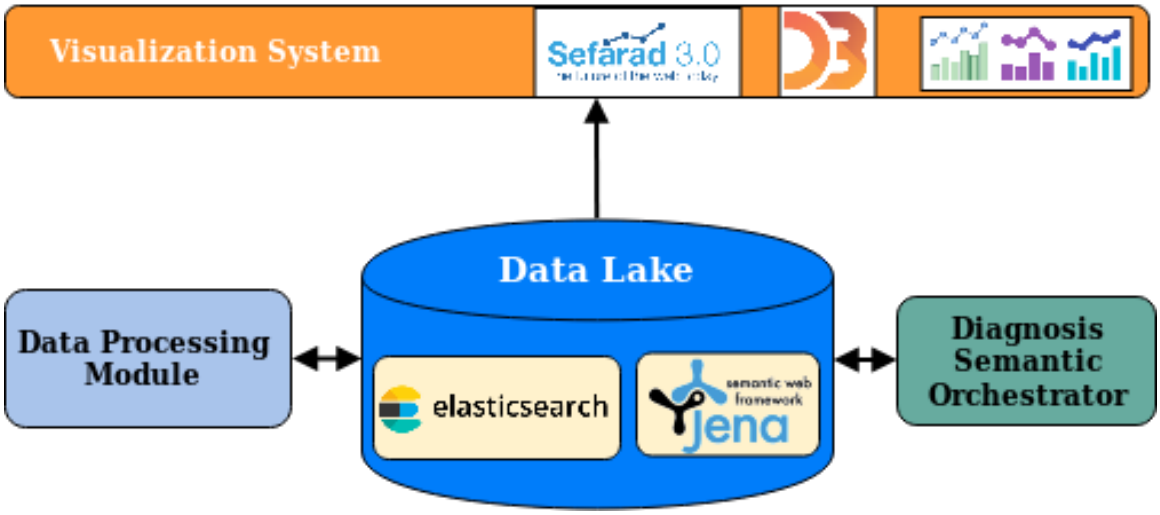


Figure 5.8: Visualization system operation schema

Figure 5.8 shows the process commented before. The visualization system queries Elasticsearch and Fuseki, depends on the kind of data it wants. For doing this, a Elasticsearch client plugin for Polymer is used in the case of Elastisearch data and a plugin for D3 in the case of making SPARQL queries.

This technologies and other Javascript libraries will allow the prototype developed in this master thesis having a complete dashboard as a visualization system. This will be divided in three parts. First of it will be related to the diagnoses performed and the symptoms collected, we detail this in Section 5.6.1. On the other hand, Section 5.6.2 describes other part of the dashboard, that part is in charge of showing the updated network statistics and the network status over time. Also a SPARQL interface to allow making queries directly.

5.6.1 Network Diagnosis and Symptoms

This section will illustrate the part of the visualization system where the diagnosis results are showed over time. It will be based in the approach followed for the case study, hence, we find in it terms defined in 3 for SDN diagnosis, as well as terms from the B2D2 diagnosis model for the diagnosis process, that has been introduced mainly in 4.

Contrary to what we will see in section 5.6.2, in this section, we do not use SPARQL for retrieving data to the dashboard. This is because diagnosis data are not such complex data as the network data, so Elasticsearch engine to get data for the dashboard will be enough. Moreover, we can take advantage that the native integration with Polymer components contributes to the performance of the visualization.

The capacity of this system resides among other features in showing the symptoms that the monitoring system will be collecting in real-time. Figure 5.9 will be showing the symptoms that the symptoms detector module will collect. At the top of the figure we can see a chart that will report information about the number of symptoms the system has collected. Below we can see in the figure a pie chart that gives information of the distribution this symptoms will have. The legend of this chart is composed by semantic terms (the prefix *sdn-b2d2* can be seen)

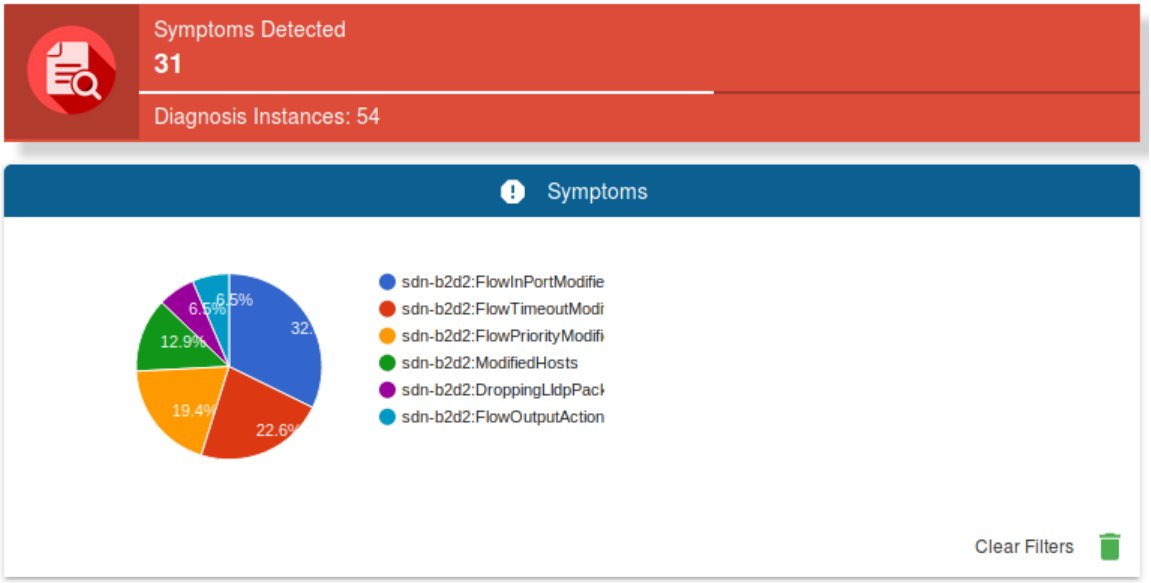


Figure 5.9: Symptoms Detected

Other information that we have wanted to show is where the diagnoses reported are located. Figure 5.10 illustrates at the bottom a pie chart with the aggregation of locations of the diagnoses generated by the system, in this case, 2 nodes have been diagnosed, in fact,

one of them, several times. On the other hand, at the top of the Figure 5.10 we can see a chart that reflects the number of diagnoses that have been completed, in this case 27[17]. It is worth highlighting that there are more diagnoses instances apart from finished, due to the other phases through which a diagnosis passes.

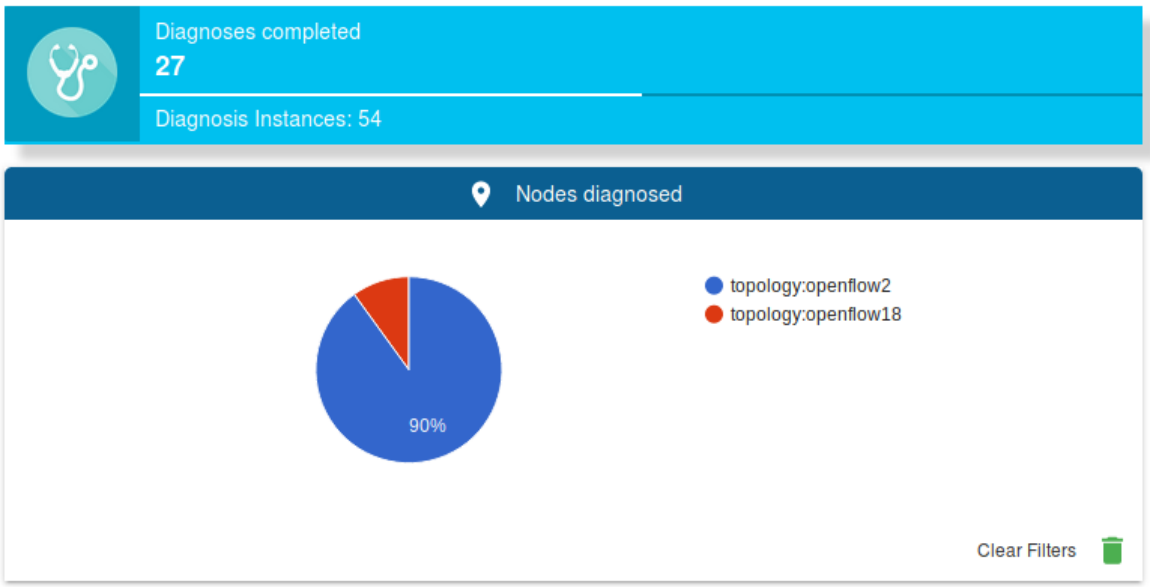


Figure 5.10: Diagnoses number and its location

In addition to this charts, we have developed a table visualization in which we will be presenting the diagnosis reports. The network operator will have here the detailed information for each diagnosis, including the possible faults, the probability of it, the symptom that has started the symptom or the node in which this symptom has been collected. All the information following the semantic model described in Section 3.3.1.1. Moreover, we can also see all the hypothesis that the reasoning module has taken into account and access to the resource in a semantic format. Figure 5.11 show the table visualization developed.

On the other hand, it was necessary adding to the dashboard additional information referred to the temporary dimension of the diagnoses and symptoms. Figure 5.12 and Figure 5.13 show both charts developed for this purpose. The network operator will see for each moment the amount of diagnosis reports the system is generating, as well as the number of symptoms are being detected in the network.

Also we can add the information about the possible faults that each diagnosis points. This information is presented additionally as a time chart when we count the number of times a diagnosis concludes a fault and present as a series depending on the time. This is shown in figure 5.14.

[17]A big number since we have provoked errors during the simulation.

Diagnoses performed

id	Possible Faults	Probability	Date	Started By Symptom	Collected From Node	Detected In Flow
1518013182	No Fault	0.989	7 Feb 2018 15:19:51	Flow Priority Modified	topology:openflow2	L2switch-38
1518013152	No Fault	0.989	7 Feb 2018 15:19:18	Flow Priority Modified	topology:openflow2	L2switch-38
1518013125	In Port Rules Modified	0.579	7 Feb 2018 15:18:56	Flow In Port Modified	topology:openflow2	L2switch-38
1518013086	In Port Rules Modified	0.579	7 Feb 2018 15:18:34	Flow In Port Modified	topology:openflow2	L2switch-38
1518012606	In Port Rules Modified	0.579	7 Feb 2018 15:10:24	Flow In Port Modified	topology:openflow2	L2switch-38
1518012579	Flows Priorities Modified	0.1	7 Feb 2018 15:09:52	Flow In Port Modified	topology:openflow2	L2switch-38
1518012548	In Port Rules Modified	0.579	7 Feb 2018 15:09:14	Flow In Port Modified	topology:openflow2	L2switch-38
1518012509	In Port Rules Modified	0.579	7 Feb 2018 15:08:35	Flow In Port Modified	topology:openflow2	L2switch-38
1518012479	No Fault	0.989	7 Feb 2018 15:08:07	Flow Priority Modified	topology:openflow2	L2switch-38
1518012452	Server Fault	0.947	7 Feb 2018 15:07:59	Flow Priority Modified	topology:openflow2	L2switch-38
1518012421	No Fault	0.989	7 Feb 2018 15:07:14	Flow Priority Modified	topology:openflow2	L2switch-38
1518012382	No Fault	0.989	7 Feb 2018 15:06:27	Flow Priority Modified	topology:openflow2	L2switch-38
1518012355	Flows Priorities Modified	0.1	7 Feb 2018 15:05:59	Flow Output Action Modified	topology:openflow2	UF$TABLE*0-27
1518012343	Flows Priorities Modified	0.1	7 Feb 2018 15:05:47	Dropping Lldp Packets	topology:openflow2	UF$TABLE*0-27

Figure 5.11: Diagnoses Table

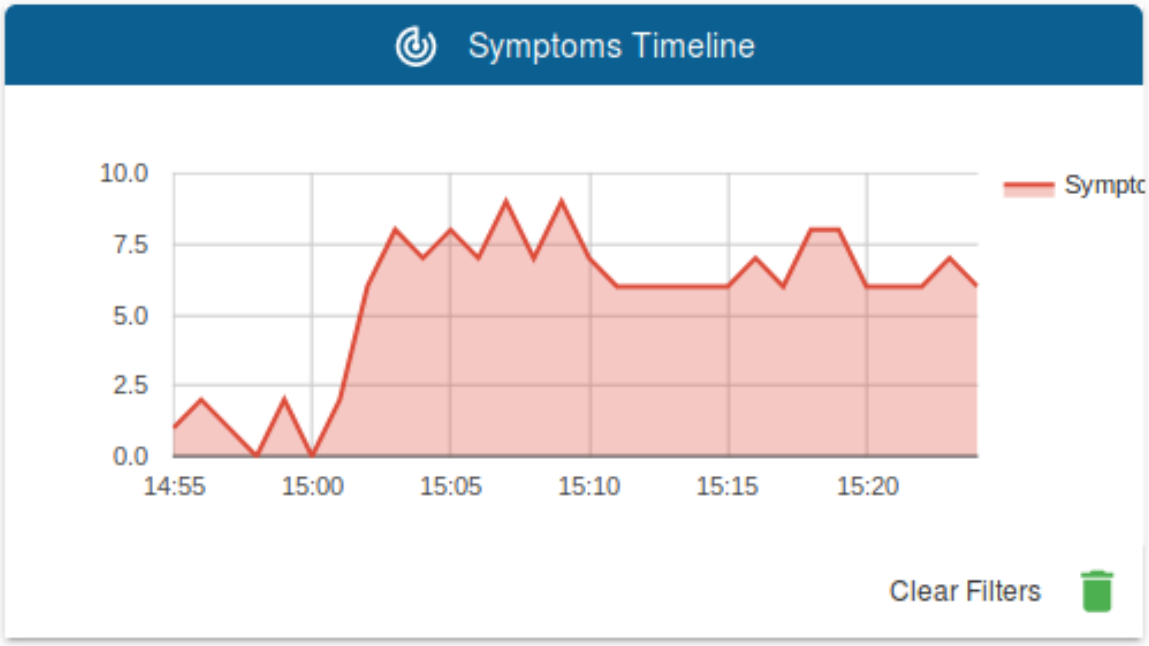


Figure 5.12: Symptoms timeline

Ultimately, to complement this information about the diagnoses and symptoms detected, we have developed a interactive widget that using SPARQL find the set of possible paths between two nodes within the network. This is possible thanks to having all the information in RDF format. We have develop this functionality as a use case when we need to know for example, if there are any diagnosis reported among the switch that belongs to the path between two host. Figure 5.15 shows the possible paths between two hosts (selected by its mac address). In strong yellow the two selected hosts and in light yellow the possible steps where the data packets would transit[18].

[18]The process could be better selecting among the interfaces that will be in forwarding mode

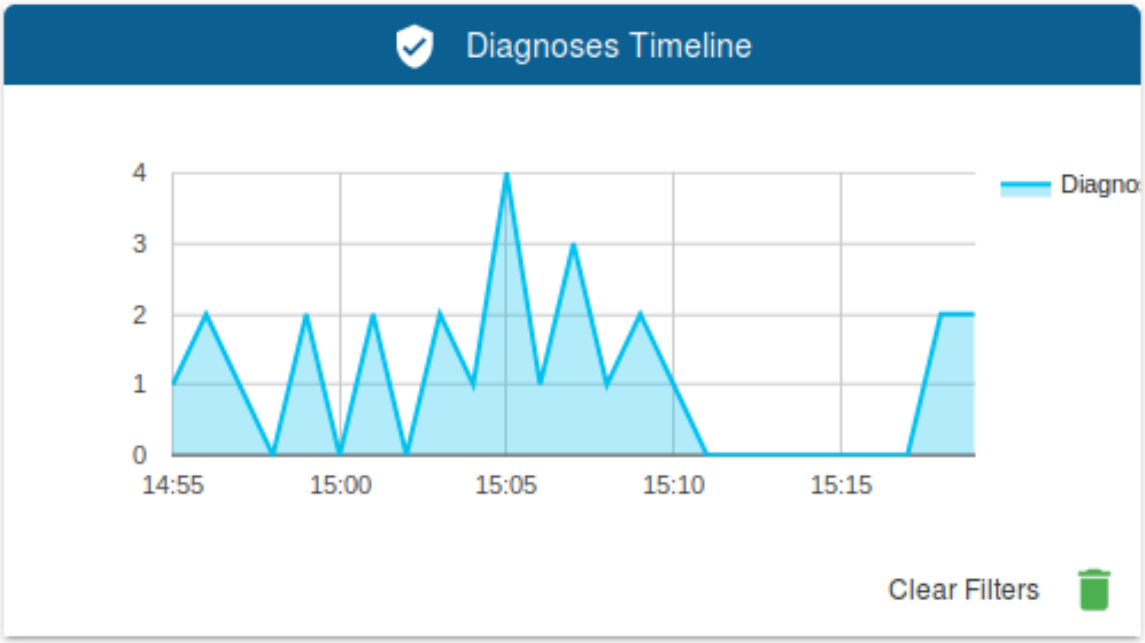


Figure 5.13: Diagnoses timeline

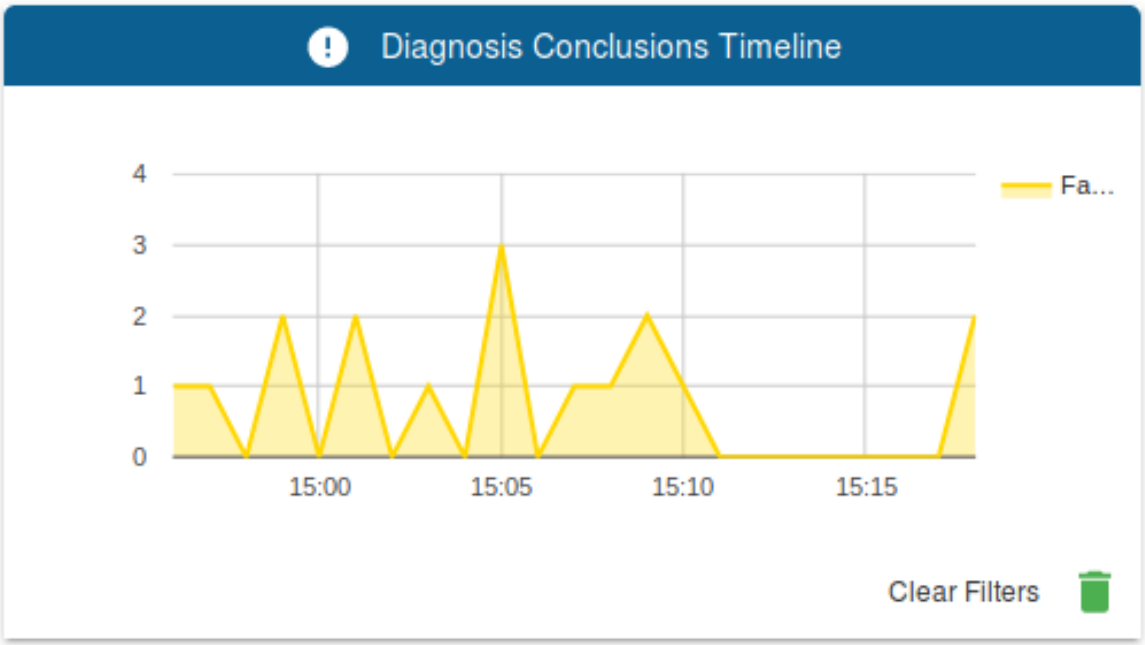


Figure 5.14: Possible faults timeline

The SPARQL query to achieve this information is also included in Listing 5.10[19]. The principal model queried is the structural model extended in this master thesis but also an

[19]Note that the query is more complex than this, but the main idea of how to do it is exposed.

advanced use of the nml[20]

Listing 5.10: Example of SPARQL query to find paths in the network topology

```
BASE <http://bayesiansdn/SIMULATION_ID/>
PREFIX topology: <http://bayesiansdn/SIMULATION_ID/topology/>
PREFIX sdl: <http://www.gsi.dit.upm.es/ontologies/sdn/sdn-ndl#>
PREFIX nml: <http://schemas.ogf.org/nml/2013/05/base#>
PREFIX ndl: <http://cinegrid.uvalight.nl/owl/ndl-topology.owl#>

SELECT DISTINCT ?step WHERE{
  {SELECT ?topology WHERE {
    ?topology  a nml:Topology .
    ?topology sdl:hasSnapshot ?snapshot .
    ?snapshot nml:start ?start .}
  ORDER BY DESC(?start)
LIMIT 1
}
?topology sdl:hasSnapshot ?toposnaphost .
?toposnaphost nml:start ?ts .
?topology nml:hasNode ?s .
VALUES ?s {topology:openflow12 topology:openflow11}
?step a sdl:OpenflowNode .
?s ndl:connectedTo|^ndl:connectedTo+ ?step .
FILTER EXISTS {?step (ndl:connectedTo) ?s}
}
```

To help the network operator to find the diagnosis to the path that have been highlighted, near that chart is attached another table visualization with the diagnoses realized to the nodes of the path discovered (if there are any). Figure 5.16 shows how this have been implemented. Also the list of the nodes of the path is given in a list to complete the path finder.

Also the dashboard include additional features to help the network operator to fix the network. These are some basic requirements, such as the allowing of search faults, or select the time range in which the data is showed to help the used isolates the problem or find more specific information concerning a time moment. Figure 5.17 show this features.

5.6.2 Network Status and Statistics

In this section we describe the most important features of our visualization system at the time of providing a network operator a visual snapshot of the network status. This will be possible thanks to some Javascript libraries, such as D3.js [9]. Also, it is worth stressing this

[20]nml is described in section 3.2.1.3

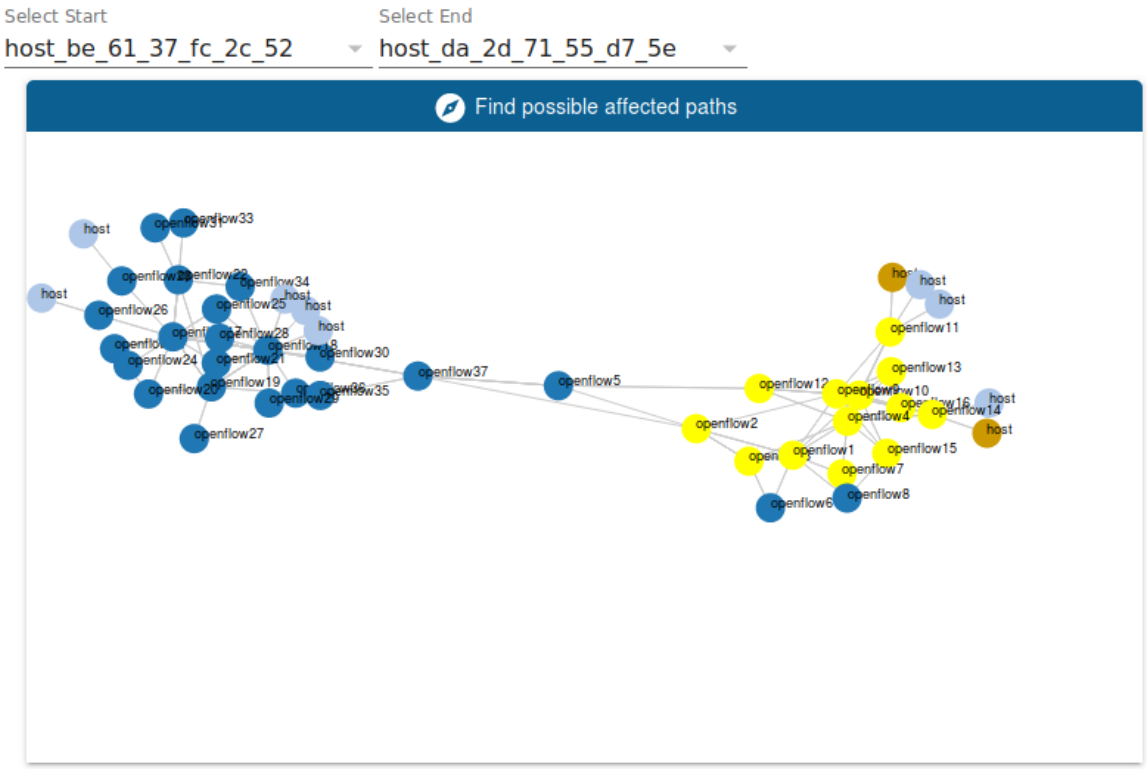


Figure 5.15: Finding affected paths

Find Diagnosis for paths

Collected From Node	Diagnosis id	Possible Faults	Date
topology:openflow2	1518013182	No Fault	7 Feb 2018 15:19:51
topology:openflow2	1518013152	No Fault	7 Feb 2018 15:19:18
topology:openflow2	1518013125	In Port Rules Modified	7 Feb 2018 15:18:56
topology:openflow2	1518013086	In Port Rules Modified	7 Feb 2018 15:18:34
topology:openflow2	1518012606	In Port Rules Modified	7 Feb 2018 15:10:24
topology:openflow2	1518012579	Flows Priorities Modified	7 Feb 2018 15:09:52
topology:openflow2	1518012548	In Port Rules Modified	7 Feb 2018 15:09:14
topology:openflow2	1518012509	In Port Rules Modified	7 Feb 2018 15:08:35
topology:openflow2	1518012479	No Fault	7 Feb 2018 15:08:07
topology:openflow2	1518012452	Server Fault	7 Feb 2018 15:07:59
topology:openflow2	1518012421	No Fault	7 Feb 2018 15:07:14
topology:openflow2	1518012382	No Fault	7 Feb 2018 15:06:27
topology:openflow2	1518012355	Flows Priorities	7 Feb 2018 15:05:59

Figure 5.16: Diagnoses for paths

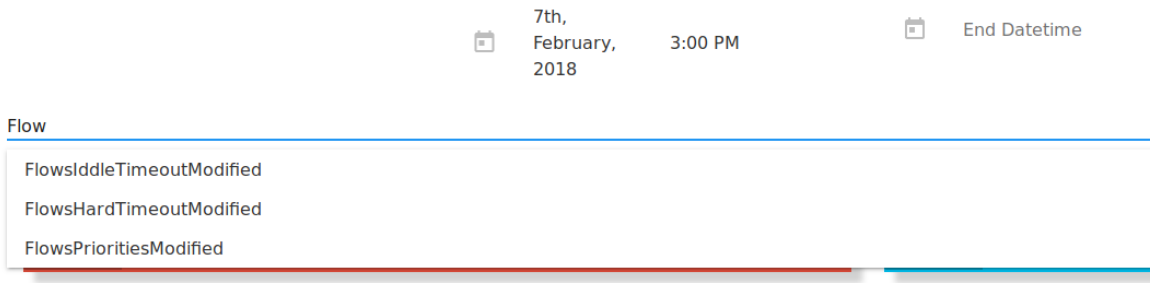


Figure 5.17: Dashboard Filters and search

part of the visualization system is possible thanks to Jena Fuseki and SPARQL. Structured information and a engine to query allows the creation of very useful and informational charts about the network statistics and status. For example, a network operator could consult this system and get statistics from the network choosing a time range or a specific node. This big capacity of the system will allow the network operator search more information about the diagnosis reported, to find the root problem.

So this will include the topology at the time of retrieving information from the dashboard. Figure 5.18 shows how this chart appears in the system. This visualization is developed based on a D3.js implementation of a visualization algorithm, the force directed graph [16]. So we have adapted this d3.js visualization to our data and our dashboard based on Polymer web components[21].

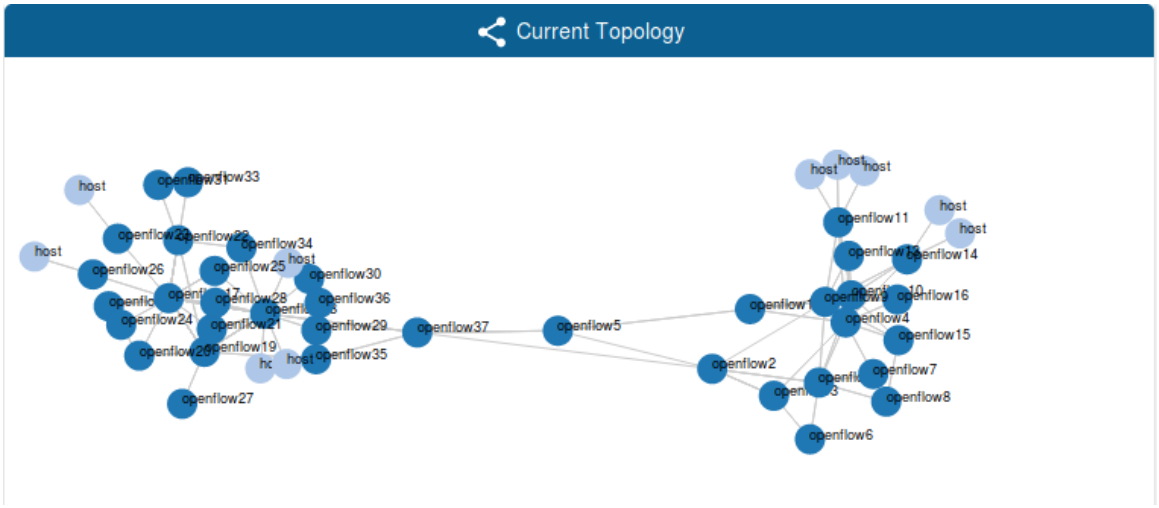


Figure 5.18: Current Topology chart

With this visualization, a general idea of how the monitored network topology is set up can be easily shown. In it, we can extract instant and first information about the neighbors

[21]For more information, refers 2.5.2

of a particular node or to which node a server is connected. Also to get a general glance of the network we include some charts with the number of network elements that compounds the network. Figure 5.19 shows this.

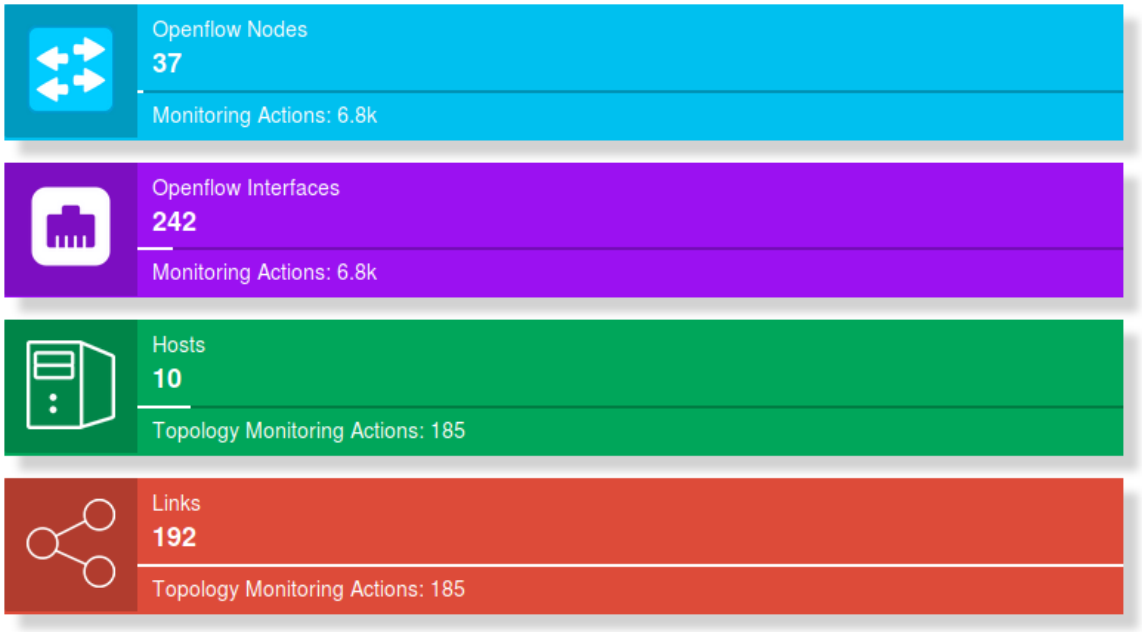


Figure 5.19: Chart with the number of network elements

As mentioned, thanks to SPARQL we can achieve time graphs as showed in Figure 5.20. This chart gives us information about the total traffic present in the network. The network simulation after its boot hits periodically peaks of approximately 500 mbps. In this case, we use a javascript library mentioned in Chapter 2 [30] to implement this kind of charts. This library is a middleware software that takes information from a SPARQL query and connects with the google visualization api to achieve charts with the information that the query contains.

An example of the queries used to feed this library and create this chart can be shown in Listing 5.11. This is a simple SPARQL query to return 2 columns, one of them, the data of the time in each data entry, and the other, the sum of the traffic transmitted by all the interfaces of all nodes in the network.

Listing 5.11: Example of SPARQL query to retrieve network total traffic

```
BASE <http://bayesiansdn/simulation2018271454_0/>
PREFIX topology: <http://bayesiansdn/simulation2018271454_0/topology/>
PREFIX sdl: <http://www.gsi.dit.upm.es/ontologies/sdn/sdn-ndl#>
PREFIX nml: <http://schemas.ogf.org/nml/2013/05/base#>
PREFIX ndl: <http://cinegrid.uvalight.nl/owl/ndl-topology.owl#>
```

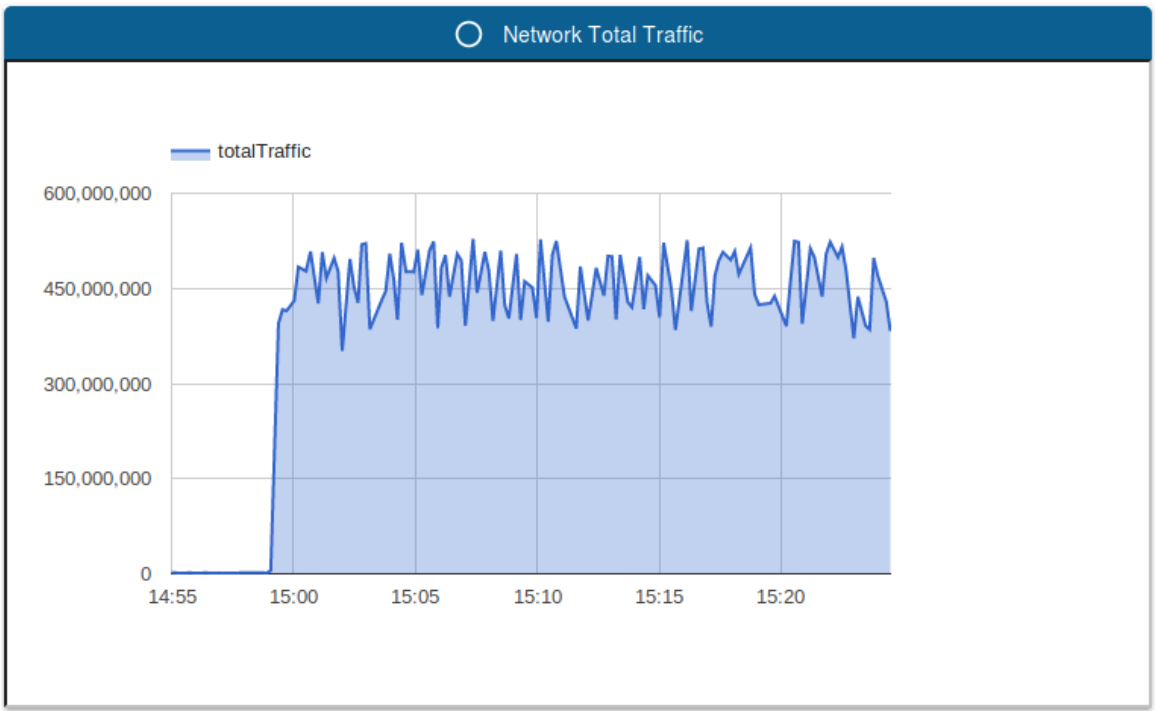


Figure 5.20: total traffic of the network

```
PREFIX schema: <http://schema.org/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?time (xsd:float((SUM(?BytesTransmitted))) as ?totalTraffic)  WHERE {
  ?nodeinstance a sdl:OpenflowNode .
  ?nodeinstance schema:dateTime ?timestamp .
  BIND(xsd:time(strbefore(strafter(?timestamp, 'T'), '.')) as ?time)
  ?nodeinstance ndl:hasInterface ?interfaces .
  ?interfaces sdl:describesInterface ?interface .
  ?interfaces sdl:hasNetworkStats ?stats .
  ?stats sdl:bytesTransmitted_rate ?BytesTransmitted .
}
GROUP BY ?time
ORDER BY ?time
```

Our prototype is created following an approach of a simple node diagnosis, not upon a group of them. Thus we allow the network operator to select a node to shown up the traffic is supporting over the time. Figure 5.21 shows the interactive chart from the visualization system that will allow the network operator to select a node among all the network nodes in a menu to know information about its total traffic. The SPARQL query in charge of this will get the argument depending on the node selected and sums the traffic received and transmitted in each of its interfaces. As it can be seen, at a certain hour we can detect a strange peak of 12 mbps.

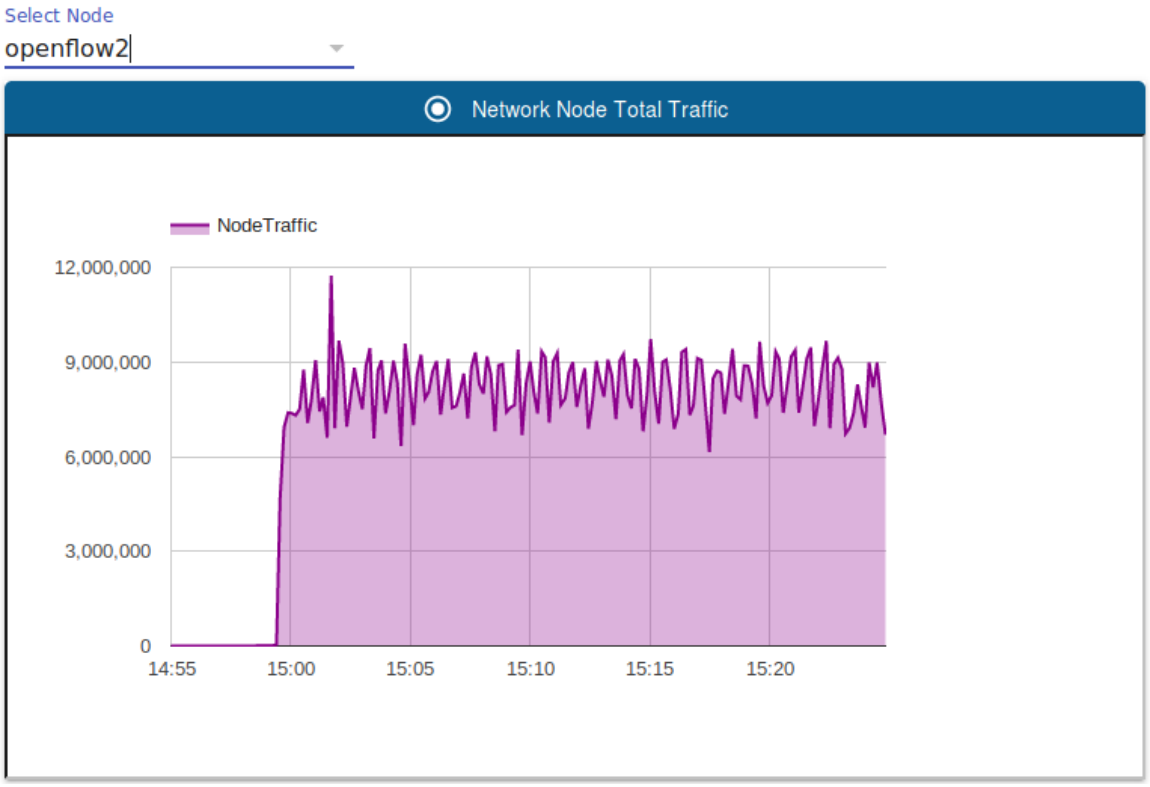


Figure 5.21: Handled traffic for a specific node

Others charts to show the possible congestion in the network are part of the dashboard developed. For example, thanks to SPARQL we can get a query that aggregates the number of active flows in a node. This graph is showed in a chart with all Openflow nodes presented in the network. For example, a certain simulation, Figure 5.22 will show us that "openflow18" is the node with most active flows. This is also useful information as commented before for detect congestion problems.

Moreover, in the dashboard also there will be a chart for showing the comparison between the packets looked up by a node and the packets that fit with its flow matches. Since, as is known, a node that implements Openflow will only handle the packets that fits with one of the match of its flows, the rest will be dropped. So this information showed by our prototype, that will be something like the Figure 5.23 shows. This two charts are possible due to the flow table stats that Opendaylight provides information upon. An example of this information is illustrated in Listing 5.12

Listing 5.12: JSON-LD serialized Example of Flow Table Stats data

```
{
```

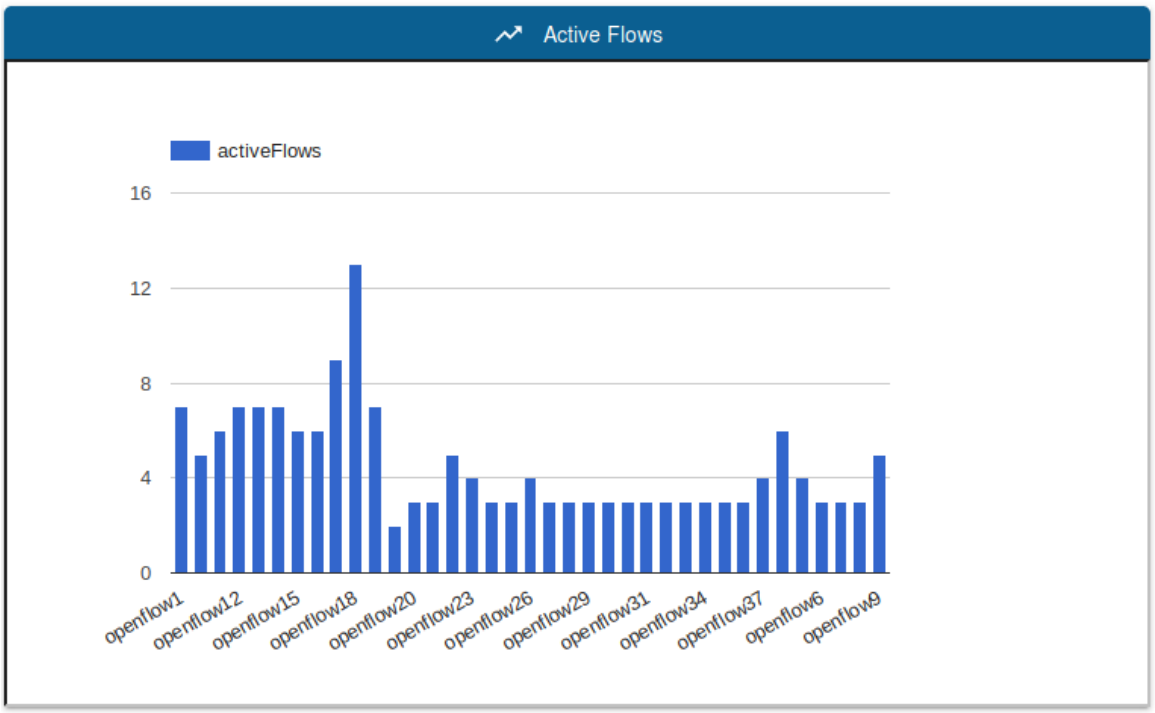


Figure 5.22: Active flows in the nodes

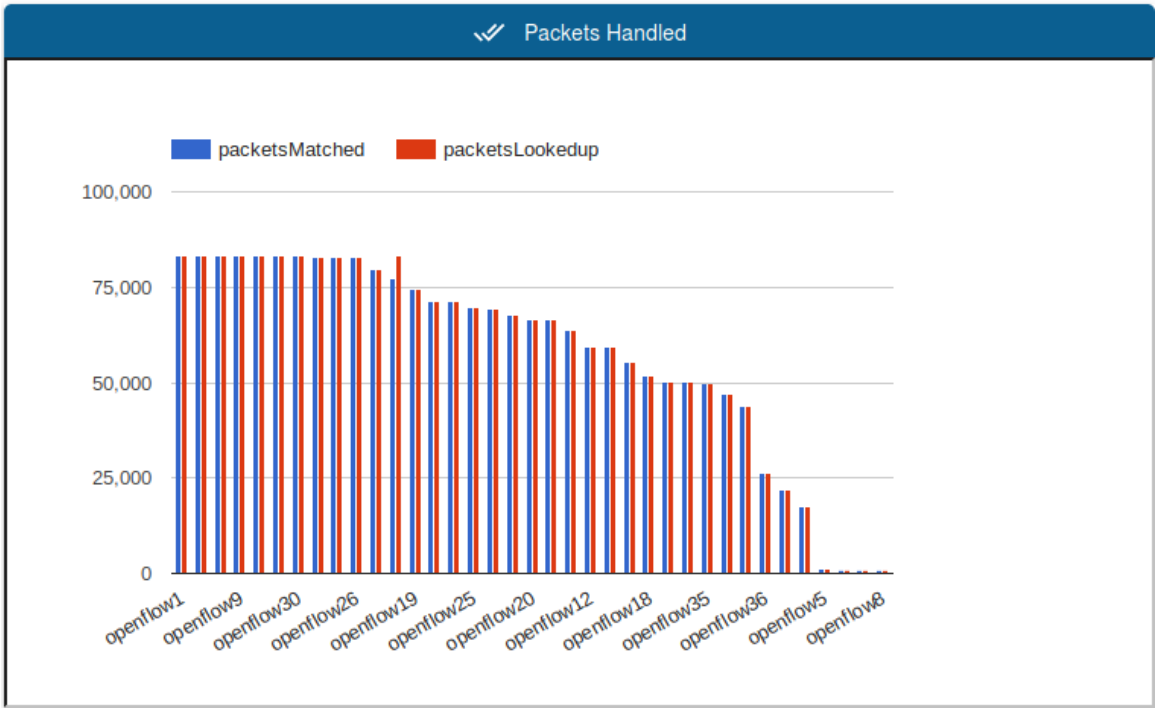


Figure 5.23: Packets handled/matched comparison

```
"@id": "table0",
```

```
    "@type": "FlowTable",

    "hasNetworkStats": {
          "@id": "flowTableStats0",
          "@type": "FlowTableStats",
          "packetsLookedup": 4410,
          "packetsMatched": 4409,
          "activeFlows": 3
        }
}
```


Since the user of this system can select a data time range, he could take a glance to a node that a a certain time window has been looking up more packets than the matched ones, this could be due to the lack of necessary flows or that the packets are not being the correct ones sent to that node.

Going down to the level of interfaces, we have provided this dashboard with more statistics about the possible anomalies that could be given in them. Note that this are different to the symptoms detected by the prototype regarding the flows configured. This anomalies are directly detected by Opendaylight controller. Opendaylight through its Stats Manager[22] annotate some stats about the interfaces configured in a network, like the errors received or transmitted, the drops received or transmitted, etc...

So Figure 5.24 shows the number of anomalies that Opendaylight has detected in its interfaces, and Figure 5.25 how a network operator can examine more information about this topic, seeing that information on each node with respect to time.

Also the status of interfaces are important, in this case to complement the automated diagnosis generated in the prototype. So we allows the network operator to take a glance to the interface status of each interface selecting a node from a menu with all nodes. Some information that Opendaylight retrieves is then showed as Figure 5.26 illustrates.

Among the information retrieved, the Spaning Tree Protocol (STP) status. STP is a typical way of avoiding loops in network environments[23]. In Opendaylight the L2 switch module provides it[24]. So the layer introduced by L2 switch adding a status to each node interface is a information to consider. Also data from Opendaylight monitoring of interfaces, as if the link to which the interface is up, or if this is blocked.

[22]The stats collected by Opendaylight are explained in its docs: https://wiki.opendaylight.org/view/TSDR:Collected_Metrics

[23]We did not mentioned yet so here you can get more information: https://es.wikipedia.org/wiki/Spanning_tree.

[24]Official Documentation of L2-switch: https://wiki.opendaylight.org/view/Project_Proposals:L2_Switch

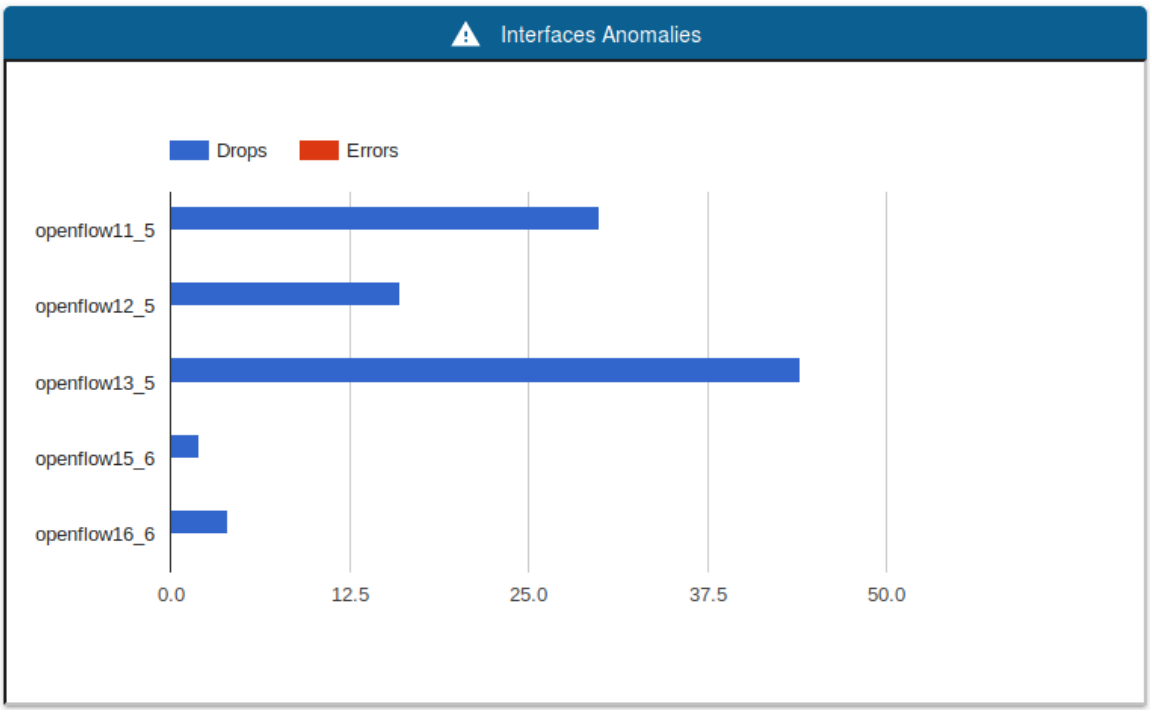


Figure 5.24: Number of interfaces anomalies monitorized

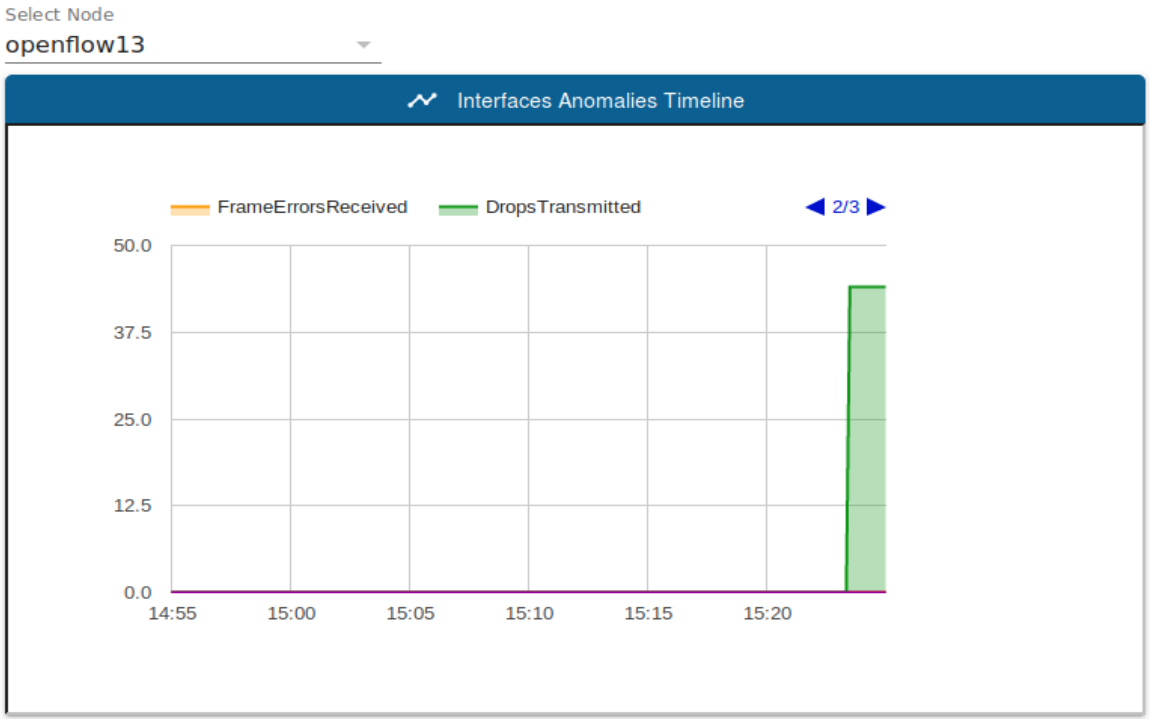
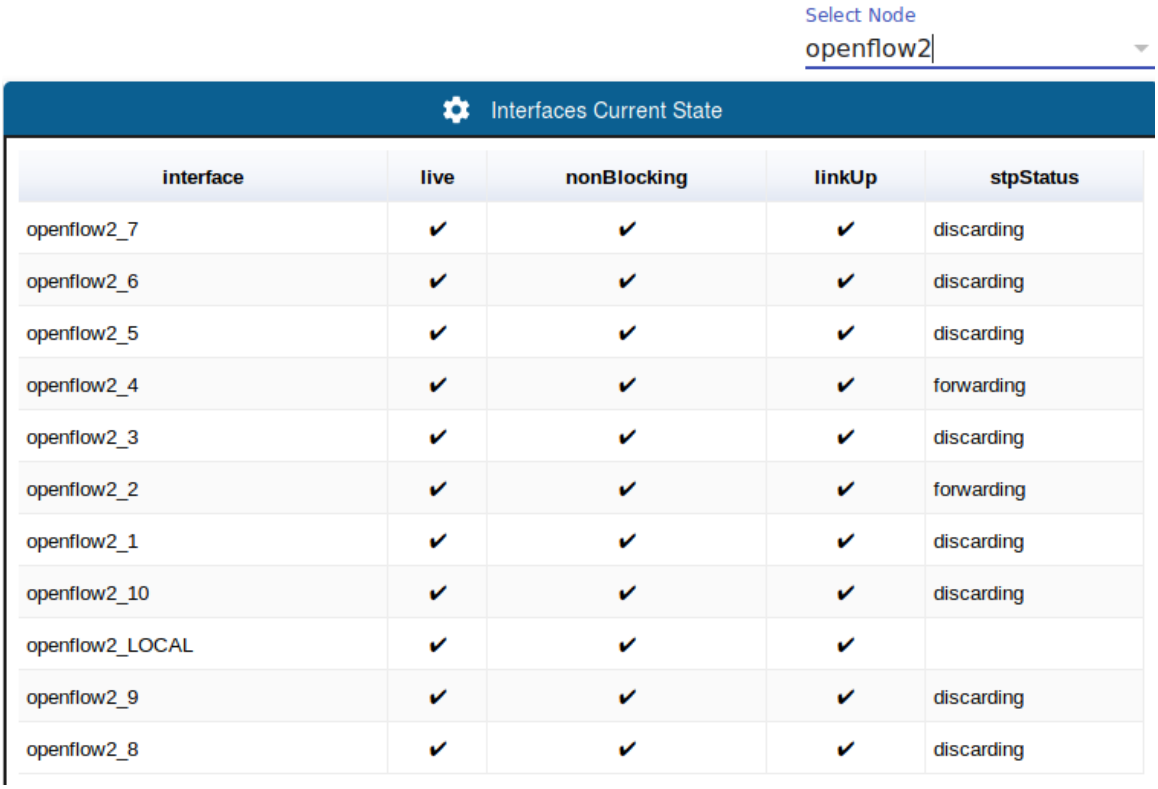


Figure 5.25: Timeline for the anomalies detected

On the other hand, we can see also the bytes received and transmitted by an interface, in a way similar to the network traffic visualizations commented before. In this case for a unique interface from a node.



interface	live	nonBlocking	linkUp	stpStatus
openflow2_7	✔	✔	✔	discarding
openflow2_6	✔	✔	✔	discarding
openflow2_5	✔	✔	✔	discarding
openflow2_4	✔	✔	✔	forwarding
openflow2_3	✔	✔	✔	discarding
openflow2_2	✔	✔	✔	forwarding
openflow2_1	✔	✔	✔	discarding
openflow2_10	✔	✔	✔	discarding
openflow2_LOCAL	✔	✔	✔	
openflow2_9	✔	✔	✔	discarding
openflow2_8	✔	✔	✔	discarding

Figure 5.26: State of the interfaces of a node

All this information will be useful in some cases. For example, let us suppose that a switch, for example "openflow2", is dropping packets unexpectedly thanks to the chart illustrated in Figure 5.27. We consult the diagnoses automatically reported by the system but there is none concerning this node. In this case we could consult the interface state and discover that is down, or bad configured. So this kind of information completes the one give by the fault management system.

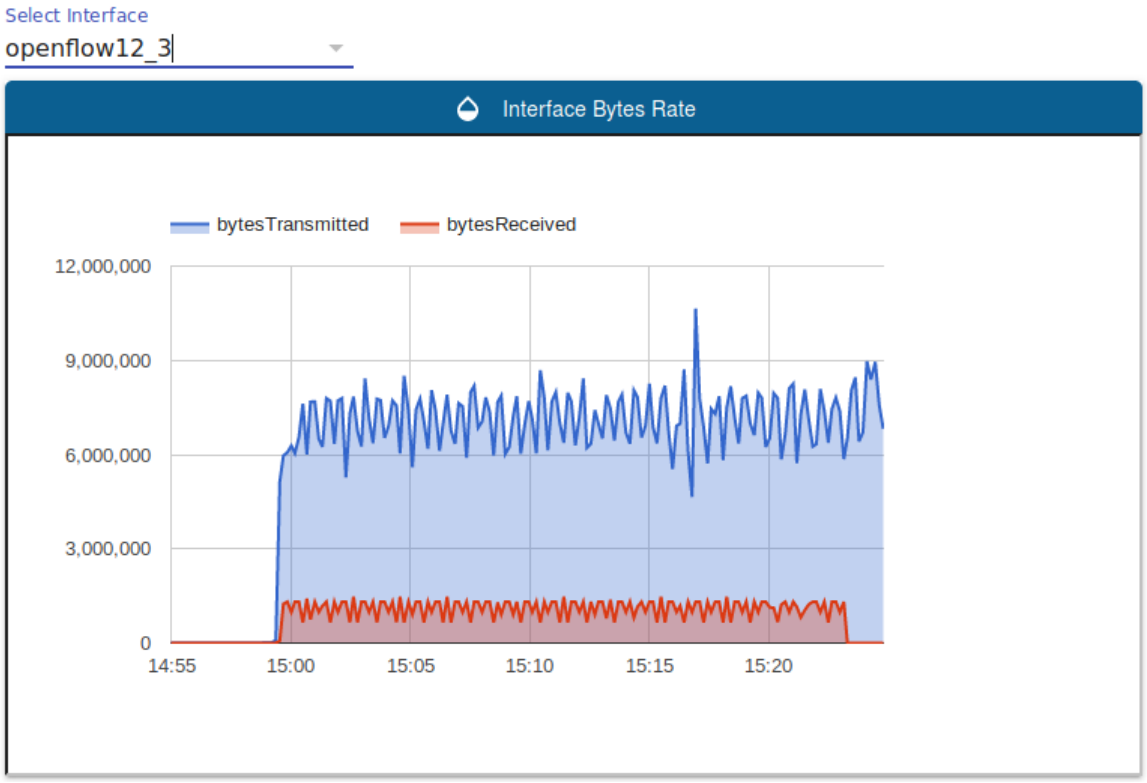


Figure 5.27: Interface bitrate

CHAPTER 6

Conclusions and Future Work

In this project we have developed a complete fault diagnosis system that was our initial objective but also, we have achieved more goals in this master thesis following some the key points developed in the project. So this objectives and the general conclusions of the project are presented in this chapter. Furthermore, this research have not finished yet. We expose some of the possible future task that will be related with this master thesis. This will be in the last section of this chapter.

6.1 Conclusion

The first conclusion is that the goals proposed before the start of this project have been achieved. We have extended the current work in network ontologies to include SDN concepts, but also, we have extended a diagnosis model to embrace SDN diagnosis peculiarities in it. Moreover, we have designed a system where using this models it is possible to perform automatic fault diagnosis of a SDN network.

After defining this models and designing a system, following a case study, we have defined and developed a prototype that first, collects data from an OpenDaylight-controlled SDN simulator, then, it processes and enriches the collected network data. We have take advantage of this data processing, the diagnosis model extended and a external reasoning module to allow this prototype to generate diagnosis of the nodes. To give value to the system, an interactive dashboard for the network administrator to show all this information has added to the prototype.

So the project has fulfilled the proposed objectives, but we must draw a general conclusion of the project.

SDN provides a number of benefits in the virtualization and management of network services. Moreover, Big Data technologies provide the foundation for collecting and processing huge amounts of raw data from the telecommunication network.

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 develop a system that not only performs diagnosis of a network environment, but also it gets it following a diagnosis model representing all the fault diagnosis steps data following the RDF standard. For that, a significant effort has been dedicated to extend the models created previously to represent networks in a semantic way.

We have implemented a prototype that collects data from a network environment and performs diagnosis for its nodes over the time following a concrete case study. This prototype take advantage of all the semantic data generated to present it in a visualization system for network administration.

6.2 Future Work

We have developed a fully functional prototype, but the research line does not end here. In this section we will expose the main possible future works that we can start working from

now on.

Publishing the extension done in this master thesis of an RDF-based SDN description language. The work done to achieve a infrastructural description model of the SDN environment could be useful for the research community.

Extending the case study and embrace more possible fault situations, coping more complex cases, in order to broad the coverage of possible faults within the monitored network.

Moreover, the designing and implementation of a self-healing service is another important aspect that will be addressed as future work. This system could be integrated to this prototype.

We would also want to introduce new services in the simulation, such as point-to-multipoint streaming, which could be a very good point to the features developed in the prototype (finding the paths in this kind of situations is more useful).

APPENDIX A

Impact of the project

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. For its part, the companies and business need better computer network systems, that does not need of continuous repairing. Specifically, companies that receives benefit through this project or the investigation around the new network paradigms and the diagnosis of network equipment, are the IT companies. In this appendix, we give general ideas of the social, economic, environmental and ethical implications of such system.

In this appendix, we are going to talk about the possible social, economic and environmental impact that this project could have in Sections A.1, A.2 and A.3, respectively. We will also give the possible ethical and professional implications of such project in Section A.4.

A.1 Social Impact

We could expect a social impact of our system in the user quality of service and experience. Due to faulty computer networks could have impact in our everyday life. If we are consuming any type of content through the Internet a faulty computer network could affect us. Even more, if we are working and faults in a computer network do not allow us to continue working. So with the development of this project this impact could be lower.

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, 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.

In fact, this is a weak point in general for SDN environments, the security. Reducing the number of devices with control of the network you are increasing the risk, because an external attacker only needs to focus in the network controller. So this could be a negative impact in the society, but this is not all related to our project, is on the SDN technology mainly.

So security-wise, our system is vulnerable to some attacks but in particular, our project, since it depends on the data sources used for diagnosing, also an attack that disables one or more sources could hamper the diagnosis of the network.

A.2 Economic Impact

The development of this project is under the development and improvement of the SDN technology. Using SDN implies laying aside legacy networks. Legacy networking is costly since there are many vendor-specific protocols and solutions, it is very expensive to develop and deploy autonomous managing solutions that work in any network environment. Of course hiring a group of networks engineers even more.

Thanks to SDN, 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 cut costs on network management.

A.3 Environmental Impact

On one hand, automated fault diagnosis can avoid certain accidents due to the non communication of sensible and key infrastructures, such as power stations or train lines. An unexpected, uncontrolled fault in a computer network in such environments could potentially cause damages or harm the environment. Taking into account that in the future, SDN will be the way companies deploy its computer networks infrastructures, the diagnosis of this networks as done in this master thesis could contribute to reduce environmental impact.

On the other hand, projects like this, that not need of physical infrastructure to form a test bed allows to do as much test as possible. This is mainly thanks to to the virtualization component that retrieves SDN. This approach saves in need to manufacture switch and routers for test beds. Moreover, the essence of Openflow protocol allows almost every computer can work as a network object so this kind of SDN features also could contribute to the the non-mass production of network components.

A.4 Ethical and Professional Implications

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, process that we could theoretically do but we do not. Thus, 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.

The main risk of our developed prototype is that the user or network operator would make inappropriate use of the visualization system or the control of the network, making damage to the network. This could be a negative professional implication. Other case could be related to the storing of some linked data of the network as web resources. Someone can violate the security to access this data and take advantage of the knowdledge of the status of the network to perpetrate a cyber attack. But this case is also improbable due to the fact that this data is not very sensitive.

APPENDIX B

Cost of the System

We have designed and developed a prototype and in the process, we have incurred in some costs, mainly from the salaries of the developers, although some costs have been provoked by hardware needs. If we wanted to commercialize our software, we would need to spend money on better infrastructures and consider taxes on software products in Spain.

In this appendix, we evaluate the possible costs that the development of this system could have. First, we are going to describe the costs in physical resources needed for our system in B.1. Then, in Section B.2 we are going to estimate the costs regarding personnel needed to design and maintain the prototype. Finally, we consider the possible taxation involved in the selling of this software in Section B.3

B.1 Physical Resources

We need a computing machine powerful enough to run our system if we want to run all the modules at the same time. the recommended requirements for a computer that would run it without any kind of issue are approximately the following:

- **Hard Disk**: 30 GB
- **RAM**: 16 GB
- **CPU**: Intel i7 processor, 2.80 GHz

On average, a machine with such capabilities costs around 1,000 € as of 2018. But 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.2 Human Resources

In this section, we will take into account the time employed in the designing and developing of this system. We will give an approximation based on the average salary of a Software Engineers, to find the cost of the development of the project.

We estimate the cost in hours in developing this project to be around 900 hours. We have come up with this number by considering the ECTS credits of this master thesis, i.e 30 credits. ECTS credits[1] approximately compute as 25-30 work hours, so the amount of works in the development could be of 750 hours (the other 150 hours have been dedicated to write this document). We estimate the salary of an engineer developing the system to be around 2500 € per month (gross). If a month has 23 working days and consider 8 hours per working day, we predict the cost of developing the prototype to be around 10,000 €.

Once this system is created and deployed, we also expect maintenance costs. The system will be deployed but we would also have to adapt it to a real-life network. In order to adapt and maintein the system, we will need a Software Engineer working full-time, so he can address any issue that occurs and keep the system updated. If we consider the salary of a Software Engineer working at full-time to be around 2500 €, the maintenance of the system will round 30.000 € per year.

[1]European Credit Transfer and Accumulation System Official site: `https://ec.europa.eu/education/resources/european-credit-transfer-accumulation-system_en`

B.3 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 consider the taxes involved in the selling of software products, in this case, in Spain.

According to [14], there is a tax of 15% over the final price of the product, as regulated by the Statue 4/2008 of the Spanish law. In the case that we want to sell the product to a foreign company, we would need to consider possible cases of double taxation.

APPENDIX C

Ontologies

Two ontologies have been defined for this project. On one hand the ontology based on structural models for defining SDN, SDL ontology. On the other hand, the extension done in a Diagnosis model for cover SDN diagnosis domain, the B2D2-SDN ontology.

C.1 SDL Ontology

This ontology allows the modeling of data from SDN environments. The SDL schema consists of classes and properties. We introduce the concepts introduced for defining the data plane of an SDN network infrastructure. Also in this ontology we present the ideas introduced in SDL model, trying to approach some concepts of the control plane of a SDN network. The whole model behind this ontology is widely explained in Sec. 3.2.2.

Available at: `http://www.gsi.dit.upm.es/ontologies/sdl/`

C.2 B2D2-SDN Ontology

This ontology allows the modeling of diagnosis in SDN environments. To do that, we extend a diagnosis model B2D2 to embrace SDN diagnosis concepts. On one hand, this ontology has some symptoms that are possible to occur in a SDN network, mainly, error in flows from Openflow protocol. Second, the possible errors that are causing this symptoms, as openflow nodes error. The model behind this ontology is widely explained in Sec. 3.3.2.

Available at: `http://www.gsi.dit.upm.es/ontologies/b2d2/diagnosis/sdn/`

Bibliography

[1] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. Detecting heavy flows in the sdn match and action model. *arXiv preprint arXiv:1702.08037*, 2017.

[2] Sean Bechhofer. Owl: Web ontology language. In *Encyclopedia of database systems*, pages 2008–2009. Springer, 2009.

[3] Dave Beckett and Brian McBride. Rdf/xml syntax specification (revised). *W3C recommendation*, 10(2.3), 2004.

[4] Fernando Benayas de los Santos, Álvaro Carrera Barroso, and Carlos A. Iglesias. Towards an Autonomic Bayesian Fault Diagnosis Service for SDN Environments based on a Big Data Infrastructure. In *Proceedings of SDS 2018*, Barcelona, Barcelona, Spain, Abril 2018.

[5] Tim Berners-Lee. Notation 3-an readable language for data on the web. 2006.

[6] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

[7] Andy Bierman, Martin Bjorklund, and Kent Watsen. Restconf protocol. 2017.

[8] Martin Bjorklund. Yang-a data modeling language for netconf. 2010.

[9] Michael Bostock et al. D3. js. *Data Driven Documents*, 492:701, 2012.

[10] Dan Brickley and Libby Miller. Foaf vocabulary specification 0.91, 2007.

[11] Álvaro Carrera Barroso. *Application of Agent Technology for Fault Diagnosis of Telecommunication Networks*. PhD thesis, 2016.

[12] Álvaro Carrera Barroso and Carlos A. Iglesias. Exploiting Structural Knowledge using Network Description Language and Causal Models for Fault Diagnosis in Wireless Sensor Networks. *Journal of Networking Technology*, 6(4):135–147, December 2015.

[13] Enrique Conde-Sánchez. Development of a Social Media Monitoring System based on Elasticsearch and Web Components Technologies. Master's thesis, Universidad Politécnica de Madrid, June 2016.

[14] FRANCISCO DE and TORRE DÍAZ. La tributación del software en el irnr. algunos aspectos conflictivos. *Cuadernos de Formación. Colaboración*, 22(10), 2010.

[15] Rogério Leão Santos De Oliveira, Ailton Akira Shinoda, Christiane Marie Schweitzer, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, pages 1–6. IEEE, 2014.

[16] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[17] M. Ghijsen, J. van der Ham, P. Grosso, and C. de Laat. Towards an infrastructure description language for modeling computing infrastructures. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 207–214, July 2012.

[18] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide*.

[19] Diyar Jamal Hamad, Khirota Gorgees Yalda, and Ibrahim Tanner Okumus. Getting traffic statistics from network devices in an sdn environment using openflow. *ITaS*, pages 951–956, 2015.

[20] Carlos A Iglesias, Mercedes Garijo, José C González, and Juan R Velasco. Analysis and design of multiagent systems using mas-commonkads. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 313–327. Springer, 1997.

[21] Joseki Jena and Pellet Fuseki. Semantic web frameworks. *Oaks, R., University of Sterling. Disponible en http://www. cs. stir. ac. uk/courses/31Z7/posters/2014/rao. pdf*, 2004.

[22] Apache Jena-Fuseki. serving rdf data over http, 2014.

[23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[24] Alberto Pascual Saavedra. Sefarad documentation, 2018. `http://sefarad.readthedocs.io/en/latest/sefarad.html`.

[25] Jorge Perez and Marcelo Arenas. Querying semantic web data with SPARQL: State of the art and research perspectives.

[26] Muhammad H Raza, Shyamala C Sivakumar, Ali Nafarieh, and Bill Robertson. A comparison of software defined network (sdn) implementation strategies. 2014.

[27] ITUTG Recommendation. 800: Unified functional architecture of transport networks. itu-t g. 800, itu-t, february 2012.

[28] Ola Salman, Imad Elhajj, Ayman Kayssi, and Ali Chehab. Sdn controllers: A comparative study. pages 1–6, 04 2016.

[29] J. Schonwalder, M. Bjorklund, and P. Shafer. Network configuration management using netconf and yang. *IEEE Communications Magazine*, 48(9):166–173, Sept 2010.

[30] Martin G Skjæveland. Sgvizler: A javascript wrapper for easy visualization of sparql result sets. In *Extended Semantic Web Conference*, pages 361–365. Springer, 2012.

[31] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. Json-ld 1.0. *W3C Recommendation*, 16, 2014.

[32] Jeroen van der Ham, Freek Dijkstra, Roman Łapacz, and Aaron Brown. The network markup language (nml) a standardized network topology abstraction for inter-domain and cross-layer network applications. In *Proceedings of the 13th Terena Networking Conference*, 2013.

[33] Jeroen van der Ham, Freek Dijkstra, Roman Łapacz, Jason Zurawski, et al. Network markup language base schema version 1. Muncie, INOpen Grid Forum, 2013.

[34] Jeroen Van der Ham, Paola Grosso, Ronald Van der Pol, Andree Toonk, and Cees De Laat. Using the network description language in optical networks. In *Integrated Network Management, 2007. im'07. 10th IFIP/IEEE International Symposium on*, pages 199–205. IEEE, 2007.

[35] Jeroen J Van der Ham, Freek Dijkstra, Franco Travostino, Hubertus MA Andree, and Cees TAM de Laat. Using rdf to describe networks. *Future Generation Computer Systems*, 22(8):862–867, 2006.

Glossary

API Application Programming Interface

B2D2 BDI for Bayesian Diagnosis

B2D2-SDN B2D2 for SDN networks

BDI Belief-Desire-Intention

CPT Conditional Probability Table

INDL Infrastructure and Network Description Language

L2-switch Layer 2 switching

MD-SAL Model-Driven Service Abstraction Layer

NAT Network Address Translation

NETCONF Network Configuration Protocol

NDL Network Description Language

NML Network Markup Language

OGF Open Grid Forum

ODL Opendaylight

ONF Open Networking Foundation

OVSDB Open vSwitch Database Management Protocol

OWL Ontology Web Language

PR-OWL Probabilistic OWL

QoS Quality of Service

RDF Resource Description Framework

SDL Software-defined Networking Description Language

SDN Software-defined Networking

SNMP Simple Network Management Protocol

SPARQL SPARQL Protocol and RDF Query Language

SQL Structured Query Language

STP Spanning Tree Protocol

STP Tuple Database

UI User Interface

UML Unified Modelling Language

YANG Yet Another Next Generation