



Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

FINAL YEAR PROJECT

TITLE: Reinforcement Learning-Based Routing in SDN Networks

DEGREE: Telematics Engineering Degree

AUTHOR: Sergi Vera Martínez

DIRECTOR: Anna Agustí Torra

DATE: 14th of June, 2023

TITLE: Reinforcement Learning-Based Routing in SDN Networks

DEGREE: Telematics Engineering Degree

AUTHOR: Sergi Vera Martínez

DIRECTOR: Anna Agustí Torra

DATE: 14th of June, 2023

Overview

Ever since the concept of the Internet was born, it has experienced a massive increase in the number of users. Similarly, there has been an exponential increase in the amount of information that they transmit, partly because of new technologies and services that have emerged, such as cloud and real-time communication. Customers do now demand a fast and reliable interaction, without any data loss or undesired delays.

In order to meet these needs and keep up with the constantly-evolving technologies different non-profit organizations needed to devise a new technology, and this is when Software-Defined Networking (SDN) was born. Unlike traditional networks to date, it decouples the control layer from the data layer, leaving the act of forwarding traffic to the network device and delegating all network decisions to a controller, thus centralizing all the decisions, which improves network operability and agility. Despite all the improvements made with this technology, we still encounter the same problems of traditional networks regarding traffic routing. Conventional algorithms such as Dijkstra and Least Loaded (LL) based on the occupancy of the links, allocate traffic without considering the impact the chosen path could have if future traffic were to be introduced in the network.

The objective of this project is to create an algorithm capable of routing traffic taking into account the future impact. To evaluate the proposed mechanism of Dijkstra, we will first obtain some experimental results using the default link weights and then base these weights on the occupancy of the links, using the Open Network Operating System (ONOS) as the SDN controller and the Multi-Generator (MGEN) tool to generate traffic. After these results, this research will use Reinforcement Learning (RL), a subcategory of Artificial Intelligence (AI), to train a RL model in Python using a network of eight interconnected switches.

After the agent is trained, we have made a comparison between RL, LL, and SP, in which we have run a series of files containing flows of different rates. In conclusion, this research will exhibit that, on average, the RL algorithm consistently beats the other two algorithms by 16%, when it comes to reducing the data loss, which will improve the efficiency of the network.

TÍTULO: Encaminamiento basado en Reinforcement Learning en redes SDN

GRADO: Grado en Ingeniería Telemática

AUTOR: Sergi Vera Martínez

DIRECTORA: Anna Agustí Torra

FECHA: 14 de junio del 2023

Resumen

Desde el surgimiento de Internet, ha habido un aumento masivo en el número de usuarios y en la cantidad de información que se transmite. Esto se debe en parte a nuevas tecnologías como la nube y la comunicación en tiempo real. Los clientes ahora demandan una interacción rápida y confiable, sin pérdida de datos ni retrasos indeseados.

Para satisfacer estas necesidades y mantenerse actualizadas, las organizaciones sin fines de lucro han desarrollado una nueva tecnología: las redes definidas por software (SDN). A diferencia de las redes tradicionales, las SDN separan la capa de control de la capa de datos, centralizando todas las decisiones en un controlador y dejando el reenvío del tráfico al dispositivo de red. Esto mejora la operatividad y la agilidad de la red. Sin embargo, persisten problemas relacionados con el enrutamiento del tráfico.

El objetivo de este proyecto es crear un algoritmo capaz de encaminar tráfico teniendo en cuenta el impacto futuro. Para evaluar el mecanismo propuesto, primero obtendremos algunos resultados experimentales usando los pesos de los enlaces por defecto y después basados en la ocupación de los enlaces, utilizando Open Network Operating System (ONOS) como controlador de SDN y la herramienta Multi-Generator (MGEN) para generar tráfico.

Después de obtener estos resultados, se empleará el aprendizaje por Reinforcement Learning (RL), una subcategoría de la Inteligencia Artificial (IA), para entrenar un modelo en Python utilizando una red de ocho conmutadores interconectados.

Después de entrenar el modelo, se realizará una comparación entre RL, Least Loaded (LL) y Shortest Path (SP), utilizando diferentes flujos de tráfico. En resumen, esta investigación demostrará que, en promedio, el algoritmo de RL supera consistentemente a los otros dos algoritmos en un 16% en términos de reducción de pérdida de datos, lo que mejora la eficiencia de la red.

ACKNOWLEDGEMENTS

First and foremost, I dedicate this project to the memory of my father, Pedro Vera Herrera, who always believed in my ability to finish the Telematic Engineering Degree. You are gone, but your belief in me and your eagerness and courage to face every adversity has made this journey possible.

To my professor, Anna Agustí Torra, thank you for all the assistance and knowledge provided from the very beginning, as well as your total willingness.

To the ONOS developers of the Google discussion group, and especially Eder Ollora Zaballa, thank you for all your helpfulness and the time spent answering my doubts about the subject on the mere fact that helping others.

To my family, and especially my mother, thank you for all the support during hard times. You have always kept me motivated and encouraged me to do my best. This would have not been possible either without the support of my girlfriend Laia, who cheered me up to conclude this project.

To my parents and all the people aforementioned, I say: Thank you.

TABLE OF CONTENTS

INTRODUCTION.....	1
CHAPTER 1. SOFTWARE-DEFINED NETWORKING.....	4
1.1. SDN Layers.....	5
1.1.1. Application layer.....	5
1.1.2. Control layer.....	5
1.1.3. Infrastructure layer.....	6
1.2. OpenFlow switch.....	6
1.2.1. OpenFlow Tables.....	8
1.2.2. Group Tables.....	11
1.2.3. Meter Tables.....	12
1.2.4. OpenFlow Channels.....	13
CHAPTER 2. OPEN NETWORK OPERATING SYSTEM.....	14
2.1. Background.....	14
2.2. Architecture.....	14
2.2.1. Application Subsystem.....	16
2.2.3. Flow Rule Subsystem.....	20
2.2.4. Topology Subsystem.....	22
2.2.5. Group Subsystem.....	23
CHAPTER 3. DESIGN AND IMPLEMENTATION OF A ROUTING APPLICATION IN ONOS.....	25
3.1. Creation, installation, and deployment of our custom application.....	25
3.2. Architecture of the routing-app application.....	27
3.2.1. Complementary tools.....	29
3.2.1.1. <i>Monitoring tools</i>	29
3.2.1.2. <i>Network topology tools</i>	30
3.2.1.3. <i>Traffic generation tools</i>	30
3.3. Results using Dijkstra as implemented in ONOS.....	30
3.3.1. Flowcharts of each component of the application.....	30
3.3.1.1. <i>Packet processing</i>	31
3.3.1.2. <i>Topology listener</i>	33
3.3.1.3. <i>Flow rule listener</i>	34
3.3.1.4. <i>Group listener</i>	35
3.3.1.5. <i>CustomLinkWeight Java class</i>	36
3.3.2. Experimental results.....	37
3.4. Results using a custom routing based on link occupancies.....	40
3.4.1. Flowcharts of each component of the application.....	40
3.4.1.1. <i>Packet processing</i>	41
3.4.1.2. <i>Links exceeding group checker</i>	42
3.4.1.3. <i>Topology listener</i>	45
3.4.1.4. <i>Flow rule listener</i>	45

3.4.1.5. CustomLinkWeight Java class.....	46
3.4.2. Experimental results.....	47
3.5. Comparison results between both implementations.....	49
CHAPTER 4. REINFORCEMENT LEARNING.....	52
4.1. Artificial Intelligence and Reinforcement Learning.....	52
4.1.1. Epsilon-Greedy algorithm.....	54
4.1.2. Q-Learning algorithm.....	55
4.2. Definition of the RL in our environment.....	56
4.3. Training of the RL model.....	59
4.3.1. Training parameters.....	59
4.3.2. Training results.....	65
4.4. Comparison between routing algorithms.....	66
4.4.1. Comparison results.....	69
CONCLUSIONS.....	71
REFERENCES.....	73
APPENDIX A. ENVIRONMENT CONFIGURATION.....	76
A.1. Installing Maven (3.6.3) and Karaf Runtime (4.2.9).....	76
A.2. Installing Oracle Java 11.....	76
A.2.1. Making mvn command global.....	76
A.3. Installing ONOS Uguisu (2.4.0).....	77
A.4. Building ONOS using Bazel (1.6.1).....	78
A.4.1. Installing gcc compiler (9.3.0), python2, python3, and pip.....	78
A.4.2. Changing ONOS IP localhost for out private address.....	79
A.5. Installing Mininet (2.2.2).....	81
A.6. Running Mininet and ONOS.....	82
A.7. Installing additional tools.....	82
A.7.1. Installing Wireshark.....	83
A.7.2. Installing IntelliJ IDEA (2019.3.4) and Bazel project configuration.....	83
A.7.3. Installing Jperf.....	92
APPENDIX B. INSTALLATION AND CONFIGURATION OF MONITORING TOOLS. 93	
B.1. Installing InfluxDB.....	93
B.1.1. Configuration process.....	93
B.2. Installing Grafana.....	94
B.2.1. Configuration process.....	94
APPENDIX C. NETWORK TOPOLOGIES, MONITORING SCRIPTS, AND OTHER FILES OF INTEREST.....	98
C.1. Network topologies.....	100
C.1.1. Topology of sections 3.3. and 3.4.....	105
C.1.2. Reinforcement Learning topology.....	108
C.2. Grafana.....	112
C.2.1. Results of section 3.3.2.....	119
C.2.2. Results of section 3.4.2.....	120
C.3. MGEN.....	121
C.3.1. How to generate MGEN flows.....	124

**APPENDIX D. REINFORCEMENT LEARNING TRAINING WITH DIFFERENT
HYPERPARAMETERS..... 129**

LIST OF FIGURES

Fig. 1.1 SDN three-layer architecture.....	5
Fig. 1.2 Logical representation of the SBI in SDN.....	7
Fig. 1.3 Wireshark headers of OSI model of an OF packet.....	7
Fig. 1.4 OpenFlow switch logical representation of the different parts and processes [4].....	8
Fig. 1.5 Representation of the different components of an OpenFlow switch pipeline.....	9
Fig. 1.6 Fields of a rule of an OpenFlow flow entry.....	9
Fig. 1.7 Fields of a group entry in OF.....	11
Fig. 1.8 Fields of a meter band in OF.....	12
Fig. 1.9 Logical representation of the three different OpenFlow channels.....	13
Fig. 2.1 ONOS seven-tier architecture.....	15
Fig. 2.2 Logical representation of the three components in an ONOS subsystem	16
Fig. 2.3 Interaction of the applications via the NBI and the REST API with the manager component.....	17
Fig. 2.4 ONOS representation and adaptation of an OF switch in the context of the application subsystem.....	19
Fig. 2.5 ONOS representation and adaptation of an OF rule in the context of the flow rule subsystem.....	21
Fig. 2.6 ONOS representation and adaptation of a SDN in the context of the topology subsystem.....	22
Fig. 2.7 ONOS representation and adaptation of an OF group in the context of the group subsystem.....	24
Fig. 3.1 List of installed and activated ONOS applications.....	26
Fig. 3.2 Logs in the terminal window running ONOS.....	27
Fig. 3.3 Directory structure of our custom application routing-app.....	27
Fig. 3.4 Directory structure of the topology core folder.....	29
Fig. 3.5 UML diagram of the different threads of the ReactiveForwarding Java class using Dijkstra as implemented in ONOS.....	31
Fig. 3.6 Flowchart of the processing of the packets using Dijkstra as implemented in ONOS.....	32
Fig. 3.7 Flowchart of the topology listener using Dijkstra as implemented in ONOS.....	33
Fig. 3.8 Flowchart of the flow rule listener using Dijkstra as implemented in ONOS.....	34
Fig. 3.9 Flowchart of the group listener using Dijkstra as implemented in ONOS.	35
Fig. 3.10 Flowchart of the link weight decision in the CustomLinkWeight class using Dijkstra as implemented in ONOS.....	36
Fig. 3.11 Visualization in ONOS GUI of the topology used for the experimental	

results for Dijkstra as implemented in ONOS and the implementation using a custom routing based on link occupancies.....	37
Fig. 3.12 BW representation from the perspective of h6 depicted in jPerf using Dijkstra as implemented in ONOS.....	39
Fig. 3.13 Percentage of flows installed per switch over the total using Dijkstra as implemented in ONOS.....	40
Fig. 3.14 UML diagram of the different threads of the ReactiveForwarding Java class using a custom routing based on link occupancies.....	41
Fig. 3.15 Flowchart of the processing of the packets using a custom routing based on link occupancies.....	42
Fig. 3.16 Flowchart of the links exceeding group checker using a custom routing based on link occupancies.....	43
Fig. 3.17 Data rate mirroring along the established path between h1 and h2... 44	
Fig. 3.18 Flowchart of the topology listener using a custom routing based on link occupancies.....	45
Fig. 3.19 Flowchart of the flow rule listener using a custom routing based on link occupancies.....	46
Fig. 3.20 Flowchart of the link weight decision in the CustomLinkWeight class using a custom routing based on link occupancies.....	47
Fig. 3.21 BW representation from the perspective of h6 depicted in jPerf using a custom routing based on link occupancies.....	48
Fig. 3.22 Percentage of flows installed per switch over the total using a custom routing based on link occupancies.....	49
Fig. 3.23 Representation of the allocation problem using conventional routing algorithms.....	51
Fig. 4.1 AI and its different subcategories.....	52
Fig. 4.2 RL agent and environment diagram.....	54
Fig. 4.3 Epsilon-greedy diagram.....	55
Fig. 4.4 Evolution of epsilon value throughout the episodes.....	55
Fig. 4.5 RL topology used to train the agent.....	57
Fig. 4.6 DNN representation of the RL agent.....	58
Fig. 4.7 Representation of the ReLU activation function.....	58
Fig. 4.8 Distribution in time of training flows grouped by rate.....	61
Fig. 4.9 Number of training flows grouped by rate.....	61
Fig. 4.10 Diagram to train the RL agent.....	64
Fig. 4.11 Total episode reward with moving averages throughout the episodes... 65	
Fig. 4.12 Total number of times a path is chosen by the RL agent.....	66
Fig. 4.13 Average link occupancy during the whole training period.....	66
Fig. 4.14 Diagram to adapt RL to the comparison with other routing algorithms... 67	
Fig. 4.15 Diagram to use the LL algorithm for the comparison.....	68
Fig. 4.16 Diagram to use the SP algorithm for the comparison.....	68

Fig. 4.17 Average number of flows allocated for the comparison of the algorithms.....	69
Fig. 4.18 Moving average of 50 comparison files for the comparison of the algorithms.....	69
Fig. 4.19 Total number of times a path is chosen by each comparison algorithm.	70
Fig. 4.20 Average link occupancy during the whole comparison period by each algorithm.....	70
Fig. A.1 Maven installation verification.....	77
Fig. A.2 List of ONOS versions.....	77
Fig. A.3 Bazel installation verification.....	78
Fig. A.4 Bazel's complementary tools verification.....	79
Fig. A.5 Network interfaces.....	80
Fig. A.6 Onos-run-karaf configuration file.....	80
Fig. A.7 List of Mininet versions.....	81
Fig. A.8 ONOS_APPS variable in bash_profile file.....	82
Fig. A.9 Bazel workspace.....	85
Fig. A.10 Copy external option to select the project view.....	86
Fig. A.11 Bazel project view summary.....	86
Fig. A.12 Bazel binary path.....	87
Fig. A.13 IntelliJ settings for ONOS project.....	87
Fig. A.14 ONOS copyright profile for Apache2 license.....	88
Fig. A.15 Mark onos directory as Sources Root.....	88
Fig. A.16 Project view errors in IntelliJ.....	89
Fig. A.17 Add routing-app directory as a Maven project.....	89
Fig. A.18 Proper project view in IntelliJ.....	90
Fig. A.19 Delete the OSGi framework under routing-app module.....	90
Fig. A.20 Set up a new project SDK.....	91
Fig. A.21 Select the Home Directory for JDK.....	91
Fig. A.22 IntelliJ debug configuration.....	92
Fig. B.1 InfluxDB service up and running.....	93
Fig. B.2 Grafana service up and running.....	94
Fig. B.3 Default login screen in Grafana.....	95
Fig. B.4 Add data source in Grafana.....	95
Fig. B.5 Add InfluxDB as data source in Grafana.....	96
Fig. B.6 InfluxDB configuration in Grafana (I).....	96
Fig. B.7 InfluxDB configuration in Grafana (II).....	97
Fig. C.1 Request to upload the full network configuration.....	102
Fig. C.2 Fields from flow_active_entries measurement.....	112
Fig. C.3 Fields from link_occupations measurement.....	112
Fig. C.4 Fields from port_load measurement.....	113
Fig. C.5 Request to get the entire network configuration.....	114

Fig. C.6 Request to get the statistics of a specified device and port.....	115
Fig. C.7 Request to get the sum of active flow entries in a device.....	116
Fig. C.8 Occupancy of the link S1-S2 depicted in Grafana using Dijkstra as implemented in ONOS.....	119
Fig. C.9 Occupancy of the link S1-S3 depicted in Grafana using Dijkstra as implemented in ONOS.....	119
Fig. C.10 Occupancy of the link S1-S4 depicted in Grafana using Dijkstra as implemented in ONOS.....	120
Fig. C.11 Occupancy of the link S1-S2 depicted in Grafana using a custom routing based on link occupancies.....	120
Fig. C.12 Occupancy of the link S1-S3 depicted in Grafana using a custom routing based on link occupancies.....	121
Fig. C.13 Occupancy of the link S1-S4 depicted in Grafana using a custom routing based on link occupancies.....	121
Fig. D.1 Total episode reward with batch size of 64 and replay memory size of 5,000.....	129
Fig. D.2 Total episode reward with replay memory size of 5,000.....	130
Fig. D.3 Total episode reward with batch size of 256, replay memory size of 5,000, and epsilon decay of 0.9975.....	130
Fig. D.4 Total episode reward with batch size of 64 and replay memory size of 1,000,000.....	131
Fig. D.5 Total episode reward with batch size of 64, epsilon decay of 0.995, replay memory size of 1,000,000, and replay start learning size of 50,000.....	131
Fig. D.6 Total episode reward with batch size of 64, epsilon decay of 0.9975, replay memory size of 1,000,000, and replay start learning size of 50,000.....	132
Fig. D.7 Total episode reward with learning rate of 0.0001, batch size of 32, epsilon decay of 0.995, and replay memory size of 1,000,000.....	132
Fig. D.8 Total episode reward with learning rate of 0.0001, batch size of 32, gamma of 0.99, epsilon decay of 0.995, and replay memory size of 200,000.....	133
Fig. D.9 Total episode reward with batch size of 64, epsilon decay of 0.995, replay start learning size of 50,000, and replay memory size of 1,000,000.....	133

LIST OF TABLES

Table. 2.1 Excerpt of an example pom.xml file for an ONOS application.....	18
Table. 3.1 Commands to generate the .oar bundle to install the application.....	26
Table. 3.2 QoS parameters obtained using Dijkstra as implemented in ONOS	38
Table. 3.3 QoS parameters obtained using a custom routing based on link occupancies.....	48
Table. 3.4 Comparison of the QoS parameters obtained using Dijkstra as implemented in ONOS and using a custom routing based on link occupancies	50
Table. 4.1 Q-Table with the Q-value for all the state-action pairs.....	56
Table. 4.2 Excerpt of the JSON file containing the flows to train the RL agent.	60
Table. 4.3 Hyperparameters used to train the RL model.....	62
Table. C.1 pom.xml file of our custom routing-app application.....	98
Table. C.2 JSON structure for the network configuration.....	101
Table. C.3 Python method to upload the network configuration to ONOS.....	102
Table. C.4 Python main method for every topology script.....	104
Table. C.5 Mininet topology for using Dijkstra as implemented in ONOS and using a custom routing based on link occupancies.....	105
Table. C.6 Mininet topology for the RL scenario.....	108
Table. C.7 Python script to monitor the network in Grafana (collector.py).....	116
Table. C.8 Excerpt of the JSON file containing the results of MGEN.....	122
Table. C.9 Python script to generate the results from MGEN log files (parser.py)	123
Table. C.10 Python file to generate the JSON files for the comparison.....	125

ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
ARP	Address Resolution Protocol
BGP	Border Gateway Protocol
BW	Bandwidth
CAPEX	Capital Expenditures
CLI	Command-Line Interface
CPU	Central Processing Unit
CRUD	Create, Read, Update, and Delete
CSV	Comma-separated values
DNN	Deep Neural Network
DNS	Domain Name System
DQN	Deep Q-Learning
DSCP	Differentiated Services Code Point
FIFO	First in, First out
GUI	Graphical User Interface
HTTPS	Hypertext Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISP	Internet Service Providers
JDK	Java Development Kit
JSON	JavaScript Object Notation
LAN	Local Area Networks
LL	Least Loaded
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
MGEN	Multi-Generator
ML	Machine Learning
NAT	Network Address Translation
NBI	NorthBound Interface
NFV	Network Functions Virtualization
OAR	ONOS Application aRchive
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OPEX	Operational Expenditures
OSGi	Open Services Gateway initiative
OSI	Open Systems Interconnection
OVS	Open vSwitch
POM	Project Object Model
QoS	Quality of Service
ReLU	Rectified Linear Unit
REST	Representational State Transfer

RL	Reinforcement Learning
SBI	SouthBound Interface
SDK	Software Development Kit
SDN	Software-Defined Networking
SLA	Service Level Agreement
SP	Shortest Path
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
vBNG	Virtual Broadband Network Gateway
VLAN	Virtual Local Area Network

INTRODUCTION

Everytime we hear about the word “Internet” or it comes to our mind, we usually tend to forget not only how we, as humankind, have managed to build such a powerful tool to communicate with hundreds of millions of individuals across the world, in almost every place of the world we can think of. Ever since the first node-to-node communication from one computer to another was made back in 1965, by the former ARPAnet, the number of people using the Internet has experienced an exponential increase, having already reached the staggering figure of 5.181 million users [1], at the time of writing.

Although at its early stages it was only used for educational and investigation purposes, many users are using lots of different applications in their everyday lives, ranging from amusement tools to work-related solutions. Accompanying this increase, and even more important than that, is the amount of information transmitted in these communications and how important it is for the parties involved to deliver this data in a fast and reliable way, without any data loss or undesired delays.

To meet all these requirements, the network solutions offered by the different Internet Service Providers (ISP) have had to adapt with cutting-edge and groundbreaking technologies, Software-Defined Networking (SDN) being one of the most striking ones. Before SDN was started to be defined circa 2004 by the Internet Engineering Task Force (IETF), even though many different routing protocols had been already put in place, at the end of the day all of them shared the same network architectural model, consisting of having coupled the control plane with the data plane. On the other hand, SDN decouples these two layers, leaving the act of forwarding traffic to the network device and delegating all network decisions to a controller, thus centralizing all the decisions. This idiosyncrasy improves agility, enhances network operability, and makes the network more efficient, among many others.

Even though it might seem the ultimate network technology, it also has its drawbacks when it comes to routing the traffic across the network topology. The behavior of some of these widely used routing algorithms, such as Dijkstra and load balancing technique based on links' occupation, is that they allocate traffic without considering the impact the chosen path could have if future traffic were to be introduced in the network. Traditional load balancers usually split traffic evenly across the different outbound links of the source network device, the main criteria being the current occupancy of the links.

Taking advantage of the fact that we are working with a technology that centralizes the decisions, we have made use of Open Network Operating System (ONOS) as the SDN controller, which has installed a custom routing application coded in Java to install all the flow rules in the switches based on the chosen path. First of all, this project will begin by evaluating the behavior of the Dijkstra algorithm, firstly using the default link weights and then basing these weights on the occupancy of the links. The analysis has been done in a

basic network with just a few switches running OpenFlow to communicate with the controller, using different tools such as iPerf, Multi-Generator (MGEN), and Grafana. After these results and their subsequent comparison, this research will use Artificial Intelligence (AI) to implement a routing solution. More precisely, the controller of the network will make use of Reinforcement Learning (RL) to make routing decisions. With this solution, the controller will not only take into account the rate of the incoming traffic but also it will analyze the future impact of that decision in the event of an increase of data being transmitted. In order to build and configure the scenario, the RL agent has been trained with the programming language Python using a network of eight switches, interconnecting a source and a destination Local Area Networks (LAN), with N servers sending information from one data center to another.

Once the agent is trained, a comparison between RL, Least Loaded (LL), and Shortest Path (SP) has been made, in which we have run a series of files containing flows of varying sizes introduced in the network at different moments in time. This research will show that, on average, the RL model consistently beats the other two routing algorithms by a hefty 16%, when it comes to reducing the data loss. In a real network, this will not only improve the efficiency of the network but also reduce the economic impact of having to install and remove many flow rules in the network devices, as the Central Processing Unit (CPU) load and CPU utilization will be reduced.

The remainder of this research is organized as follows:

- **Chapter 1** describes in detail what SDN is, explaining the transition from traditional networks, in which the control and data layers were not decoupled, to software-defined networks, the different layers there are, as well as a thorough explanation of the OpenFlow protocol, which is the bedrock of the SDN technology.
- **Chapter 2** deals with the ONOS controller, analyzing in detail its architecture. Namely, the system components, the network state, the different types of devices, as well as the different applications it supports.
- **Chapter 3** explains how the custom application in ONOS has been built, in addition to the implementation of the Dijkstra algorithm and the one based on the occupancy of the links to show the problems, aforementioned in this introduction, of using such routing protocols and techniques. In addition, it makes a thorough analysis of the different modules of the customer application built in ONOS to govern the routing decisions and the subsequent installation of the flow rules
- **Chapter 4** is about describing in detail what RL is and the analogy with the scenario we want to solve, including the most important concepts of this disrupting AI technology. Furthermore, it describes how the RL model has been trained with the different network traffic, the results obtained, and the comparison with LL and SP algorithms.
- Finally, the **conclusions** of this research and the results obtained will be given, besides all work that could be implemented in the future to enhance the current research.

Additionally to the main chapters, there has also been included some appendices to further explain in detail topics that are not cover in the main sections, in order not to clutter the explanation making the lecture cumbersome:

- **Appendix A** explains environment configuration in detail, covering the installation of Mininet, ONOS, Python, and all the software required to do the research.
- **Appendix B** covers the installation and configuration of the monitoring tools, which are InfluxDB and Grafana.
- **Appendix C** shows the different monitoring scripts, configuration files, and other files of interest for Python, Grafana, and MGEN.
- **Appendix D** contains some of the training results of the RL agent using different hyperparameters values, to exhibit the difficulties encountered when training this kind of model.

CHAPTER 1. SOFTWARE-DEFINED NETWORKING

Software Defined Networks (SDNs) are starting to play a very important role in today's society, with many enterprises around the world using this solution as their preferred technology to run their operations, with expectations to grow even faster in the near future [2]. But how and why has it achieved such a tremendous use? The idea of decoupling the control layer from the application layer had already started in 2004, when, albeit had not been defined yet as SDN, the IETF began releasing their first publication about this topic named "Forwarding and Control Element Separation". It was not until 2008 that the first protocol intended specifically for SDN networks was defined, called OpenFlow, which eventually exposed an Application Programming Interface (API) to be used by the different network devices to communicate with each other and with the controller. Finally, in 2011, the Open Networking Foundation (ONF) was created as a non-profit organization with the aim of promoting the use of SDN and the OpenFlow protocol as well as all related technologies, taking part into different projects to speed and enhance the use of this technology. Let's get into the details to see all components and characteristics that make SDN what it is nowadays.

As has been aforementioned, this technology separates completely the application layer from the control layer, giving full programmability and operability to the controller, making all infrastructure devices somewhat agnostic, detaching all the network logic from them. The benefits of using SDN are plentiful and are not only limited to having a centralized management that can accelerate the different network decisions. It also has a tremendous impact on cost reduction, when it comes to Capital Expenditures (CAPEX) and Operational Expenditures (OPEX), as network administrators need to spend less on new devices and maintaining the existing ones. In relation to (Quality of Service) QoS, adopting this technology will increase the bandwidth, and reduce the packet loss and the delay in the network.

When it comes to the content of this chapter, we are going to thoroughly explain the different layers we can find in SDN from a logical viewpoint. Moreover, if it is true that there are nowadays different protocols that the network devices can use to communicate with the controller and vice versa, our switches will use OpenFlow. We will see the most relevant information about them for this project, to better understand later in section two their connection with ONOS, the controller.

As shown in Fig 1.1, the infrastructure layer communicates with the SDN controller by means of the OpenFlow protocol, whilst the control layer uses an API to talk with the application layer. Let's see thoroughly the different layers:

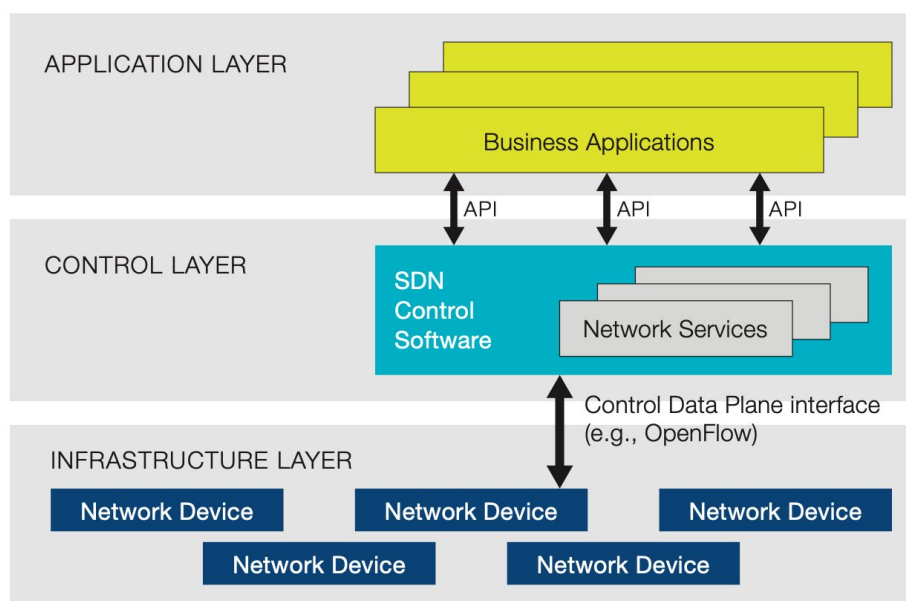


Fig. 1.1 SDN three-layer architecture

1.1. SDN Layers

1.1.1. Application layer

The application layer is composed of programs, also known as business applications, that can communicate programmatically to the SDN controller the intention to introduce or modify the already existing features to amend the network behavior, not only in terms of traffic forwarding but also security-related. Thus, having always a global view of the status of the network provided by the controller, gives these applications the possibility to be always managing the network. As depicted in Fig 1.1, the way to convey all this information from the application plane to the control plane is via an API, an interface that receives the name of NorthBound Interface (NBI). In this layer, we might find multiple network orchestration services that take care of managing and monitoring the communication emitted by each device using Link Layer Discovery Protocol (LLDP) or other softwares behaving as a proxy for Address Resolution Protocol (ARP) mapping Internet Protocol (IP) addresses with Media Access Control (MAC) addresses, among many others. As we will see in next chapters, we will install the AI routing application in this layer.

1.1.2. Control layer

The control layer, also known as control plane, is the backbone of the SDN layer distribution architecture, and it is where the controller resides. In some cases, if the network structure is extremely complex or certain Service Level Agreements (SLA) have to be met, there might be multiple controllers operating concurrently between them, although it is not the standard as centralization fades away a bit. The control plane acts as the bridge between the application

layer and the infrastructure layer, translating the messages and information received via the NBI and communicating them back to the network devices via the SouthBound Interface (SBI). This communication flow is not only limited to this direction, as this layer also conveys information from the SBI to the NBI if needed for the network to operate adequately. The preferred protocol used by the SBI to communicate with the devices is OpenFlow, which is considered the best fit for switches.

1.1.3. Infrastructure layer

This last layer, also known as data plane, is made up by the different network devices that conform the network. The most common forwarding devices that can be found are switches and routers, being the first ones the favored option as the OpenFlow is in constant evolution to enhance the communication with switches. These devices perform the actions sent by the controller including forwarding a packet to the specific port or dropping it if the rules are not met. Apart from this, they periodically send vital information for the control plane such as the number of active ports or who their neighbors are at that given moment. Likewise, they also gather information to provide the controller with statistics, namely but not limited to port load, information about the flows received, and so on and so forth.

1.2. OpenFlow switch

The OpenFlow protocol is a standard in SDN architecture managed by the ONF, and is widely supported and used across many different switches, including Radisys, HP, Cisco, among many others. The precursor of this protocol was Ethane in 2007, which introduced centralized and reactive flow management, whereas OpenFlow was initially conceived in 2008 and did not see the light until 2009. Since that moment until nowadays, it has been adopted by many controllers, such as OpenDaylight, Ryu, and ONOS. The functioning of this protocol resides in the idea of moving the control and management of the network out of the device in the infrastructure layer, into the software that runs inside the controller so it can be centrally, openly, and locally managed. Before explaining thoroughly the characteristics of this protocol, it is important to reinforce the fact that the network devices, mostly switches, only operate in the layer 2 of the Open Systems Interconnection (OSI) model [3]. As they operate in the link layer, the role of these devices is just to forward Ethernet frames from one switch to another, based on the forwarding rules they have installed. As we will see below, the OpenFlow controller is in charge of establishing a communication, known as OpenFlow channel, with every switch in the network to forward all the traffic rules as well giving information about its hardware properties. It is worth knowing that the information exchanged in this channel will never affect the rest of the network, meaning that other switches would not receive any packet exposing this information. The Fig 1.2 provides a general perspective of the communication between OpenFlow switches and the OpenFlow controller using the SouthBound API.

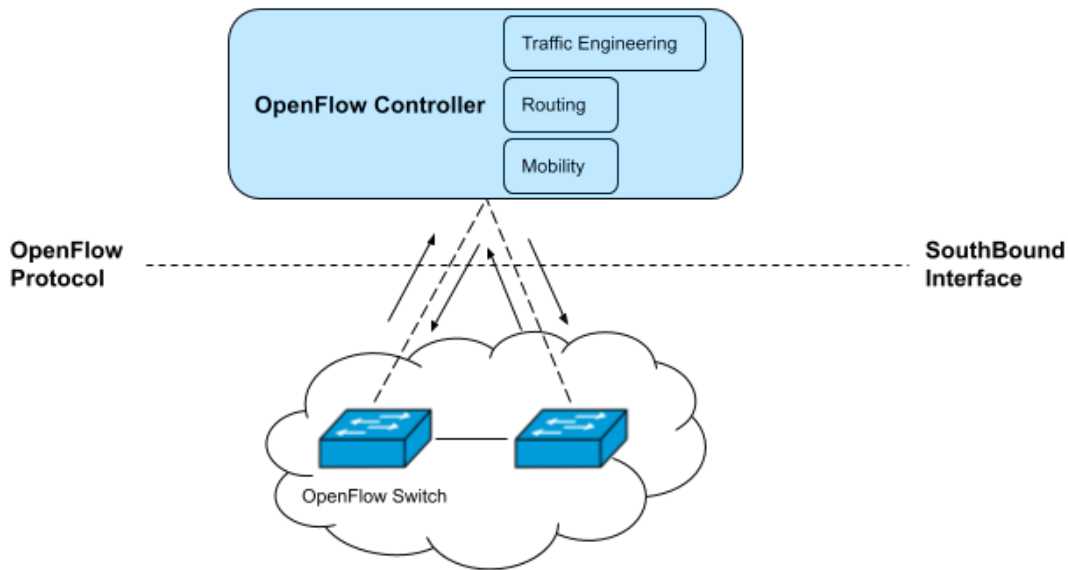


Fig. 1.2 Logical representation of the SBI in SDN

One of the most striking advantages of using this protocol is the flexibility it gives to the different vendors. Even though they might have different interfaces and scripting languages for their Operating System, OpenFlow creates some kind of an interface to abstract this communication and create one that is shareable, so the devices can be managed remotely using a single protocol. When it comes to the layer of the OSI model in which it operates, it is layered on top of Transmission Control Protocol (TCP) using the port 6653, which is the one the controller is listening to, in order to establish a connection and send packets. In Fig 1.3 we can see how, from top to bottom, the whole OpenFlow packet is built, with physical bytes, the ethernet frame which is shown in this picture as “Linux cooked capture”, the IP and TCP headers, and finally the OpenFlow 1.0 packet data.

```

117 1.326760378  localhost      localhost      OpenFlow      292  CS6 Type: OFPT_FEATURES_REPLY
> Frame 117: 292 bytes on wire (2336 bits), 292 bytes captured (2336 bits) on interface 0
> Linux cooked capture
> Internet Protocol Version 4, Src: localhost (127.0.0.1), Dst: localhost (127.0.0.1)
> Transmission Control Protocol, Src Port: 45474 (45474), Dst Port: openflow (6653), Seq: 9, Ack: 29, Len: 224
▼ OpenFlow 1.0
  .000 0001 = Version: 1.0 (0x01)
  Type: OFPT_FEATURES_REPLY (6)
  Length: 224
  Transaction ID: 2736994518
  > Datapath unique ID: 0x0000000000000003
    n_buffers: 0
    n_tables: 254
    Pad: 000000
  > capabilities: 0x000000c7
  > actions: 0x00000fff
  > Port data 1
  > Port data 2
  > Port data 3
  > Port data 4

```

Fig. 1.3 Wireshark headers of OSI model of an OF packet

In the next subsections we are going to see how the pipeline of an OpenFlow switch works. Fig 1.4. depicts at a glance the logical schema of this kind of device, with a total of five main components. The *datapath* is composed of the **group** and **meter** tables for common actions and statistics affairs. Falling under the datapath, we find the *pipeline*, which is composed of the input and output **ports** and the **flow tables** that govern and store the different flow rules. Last but not least, there are the different **control channels** a switch can establish with the controller.

As we are only interested in explaining in detail the parts we will use and, hence, adapt to our project, of the list aforementioned ports is the only part that does not fall under the scope of this project. In broad strokes, there are three different types of ports: the **physical** ones correspond to the hardware interface of the switch; the **logical** ones do not directly correspond to a hardware interface, and they are usually used to differentiate them from the ones assigned to protocols such as Hypertext Transfer Protocol Secure (HTTPS), Domain Name System (DNS), and so on; finally, the so-called **reserved** ports are those only intended to be used for OpenFlow-related actions, like forwarding a packet to a neighbor or send a packet to the controller.

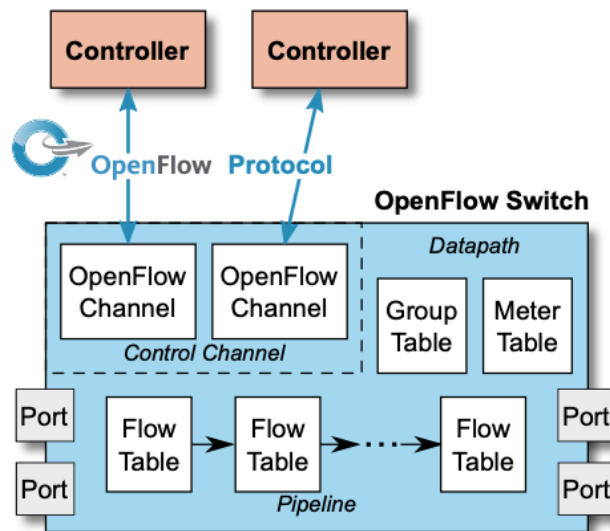


Fig. 1.4 OpenFlow switch logical representation of the different parts and processes [4]

1.2.1. OpenFlow Tables

The pipeline is the most important part of an OpenFlow switch. It is the bedrock of the packet processing, and is composed of a set of flow tables connected one after the other forming a chain of tables. The switch must have at least one table up to a total of 256 but this figure might vary to the upside depending on the version and the capacity of the switch. Although we are not going to cover

this part, it is worth mentioning that some switches have two pipelines; the ingress and the egress pipeline, to also perform decisions before the packet is sent through the output port. Inside of each flow table, we can find what is known as a flow rule or flow entry. In plain terms, these are the instructions the switch uses to take the necessary actions based on the incoming traffic. As we have seen at the beginning of this chapter, in SDN all the decisions will be made by the controller, so it will be the one sending the flow rules to install, update or delete to the required switch. The Fig 1.5 depicts the different components of a pipeline from a logical point of view.

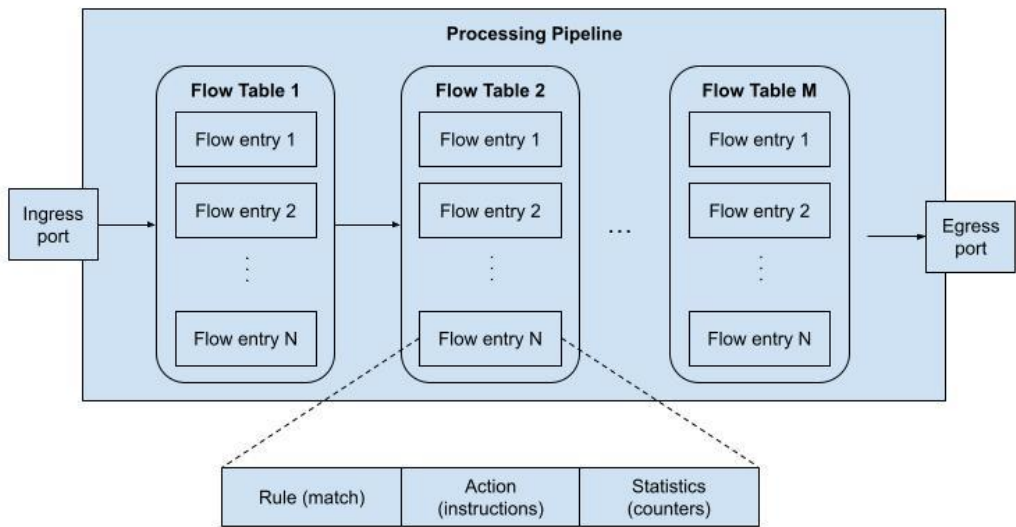


Fig. 1.5 Representation of the different components of an OpenFlow switch pipeline

Having seen the different components in broad terms, let's now see in detail how a flow entry is structured and the various elements we can find inside. First and foremost we find what is known as the rule matching. Every time the switch receives a packet, the first operation it performs is to extract all the fields from the incoming packet, including L2, L3, and L4 labels, being the majority of the time Ethernet, IPv4, and TCP fields, respectively. All these fields are shown in Fig 1.6 below.

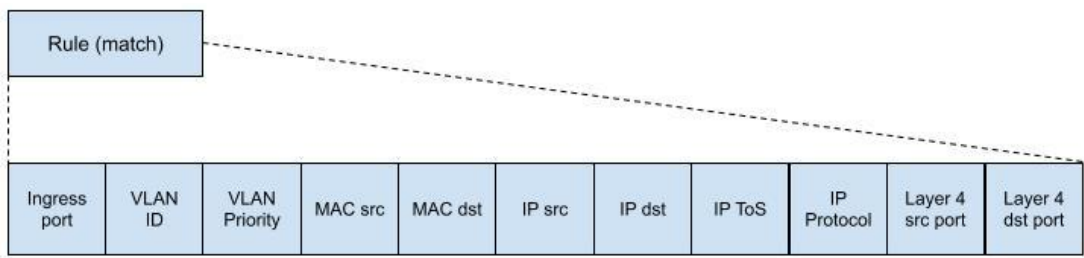


Fig. 1.6 Fields of a rule of an OpenFlow flow entry

Once the switch has extracted and processed all these fields, it will be able to perform the matching across all the different flow entries in the first flow table of the pipeline. In order for a packet to be matched, all values without exception must coincide, and the first flow entry that matches the criteria will be chosen.

Upon performing the matching and selecting the flow entry, now it is time for the switch to perform the action or actions specified in the entry, also known as instructions. In the following list we will see briefly the different required types:

- The **Output** action is the most common one and the most important, as it states the physical, logical, or reserved egress port where the packet will be sent through. For this case, the most typical situations will include switching (just sending a packet through an egress port), routing (when the matching is only based on L3 fields), or Virtual Local Area Network (VLAN) switching (when the matching is based only on L2 fields), among others.
- The **Drop** action is not performed explicitly when it is only defined. For instance, for the situations in which no actions are set or no flow entry has been matched, the packet will be automatically dropped. However, when the action is set, the most common use case is to perform a firewall action, typically based on L2 and L3 fields.
- For the **Group** action, the switch processes the incoming packet through a specified group. This will be seen in detail in section 1.2.3, but a group can trigger a set of actions affecting different flows.

For the optional actions, these are the useful ones to know about for this project:

- As for the **Meter** actions, if the switch version supports meters, the packet will be processed using a meter. It will be seen in section 1.2.4, but in rough outlines it is defined what to do in case a packet goes beyond a defined limit parameter.
- The **Set-Queue** action is very useful for QoS purposes, as the switch sends the packet to the specified queue of the selected egress port.
- Finally, for the **Goto-Table** action, if the flow entry matches, the switch will move the packet to the next flow table to look up in the processing pipeline.

Lastly, the third main block are the counters, also known as statistics. These are updated every time a packet is processed by a specific flow table, regardless of whether the packet will eventually be subjected to any action or it will be dropped. Some other scenarios might include packets that are sent straight to another flow table or even when the port of the switch is down. In all these cases the counters will also need to be updated.

There are not only counters for tables but also for specific ports, flow entries, queues, groups, and meters. The required counters collect statistics about the number of active flow entries in a specific table, how long an entry has been installed (in seconds), the number of received and transmitted packets through

a specific port, among many others that, all together, make these statistics extremely important for QoS purposes.

1.2.2. Group Tables

As we have seen in the previous section, in some cases the action taken by the switch will be to submit the packet to be processed by a group. In OpenFlow, a group table is nothing but a list of group entries constituting a more complex decision than a simple output port and that could not be performed or it would be very difficult if it were not for these group tables. Unlike flow tables, groups do not have the ability to send packets to another flow table to perform chaining, nor do they perform matching. Taking into account this, let's understand better what is inside a group entry and the different types there are.

As can be seen in Fig 1.7, a group entry is composed by an identifier, the specific type, counters for statistics, and the most important part: the **bucket**. Each individual action is referred to as a bucket and its behavior will be determined by the group type defined.

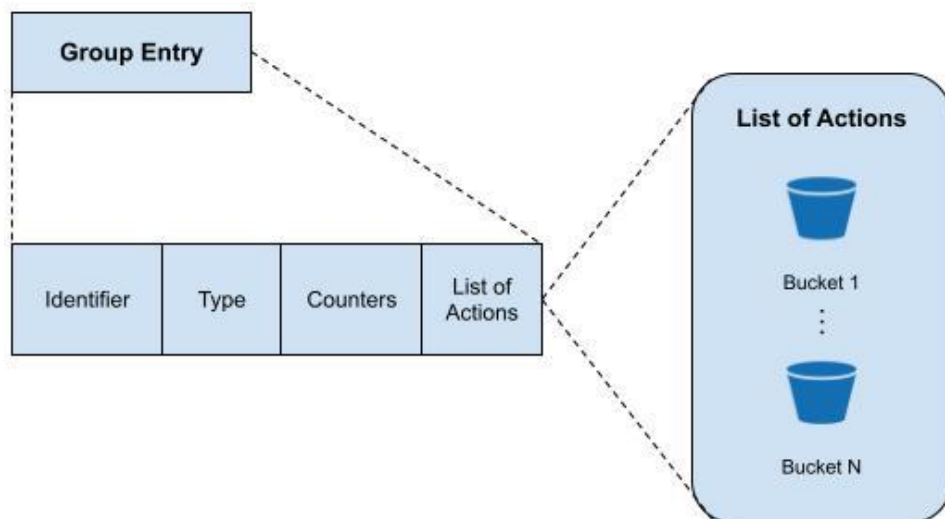


Fig. 1.7 Fields of a group entry in OF

Next, the four different types of groups are explained. From this list, the first two are mandatory in every version of the switch whilst the last two are optional.

- The **ALL** group is the most basic of all and, for every incoming packet, it will duplicate it to be handled independently by every bucket in the entry. This group type is commonly used for multicasting and broadcasting purposes, as a normal switch would do in a network.
- The **INDIRECT** group only contains a single bucket that contains a set of actions. Even though at first glance might not be very useful, it is really worth it when many flow entries, though having different matching criteria, share the same action/s. For these cases, it is better to

encapsulate all these actions into a single group, which, at the same time, will reduce the memory usage in the switch.

- The **Fast-Failover** group is meant to behave as a workaround to surmount ports and, consequently, link failures. Similar to the ALL group, this type will have a set of buckets with a list of actions but with the addition of watchers, that will continually monitor the state of a port or a group to know if it is up and running. In case the port or group is down, this bucket will become inoperative and the first lived bucket will be chosen. The most striking advantage of this type, albeit not available in all switches, is that having these backup actions in case a port fails will save many enquiries to the controller.
- Finally, the **SELECT** group will have a set of buckets with a particular weight assigned to each of them. This weight can be defined by the user or simple round robin. The ultimate objective of this type is to serve as load balancer for SDN.

1.2.3. Meter Tables

The last tables we are going to cover in this chapter about OpenFlow are the so-called meter tables. This table is composed of a list of meter entries, which has as components an identifier, meter bands, and counters (see Fig 1.8). The applicabilities and use cases of a meter entry might be varied and of different complexity but all of them have in common the need to offer a better service implementing rate-limiting to improve QoS metrics.

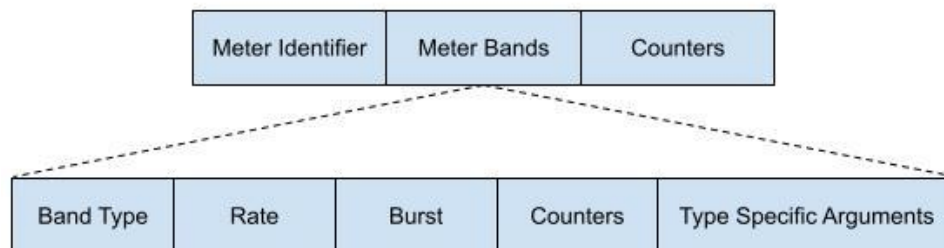


Fig. 1.8 Fields of a meter band in OF

Meters, like groups, are attached directly to a flow entry, meaning that the action will be to point to a specific group with that identifier. The core of the group entry is what is known as bands, where each one specifies the maximum rate or the threshold and the action to take if thereof is reached. As it happened for group entries, the meter band is not subjected to only one but a list of them can be defined. By default, the mentioned target or maximum rate is 0, which means that all the packets will go through without applying a specific action. Let's see now the different fields of a meter band:

- The **Band Type** is an optional field and defines the action taken for this packet. The most common practice is to drop the packet when it reaches the target rate, though more complex traffic classification can be applied by changing the Differentiated Services Code Point (DSCP) value [5].

- The **Rate** defines the target rate value for that band or, in other words, the lowest rate at which the band will apply.
- The **Burst** adds more granularity to the band, in the sense that apart from setting a rate it can also be set a length of the packet.
- The **Counters** serve as the statistics for the bands, with all the values updated every time a packet is processed by a band.
- Finally, the **Type Specific Arguments** field specifies extra arguments for other band types.

1.2.4. OpenFlow Channels

To conclude this chapter, we will define what is an OpenFlow channel and the different messages exchanged through them. This channel is an interface connecting the OpenFlow switch with the controller, and is where all the messages are exchanged, namely configuration, management, events, data, and so on. There are three types of messages:

- The **Controller-to-Switch** are the ones initiated by the controller and not always conveys a response back from the switch. As an example, in these messages the controller may request all the available features of a specific switch (Features in Fig 1.9), or send a packet containing the list of actions to be executed (Packet-out in Fig 1.9), among many others.
- The **Asynchronous** messages are not requested to be sent by the controller. Is the switch who decides to send these packets to inform the controller about the removal of a flow entry (Flow-removed from Fig 1.9), or to inform about a change on a port.
- Lastly, the **Symmetric** messages are sent in either direction and without any request. Some examples can be the message exchanged when a switch turns on for the first time (Hello in Fig 1.9) or when the switch wants to inform the controller about a request that went wrong (Error in Fig 1.9).

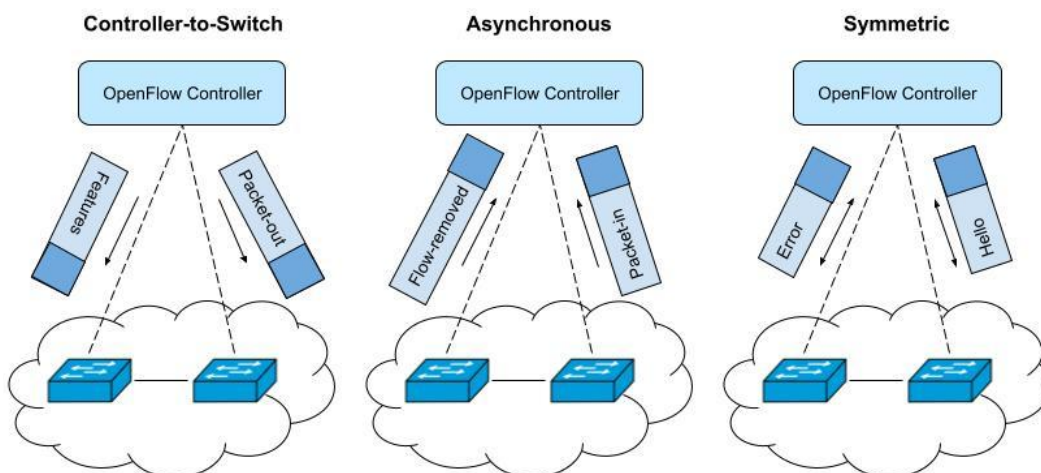


Fig. 1.9 Logical representation of the three different OpenFlow channels

CHAPTER 2. OPEN NETWORK OPERATING SYSTEM

Having already seen the basics of a SDN architecture and the functioning and behavior of the OpenFlow protocol, now it is time to delve into the controller to understand the backbone of these types of networks. Every controller needs to have a proper network programmability and an easy to access but sophisticated API, through which the application can precisely tell the controller what to do. Out in the market, if it is true that there are a plethora of controllers, most of them open source, only a few of them have made a name for themselves and offer the best possible SDN controller. Namely, NOX [6], POX [7], Floodlight [8], OpenDaylight [9], ONOS [10], Trema [11], and so on and so forth. For this project, we have chosen ONOS as our preferred controller given its history and all its different range of possibilities. Let's see first a bit of background about this controller.

2.1. Background

The trajectory of ONOS began in 2012 and its name was coined soon thereafter, but it was not until 2014 and 2015 that the software actually gained lots of popularity with AT&T and Linux Foundation joining the project. The reasons to choose this controller instead of the others aforementioned are varied but could be abridged in three criteria: firstly, due to its popularity and large adoption all the information is quite detailed and there are many forums to solve all kinds of doubts [12]; secondly, it is a controller specifically designed to be performant, providing all the necessary tools to support many devices in huge networks all with a great Graphical User Interface (GUI), Command-Line Interface (CLI) and a complete API; lastly, it is versatile in the sense that it not only provides and has already installed many different applications to be used, but it also allows the customers to create their own applications.

When it comes to the versioning, we have installed the version 2.4.0 known as Uguisu and the software Bazel to build and run ONOS (see Appendix A, section 3 and 4 for more information about the installation). This version was released in late 2020, so it is already stable enough to be used.

2.2. Architecture

After introducing the ONOS controller and justifying its use for this project, let's now describe in detail its architecture. In Fig 2.1, we can see the seven different tiers that this controller is composed of. If you pay close attention, you will see that its structure is the same as the one defined for SDN in Fig 1.1, having the infrastructure, control, and application layers decoupled from one another. Among the different tiers, let's define the one that stands out the most and is the bedrock of the architecture. The **Core** one contains what ONOS calls subsystems, and these ones are accessed and controlled through the NorthBound API, using either the GUI or CLI provided -we will see this

thoroughly in the next sections of this chapter-. A subsystem is a logical aggregation of a different set of services, standing out device, host, link, topology, path, and flow. All of them contain different services composed of multiple components that are in charge of managing infrastructure devices, infrastructure links or topologies depending on the subsystem.

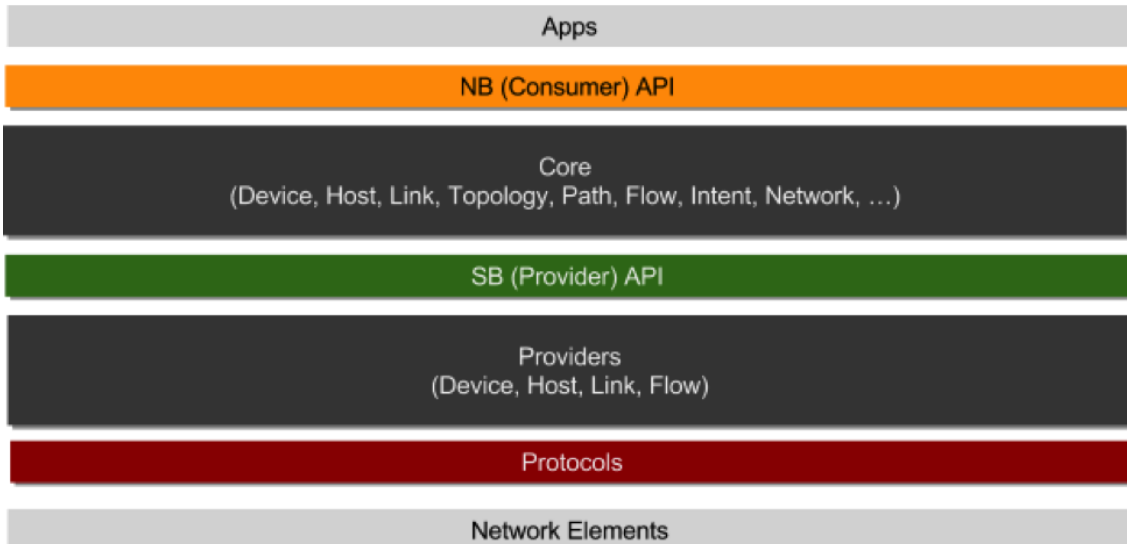


Fig. 2.1 ONOS seven-tier architecture

Before taking a look at the different subsystems mentioned above, let's analyze in more detail how they are structured. All subsystems belong to one of the three components defined in the ONOS paradigm, which are the application, the manager, and the provider (see Fig 2.2 for a visual representation).

- The **Application** component is the uppermost one and it consumes all the information sent by the manager component through the NorthBound API. The list of applications might be endless, from monitoring different aspects of the network to establishing paths based on different network data. Moreover, this is where third-party applications will be located, consuming the API offered by ONOS. For reference, this will be the component in which our routing application will be located.
- The **Manager** component is the core one as it gathers the information sent by providers and presents it to the applications, thus acting as a bridge. It has four main services:
 - The **AdminService**, as its name suggests, receives administrative orders and applies them into the network.
 - The **Service** is the NorthBound API that applications use to get information from the network.
 - The **ProviderService** is the one to keep alive the communication and exchange messages with the provider.
 - Finally, the **ProviderRegistry** is responsible for registering and unregistering in the network the different devices once they are added, removed or set as inactive.

- The **Provider** is the lowest component of ONOS, as protocols and network elements are already part of the infrastructure layer. Providers have the responsibility of, on one hand, communicating with the core subsystems and, on the other hand, communicating with the network devices via the SBI. It is very important to bear in mind that these three components shown in Fig 2.2 are from the controller perspective. This means that the SBI is not using OpenFlow protocol and the provider component does not make reference to the OpenFlow physical switch. As we will see in the next subsections, this is always from the controller's perspective, and how ONOS maps physical components of the network to their logic.

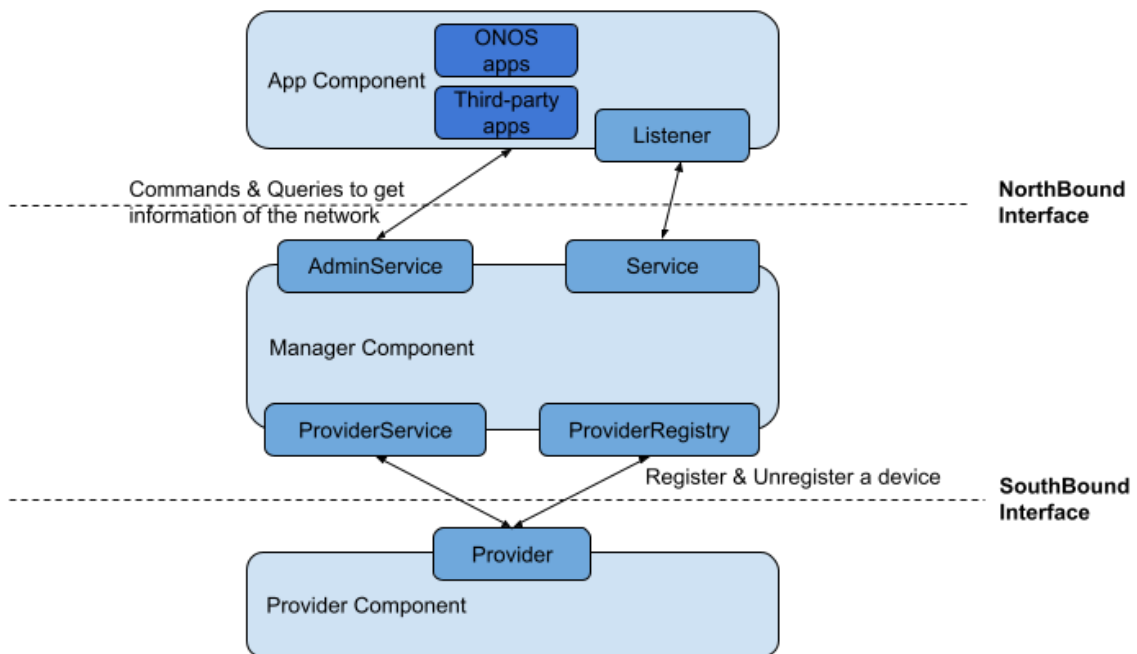


Fig. 2.2 Logical representation of the three components in an ONOS subsystem

2.2.1. Application Subsystem

Having seen the architecture of ONOS, now it is time to analyze in detail the most important subsystems that we will use in the project, starting from the core one: the application subsystem. It is in charge of delivering and managing the installation and subsequent initialization of each application in the ONOS cluster, and, if applicable, delivering these applications to multiple ONOS instances called clusters. In this project we will only work with one instance but this proves the power of this controller. As can be seen in Fig 2.3, every application managed by this subsystem interacts with the northbound API through the Java programming language or the Representational State Transfer (REST) interface, having an easy and fast installation. Applications can interact with the network and tell the controller to create, remove or update network elements including flows, devices, hosts, links, and so on. Some examples of different applications are the SDN-IP peering - very useful for treating the

network as a Border Gateway Protocol (BGP) autonomous system -, video streaming or IPTV to transmit from one sender to multiple receivers, or a Virtual Broadband Network Gateway (vBNG) to provide connectivity between a private host and the Internet. Although there are dozens of applications available, by default ONOS only has installed the following: *hostprovider*, *drivers*, *optical-model*, *openflow-base*, *lldprovider*, *gui2*, *openflow*, and *proxyarp*. Only with this set of applications can we perform the basic communication using Openflow in SDN between two devices in the network.

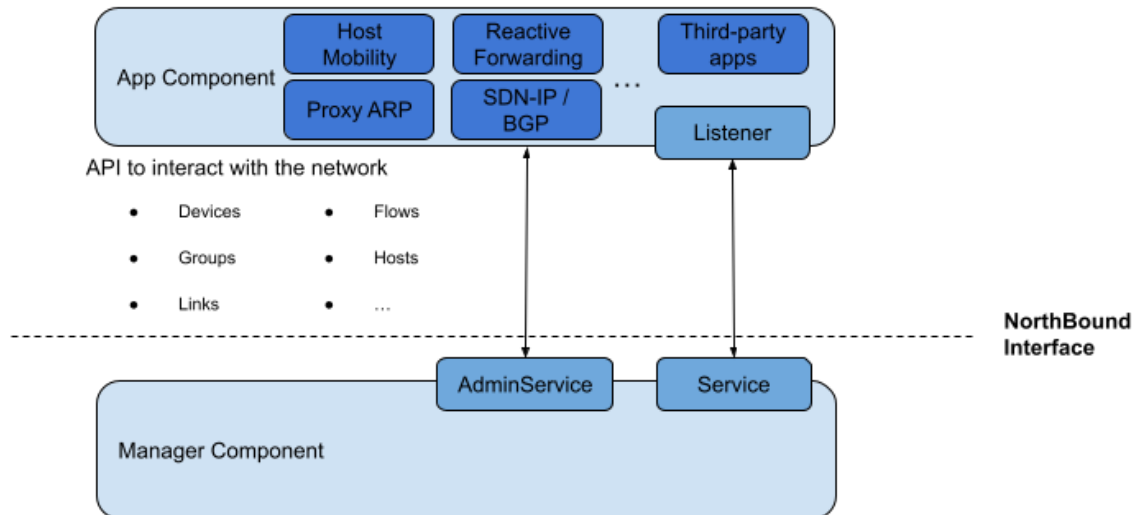


Fig. 2.3 Interaction of the applications via the NBI and the REST API with the manager component

To understand in rough outlines how the installation of an application works, they are built using the Apache Karaf, which is an Open Service Gateway Initiative (OSGi) container. OSGi significantly reduces complexity of Java systems and, as it is modularized, the code is easier to develop. The way to define the package and all the information of our application is by creating an ONOS Application aRchive file (OAR), which has defined a *pom.xml* file.

To have a general understanding of what eXtensible Markup Language (XML) is, let's explain which software uses XML. First of all, XML is a language that allows developers to specify information about that application, all following a human-readable syntax. On the other hand, a custom application created in ONOS will use Maven, a software intended for managing Java applications. Therefore, inside the *pom.xml* file we will define important values such as the *groupId*¹ to which the application belongs, the *artifactId*², the version, and specific properties of the application such as the name, the title, the Uniform Resource Locator (URL) if any, etc, as can be seen in Table 2.1.

¹ In this case, the group ID is the package of the application. In Java, a package groups related classes, as a normal folder does in a file directory.

² In Maven, an artifact is an element that an application can consume, and is defined by a *groupId*, *artifactId*, version, and packaging.

Table. 2.1 Excerpt of an example pom.xml file for an ONOS application

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.onosproject</groupId>
    <artifactId>onos-dependencies</artifactId>
    <version>2.4.0</version>
  </parent>

  <groupId>org.myapp.app</groupId>
  <artifactId>myapp</artifactId>
  <version>version-number</version>
  <packaging>bundle</packaging>

  <description>ONOS OSGi bundle archetype</description>
  <url>http://onosproject.org</url>

  <properties>
    <onos.app.name>myapp-name</onos.app.name>
    <onos.app.title>myapp-title</onos.app.title>
    <onos.app.origin>Sergio Vera-UPC</onos.app.origin>
    <onos.app.category>default</onos.app.category>
    <onos.app.url>http://onosproject.org</onos.app.url>
  </properties>
  ...
</project>

```

Finally, and as has been explained in the architecture, thanks that ONOS applications provide channels and interfaces to run different commands and perform a set of operations, we can use the REST API to install and activate them, running the commands *onos-app install pox.xml* and *onos-app activate <app-name>*.

2.2.2. Device Subsystem

Once we have defined what an application subsystem is and how it is integrated with the controller, we will analyze one of the simplest but more important parts of the network: the device. As in every software application that is run by a programming language, it is required to create a mapping or a logical representation of the physical component we are working with; and in that regard, ONOS is no different. Fig 2.4 depicts the mapping of the different parts and aspects of an OF switch to Java classes. As for the “Switch Representation” in the image, it is not the physical switch but how ONOS translates it into Java to be able to work with it.

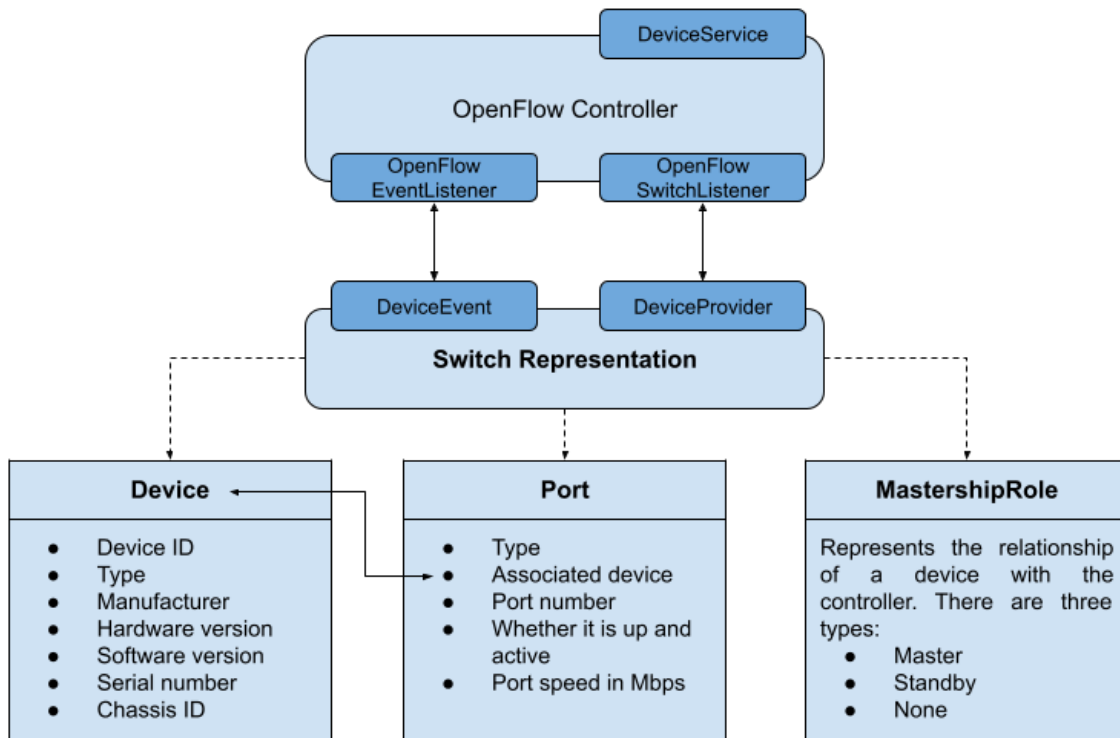


Fig. 2.4 ONOS representation and adaptation of an OF switch in the context of the application subsystem

As we can see, ONOS uses three interfaces to define logically what they call a *DeviceElement*, which in our project is an **OF switch**. The term of interface in programming is an abstraction of an entity from which many objects can inherit [13].

- The **Device** interface is the representation of a network infrastructure device. ONOS supports not only switches but routers, firewalls, fiber switches, servers, and so on, as supported SDN devices. Each device has an ID, which is defined as a text and as a Uniform Resource Identifier (URI). Apart from this, there is extra information that might be useful for external applications to know, such as the manufacturer, software and hardware versions, etc.
- The **Port** interface is the abstraction of a network port. ONOS defines different types, including copper, fiber, and virtual. In order to relate this port with its corresponding device, the port has a method that establishes a linkage with the device. In addition, it stores information like the port number, the status, and the port speed in Mbps.
- Finally, the **MastershipRole** interface defines the relationship between the controller and a particular device. For our project, and all the devices in the network, this relationship will be of type master, meaning that the controller instance will be the master to that particular device.

When it comes to the architecture of the subsystem, the device has defined the provider, called *DeviceProvider*, as well as the *DeviceEvent*, which are like an internal bridge that ONOS uses to communicate the logical device with the manager component that contains the listeners, being *OpenFlowSwitchListener* and *OpenFlowEventListener*. The first one listens to OF messages sent by the physical OpenFlow switch, such as switch and link addition or removal, and sends this information to the *DeviceEvent*. On the other hand, the flow event listener sends the information received to the provider about different open flow message events, as it is the case for any update regarding the different flow rules installed or to be installed.

2.2.3. Flow Rule Subsystem

The next subsystem that is very important to comprehend how it works is the flow rule one, which is responsible for managing the flow rules and installing or removing them in the corresponding devices. A peculiarity of this subsystem is that the original flow rules are actually generated first in the controller, and then the copy of these flow rules is sent to the corresponding devices. They are installed using the *FlowRuleService*, as can be seen in Fig 2.5, and they can only exist in the system in one of the following states:

- **PENDING_ADD** means that the subsystem has received the request from the application to install a flow rule via the *FlowRuleService* but it has not detected yet any new flow being added to the switch. For this, the *FlowRuleProvider* will report any change to the *FlowRuleListener*.
- **ADDED** is when the subsystem receives the notification from the provider that the new flow is already installed in the switch.
- **PENDING_REMOVE** is the other way around of the **PENDING_ADD**. In this case, the service receives a request to remove an already installed flow entry but the listener has not detected any removal yet.
- The status of the flow is set to **REMOVED** when the provider notifies the subsystem about its removal, to soon thereafter remove it definitely from what ONOS called a store, which manages the inventory of flow rules in this case.
- Lastly, **FAILED** points out that something went wrong when adding or removing a flow rule.

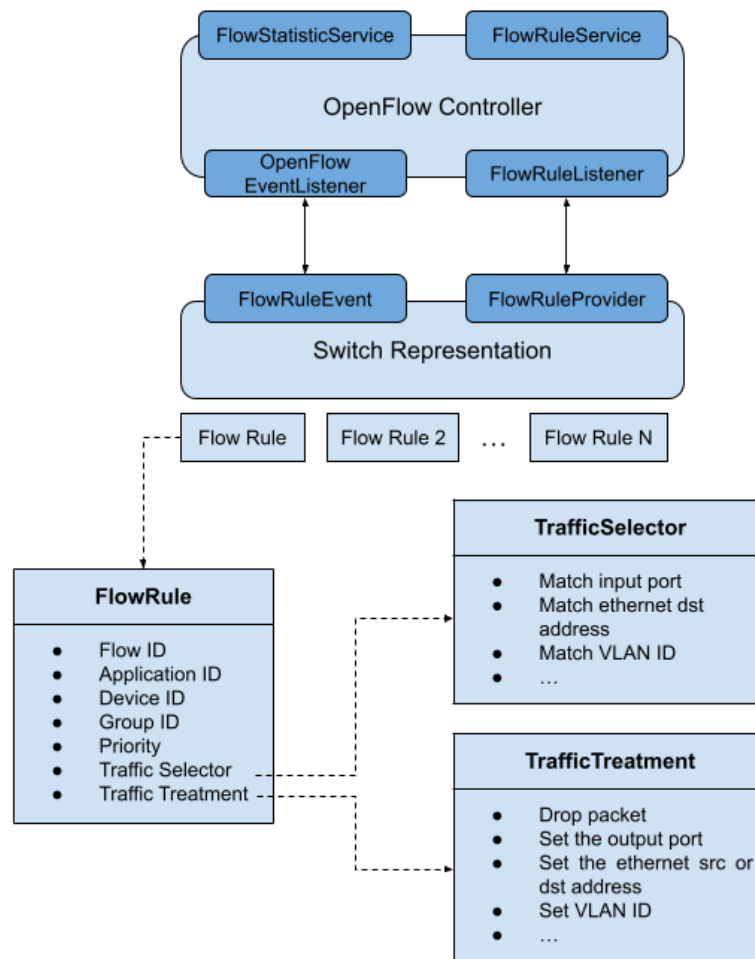


Fig. 2.5 ONOS representation and adaptation of an OF rule in the context of the flow rule subsystem

In the above picture we can see the mapping ONOS to represent a flow rule, as we saw in the previous section for the device. The **FlowRule** interface represents the actual flow installed in the device. It is identified by a number (Flow ID) and is linked to the device through its URI (Device ID). Although we are only using one application, in case there were flows installed in different applications, one could get the application ID in which is installed. As we saw in Fig 1.6 when defining the flow entry, the matching criteria is represented in ONOS by the **TrafficSelector** interface and the actions to take by the **TrafficTreatment**. The selector sets the input port the packet came from, the Ethernet destination address, the VLAN ID, among many others. Likewise, the treatment defines what to do with that packet if it matches the criteria. The list of options in ONOS is large but it can be dropped, set a VLAN ID, set a new Ethernet destination address, etc. To see all the possible values to set for the selector and the treatment please refer to [14] and [15], respectively.

Finally, an external application can obtain all individual flow statistics through the *FlowStatisticService*. Thanks to this service, values and counters such as total bytes sent, total time since this flow was installed, and many others, can be obtained.

2.2.4. Topology Subsystem

The way ONOS represents a network is as a directed graph, also known as digraph, meaning that the edge (link) between devices has a specific direction, pointing from one vertex (device) to another one. This subsystem is in charge of listening to any update or change in the network such as a link being deactivated, and periodically creating a topology graph acting as a picture of the network at that given moment in time. Apart from this, it is also in charge of assigning the weights to each link and computing the path given a source and destination. The logical representation of a network in ONOS is composed of the following entities: device, port, host, link, edge link, path, and topology. In section 2.2.2 we have thoroughly analyzed the subsystem for the device and now we will do the same for the topology. Fig 2.6 depicts the topology subsystem.

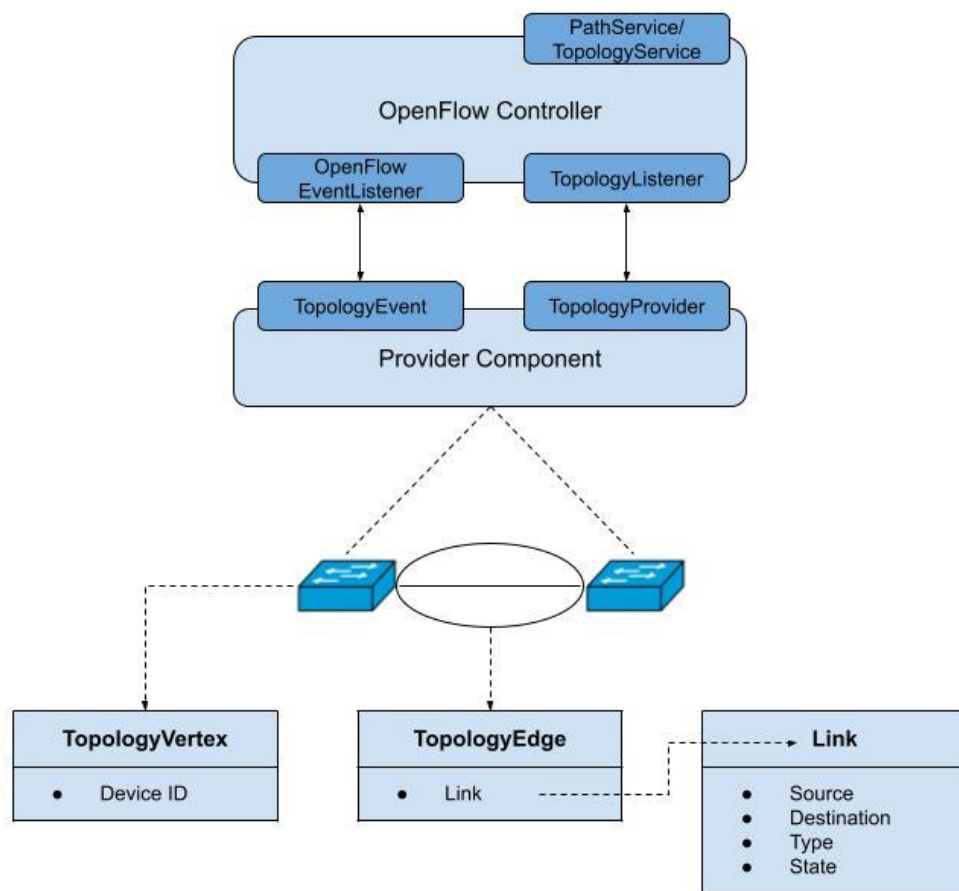


Fig. 2.6 ONOS representation and adaptation of a SDN in the context of the topology subsystem

As can be seen, the logical representation in ONOS for a device and a link in a digraph is through the following interfaces:

- The **TopologyVertex** defines a physical switch, either a source or destination of a particular link. It just has one value, which is the URI that identifies that switch.

- On the other hand, the **TopologyEdge** defines a physical link. This link is composed of source and destination connection points. These are an abstraction of a pair of the element ID, the switch in our case, and its port number. Besides these values, it is defined the type of the link, namely direct, indirect, edge, or virtual, and whether it is active or inactive.

As for the architecture of the subsystem, the *TopologyProvider* is in charge of recalculating the topology when told so by the controller, and the *TopologyEvent* serves as the sender of a topology change. Every time a link is down, or a switch has been disconnected, the topology will be recalculated and the provider will notify the controller about these events. Lastly, on the one hand the *TopologyService* provides network information, such as the current graph of the network and the list of clusters, in case there is more than one controller. On the other hand, the *PathService* is in charge of computing the path between a source and a destination. By default, ONOS always returns the list of shortest paths using Dijkstra, but it also has other routing algorithms like Tarjan [16], Suurballe [17], KShortest³, and LazyKShortest⁴.

2.2.5. Group Subsystem

The last subsystem we are going to talk about is the one related to groups. As we saw in section 1.2.3, groups are ideal when the action to be taken when flows match the different criteria is shared for different flow rules, among other purposes. As it is depicted in Fig 2.7, the manager of this subsystem, that resides in the controller, exposes a service called *GroupService*, with which external applications can request the controller to perform different operations such as adding a group, adding buckets to a specific group, remove a particular group, and so on and so forth. In this same manager, there are as well the *OpenFlowEventListener* and the *GroupListener*, being the ones in charge of receiving any event related to groups, including addition, removal, updates, requests, etc, from real switches in the network by means of the OF protocol. When it comes to the logical representation in ONOS for a group entry and a bucket is through the following interfaces:

- The **GroupDescription** is the representation of a group entry. To begin with, it has a number that serves as an identifier (Group ID), the switch in which this entry is installed using the URI, and the application ID in case there are many applications running at the same time, each using different flow and group entries. Likewise, ONOS supports the four different OF groups, separating in this case the all group with another one called *clone*, used for duplicating packets independently of the output decision. The group description can point to multiple group packets, having a one to many relationship.

³ Runs K shortest paths algorithm on a provided directed graph. It will return paths in ascending order according to the provided EdgeWeight.

⁴ Is the same algorithm as KShortest but in this case it will only compute the path until it becomes necessary, if ever.

CHAPTER 3. DESIGN AND IMPLEMENTATION OF A ROUTING APPLICATION IN ONOS

In the first two chapters we have seen what OpenFlow is, why we have chosen ONOS as the preferred controller for this project and how it works. In this third chapter, we will cover the installation of our custom application, the different classes and components there are, as well as the first steps when it comes to routing packets with tools such as Grafana, JPerf, and Mininet, among others, laying on the table the limitations of traditional routing algorithms. Our application will be in charge of routing the packets that want to enter the network. Initially, this application will use the same routing algorithm as the already existing one named **fwd**. After analyzing the behavior in a simple topology, we will adapt the link weights so the routing algorithm selects the path depending on the occupancy of the links. Finally, both results will be compared and we will propose the use of RL to allocate traffic taking into account the future impact.

3.1. Creation, installation, and deployment of our custom application

Before the creation and subsequent installation of our application, first is needed to install the ONOS controller in our machine with all the necessary tools and extensions. As this process is very long and cumbersome, it is explained thoroughly step by step in Appendix A. What we will see in section 3.3 onwards has been built upon the already existing **fwd** application. This application is of type reactive, meaning that it only reacts when a packet is sent to the controller. After this initial installation process of ONOS, we will just need to load our routing application into the ONOS controller, which will be named **routing-app**.

During the process, it will be necessary to specify the *groupId* and the *artifactId* of the application, which basically defines the package and the application name respectively, as we previously saw in Table 2.1. Accessing the ONOS CLI using the IP address of our local ONOS instance, running the command **onos 192.168.99.106⁵** in the \$ONOS_ROOT directory, ~/onos in our case, will allow us to show all the activated applications (see Fig 3.1).

Apart from this, to make sure that the application is running we can take a look at the logs on the terminal running ONOS (see Fig 3.2). Table 3.1 shows the commands that we will execute in order to generate the necessary “.oar” bundle to install the application:

⁵ This IP address might be different depending on the environment where ONOS is run, since this can be defined by the user in the ONOS configuration. By default, we have decided to use a localhost address.

Table. 3.1 Commands to generate the .oar bundle to install the application

```

$ cd ~
$ cd onos/apps
$ mvn archetype:generate -DarchetypeGroupId=org.onosproject
-DarchetypeArtifactId=onos-bundle-archetype -DarchetypeVersion=2.4.0
-   groupId: org.routing.app
-   artifactId: routing-app
$ cd routing-app
$ nano pom.xml
-   Uncomment the properties tag in order to generate the corresponding
    ONOS application and give it a brief description modifying the
    description tag. Besides that, add the necessary dependencies to run
    (Table C.1 of Appendix C)

```

Next, there is a list of commands to use depending on what we want to execute. Since we use Maven, all these commands have to be executed in the `$ONOS_ROOT/apps/routing-app` directory, where the `pom.xml` file is located:

- **mvn clean install** → Build the application using Maven
- **onos-app 192.168.99.106 install!**
target/routing-app-1.0-SNAPSHOT.oar → Install and activate the application using as a target the .oar file of this application. If we were to make some changes and we would need to install the application again, use **reinstall!** instead.

```

sdn@Sergl: ~/onos
sdn@Sergl: ~/onos
sdn@Sergl: ~/onos/apps/rou...
sdn@Sergl: ~/onos/utlis/ml...

sdn@root > apps -a -s
* 6 org.onosproject.hostprovider      2.4.0 Host Location Provider
* 26 org.onosproject.drivers          2.4.0 Default Drivers
* 27 org.onosproject.optical-model    2.4.0 Optical Network Model
* 37 org.onosproject.openflow-base    2.4.0 OpenFlow Base Provider
* 38 org.onosproject.lldp-provider    2.4.0 LLDP Link Provider
* 59 org.onosproject.gui2             2.4.0 ONOS GUI2
* 67 org.onosproject.openflow         2.4.0 OpenFlow Provider Suite
* 79 org.onosproject.proxyarp         2.4.0 Proxy ARP/NDP
* 194 org.routing.app                 1.0.SNAPSHOT Custom Routing Application
sdn@root >

```

Fig. 3.1 List of installed and activated ONOS applications

```

sdn@Sergil: ~/onos
...
2021-01-02T17:28:33,498 | INFO | Features-3-thread-1 | FeaturesServiceImpl | 11 - org.apache.karaf.features.core - 4.2.8 | Done.
2021-01-02T17:28:33,497 | INFO | Features-3-thread-1 | ReactiveForwarding | 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT
| Custom Routing Application Started!
2021-01-02T17:28:33,500 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Packet-out only forwarding is disabled
2021-01-02T17:28:33,500 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Forwarding using OFPP_TABLE port is disabled
2021-01-02T17:28:33,500 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. IPv6 forwarding is disabled
2021-01-02T17:28:33,500 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Match Dst MAC Only is disabled
2021-01-02T17:28:33,500 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Matching Vlan ID is enabled
2021-01-02T17:28:33,500 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Matching IPv4 Addresses is disabled
2021-01-02T17:28:33,500 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Matching IPv4 DSCP and ECN is disabled
2021-01-02T17:28:33,500 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Matching IPv6 Addresses is disabled
2021-01-02T17:28:33,500 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Matching IPv6 FlowLabel is disabled
2021-01-02T17:28:33,500 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Matching TCP/UDP fields is disabled
2021-01-02T17:28:33,501 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Matching ICMP (v4 and v6) fields is disabled
2021-01-02T17:28:33,501 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Ignore IPv4 multicast packets is disabled
2021-01-02T17:28:33,501 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. record metrics is disabled
2021-01-02T17:28:33,501 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Flow Timeout is configured to 5 seconds
2021-01-02T17:28:33,502 | INFO | CM Event Dispatcher (Fire ConfigurationEvent: pid=org.routing.app.ReactiveForwarding) | ReactiveForwarding
| 214 - org.routing.app.routing-app - 1.0.0.SNAPSHOT | Configured. Flow Priority is configured to 10
2021-01-02T17:28:33,502 | INFO | onos-store-app-activation | ApplicationManager | 193 - org.onosproject.onos-core-net - 2.4
| Application org.routing.app has been activated
2021-01-02T17:28:37,995 | INFO | onos-event-dispatch-programming0 | ReactiveForwarding | 214 - org.routing.app.routing-app - 1.
0.0.SNAPSHOT | Size of installed forwarding objectives: 0
2021-01-02T17:28:37,995 | INFO | onos-event-dispatch-programming0 | ReactiveForwarding | 214 - org.routing.app.routing-app - 1.
0.0.SNAPSHOT | Flow Rule Added
2021-01-02T17:28:37,999 | INFO | onos-event-dispatch-programming0 | ReactiveForwarding | 214 - org.routing.app.routing-app - 1.

```

Fig. 3.2 Logs in the terminal window running ONOS

3.2. Architecture of the routing-app application

Having explained thoroughly how ONOS behaves and which are its main parts, now we will see in detail all the necessary files to run the environment of this project, although in this section we will focus mainly on the different Java classes that are used in the routing-app application. First things first, to give a general perspective about how the project folders and files are organized, Fig 3.3 depicts a portion of the directory structure for this application. Nonetheless, although they are present, some folders and files have been omitted just to simplify the representation, but in subsequent sections we will also see the network topologies and the different scripts to interpret and visualize the results.

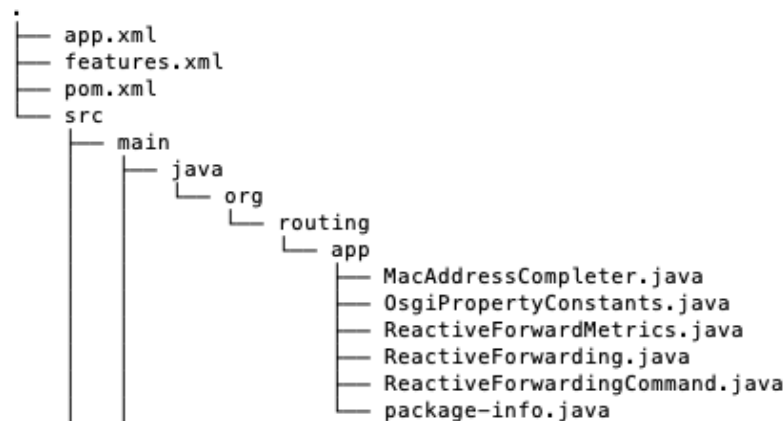


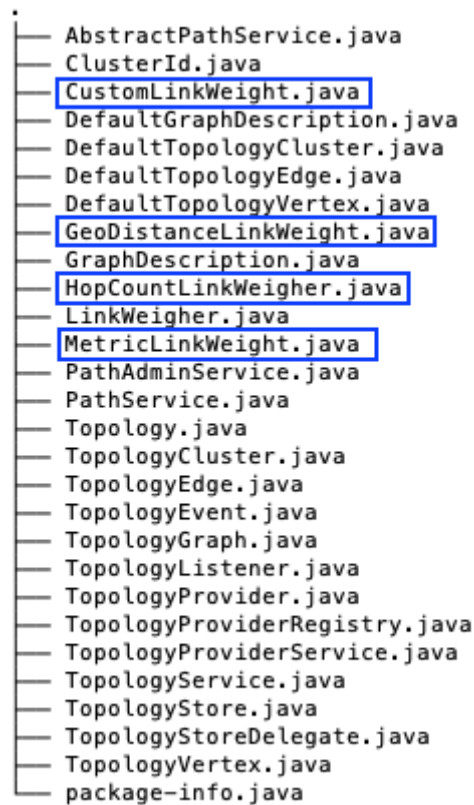
Fig. 3.3 Directory structure of our custom application routing-app

To begin with, as we saw in section 2.2.1, every custom application in ONOS has a set of configuration and definition files of which, in our case, we will only modify the *pom.xml*. This file is of type XML and is the place where, apart from defining the different information of our application, it will allow us to specify and install afterwards different dependencies we might need in our application. To name but a few, ONOS API artifacts such as the different components for services, different core serializers, the CLI seen in section 2.3.2, and non-related ONOS artifacts. To see the file completely please refer to Table C.1 of Appendix C.

Continuing with the structure, as can be seen in Fig 3.3, even though there are six classes inside the */app* folder, in our application only *ReactiveForwarding* will be modified in section 3.3 and the remaining of the project, being the main file and the core logic. The **ReactiveForwarding** class defines the complete logic of our application, being in charge of processing the incoming packets, the flow rule, topology, and group changes, as well as monitoring if any link exceeds. As its name suggests, reactive forwarding means that our application will only act upon traffic sent from one host to another, thus configuring the devices accordingly with the necessary flow rules to properly forward this traffic.

Once we have seen the different classes of our application, it is very important to understand another class that we will modify and will have a great impact on our application. It is the one in charge of assigning the weights to each link of the network based on our criteria. We can find this class under *core/api/src/main/java/org/onosproject/net/topology* called *CustomLinkWeight*, as shown in Fig 3.4. ONOS offers the possibility to decide the weight of a link based on different criteria. Be it the geographic distance between link vertices determined by the latitude and longitude thereof, by the number of hops between a source and a destination, and by the link *metric* annotation, which can be set to a fixed value when defining the network. Finally, the **CustomLinkWeight** class gives the creators and managers of the application complete freedom when it comes to the criteria to determine the link weight. This is the class that we will modify in our application as we will see in the coming sections.

By default, the Dijkstra routing algorithm used in *fwd* application assigns a weight of **one** to all the links in the network. Since it has reactive behavior, when asking the controller this will only tell the switch where to forward the packet. The next switch will receive this packet and will ask the controller where to send the packet to get to the destination, and so on and so forth until the packet reaches the last switch immediately connected to the destination host.



```
├── AbstractPathService.java
├── ClusterId.java
├── CustomLinkWeight.java
├── DefaultGraphDescription.java
├── DefaultTopologyCluster.java
├── DefaultTopologyEdge.java
├── DefaultTopologyVertex.java
├── GeoDistanceLinkWeight.java
├── GraphDescription.java
├── HopCountLinkWeigher.java
├── LinkWeigher.java
├── MetricLinkWeight.java
├── PathAdminService.java
├── PathService.java
├── Topology.java
├── TopologyCluster.java
├── TopologyEdge.java
├── TopologyEvent.java
├── TopologyGraph.java
├── TopologyListener.java
├── TopologyProvider.java
├── TopologyProviderRegistry.java
├── TopologyProviderService.java
├── TopologyService.java
├── TopologyStore.java
├── TopologyStoreDelegate.java
├── TopologyVertex.java
└── package-info.java
```

Fig. 3.4 Directory structure of the topology core folder

3.2.1. Complementary tools

Apart from the architecture strictly related to ONOS core files and its belonging API, there are also other indispensable files and tools we have used in order to configure all our environment, to generate traffic, and to see the results. These tools are grouped in three different types.

3.2.1.1. Monitoring tools

Every time we generate new traffic between hosts, we will need to use tools in order to monitor the desired parameters, including the storage and the visualization. On the one hand, we will use **InfluxDB** as the open-source time-series database to store all the measurements and its values. On the other hand, we will use **Grafana** as the open-source platform for monitoring and observability. Inside the Grafana folder we will have different dashboards for our switches, as well as a Python file called *collector.py*, which is in charge of permanently requesting data to ONOS and collecting it so as to store it using InfluxDB. For more information about the installation process of these tools and the detail of files, please refer to the Appendix B and Appendix C.2.

3.2.1.2. Network topology tools

To configure all the switches, links, and hosts for our topologies, we will use **Mininet**. This tool is very powerful and easy to use, as it creates a realistic virtual network on a machine (either virtual, native or in the cloud). In summary, the python scripts under *mininet/topologies* folder are in charge of defining the network in Mininet and upload it to ONOS using the API. To see thoroughly the installation process of this tool, please refer to Appendix A.5 and Appendix A.6, and to see how the different files are developed, refer to Appendix C.1.

3.2.1.3. Traffic generation tools

The last group of these tools makes reference to the files and the software required to generate traffic between hosts. In this project we have made use of **ping**, which is a software utility that sends Internet Control Message Protocol (ICMP) packet requests and the receiver sends back packet replies. Likewise, another tool is **iPerf**, with its GUI called JPerf, that sends either TCP or User Datagram Protocol (UDP) packets to a destination device. Lastly, we have made use of MGEN, a very powerful tool to send TCP and UDP traffic defining intervals of time, speeds, traffic distribution patterns, and many other options. The installation of these tools and all its files is detailed in Appendix A.7.3 and Appendix C.3.

3.3. Results using Dijkstra as implemented in ONOS

Having seen all the configuration of the custom application and the most striking aspects of its architecture, now it is time to explain thoroughly how the application behaves and, therefore, the results obtained when using Dijkstra with the weight of links set to **one** by default.

3.3.1. Flowcharts of each component of the application

First of all, we will give a clear and detailed explanation about the main parts of the application as well as how they behave as a whole. As depicted in Fig. 3.5, the **ReactiveForwarding** Java class, by default, has four well-differentiated threads that work in parallel, each of them executing a different set of actions every time they receive an event. Nevertheless, even though the main threads are kept intact, in some parts, we have optimized the way flows are installed once a path by ONOS is found. The purpose of doing this is avoiding every device of the established path to ask the controller the flow rules it needs to install in order to route the incoming packet. We will see in the next section why we have tweaked this behavior. Furthermore, we have added the possibility to handle device, link, and topology events. Bear in mind that in every diagram of this section, the new added parts are highlighted in blue, for a better differentiation.

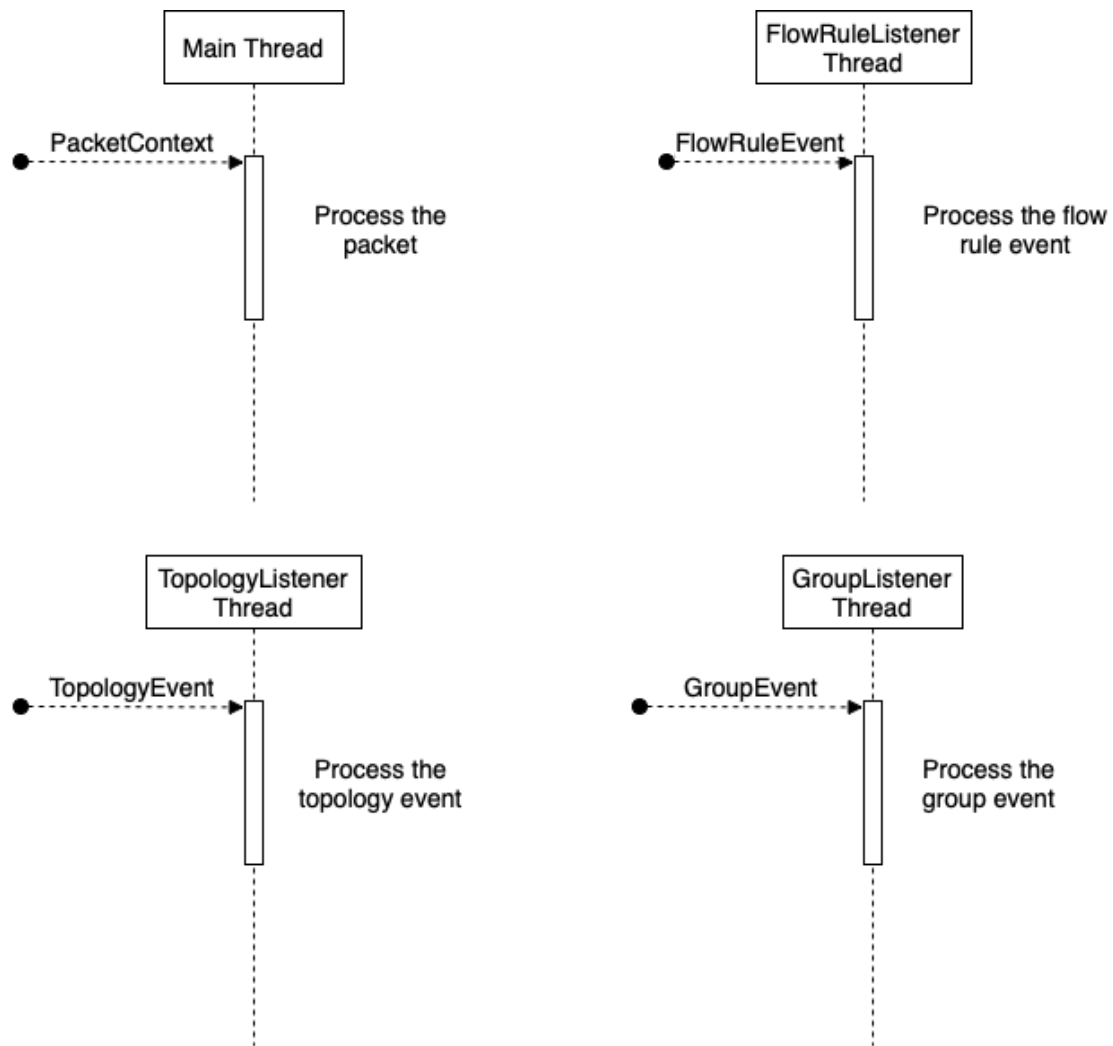


Fig. 3.5 UML diagram of the different threads of the ReactiveForwarding Java class using Dijkstra as implemented in ONOS

3.3.1.1. Packet processing

Before all else, we will begin by explaining what actions are taken by the **ReactivePacketProcessor** class -the main thread in Fig 3.5- when the controller receives a packet. Bear in mind that this class is inside the parent one called *ReactiveForwarding*. Fig 3.6 depicts clearly how this class reacts to this event.

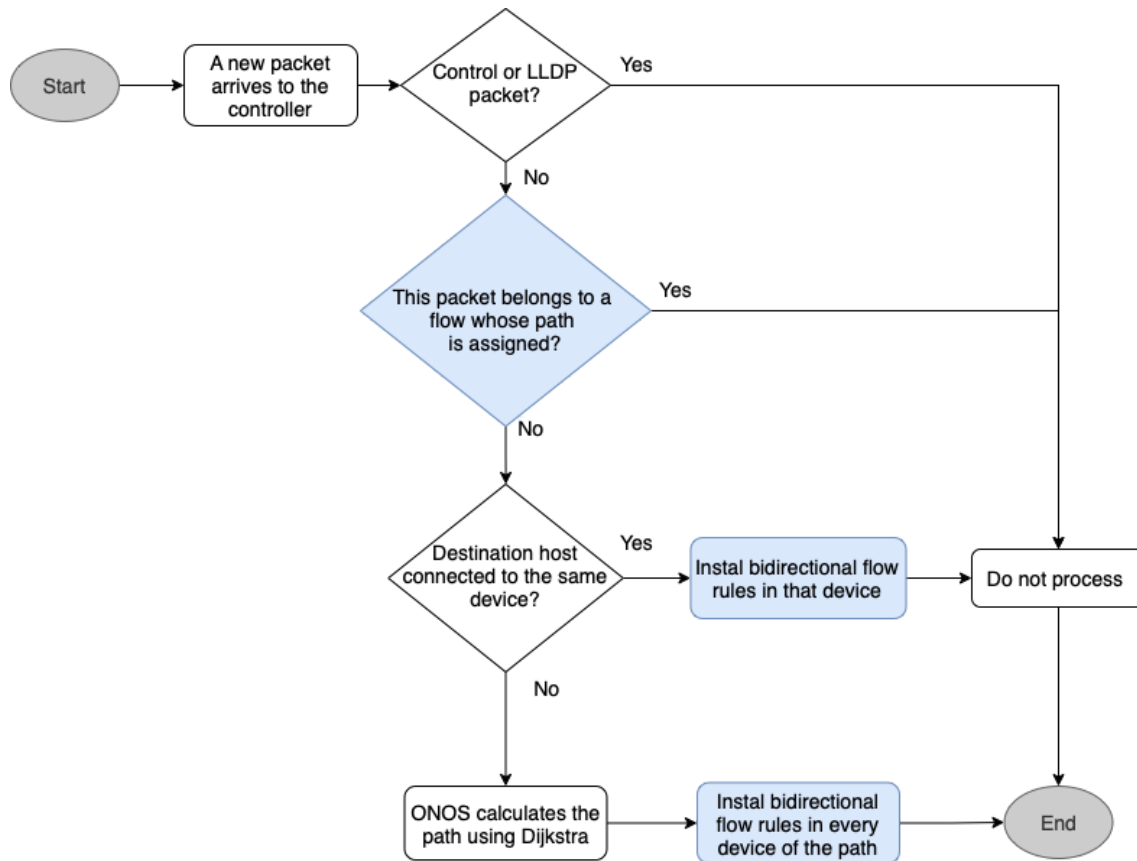


Fig. 3.6 Flowchart of the processing of the packets using Dijkstra as implemented in ONOS

Due to the fact that we are not interested in processing control packets or the ones of type LLDP (first diamond figure in Fig. 3.6), this class will only process those that are intended to send traffic along the network. To understand why is checked whether there is any forwarding objective (second diamond figure) -name given in ONOS to the class in charge of installing the flow rule- pending to be added or removed, it is essential to recall how the default forwarding application -named *fwd*- works. When a device receives a packet and does not know what to do with it, it asks the controller. Then, the controller finds the shortest path and installs the flow rule. This is repeated by every device until this packet reaches the destination. If applicable, the process will be repeated for the reverse path. To avoid finding a different way back only the packets sent by the source device will be taken into account.

Having explained this, let's see what actions are taken when the packet fails to meet the first two criteria (first two diamond figures). The most straightforward situation occurs when the destination host is connected to the same device as the source host. Given that it is not required to find a path, the required flow rules will be installed directly. On the contrary, if the flow is new -is not pending to be installed or removed- the topology service of ONOS will get all the possible shortest paths between that source and destination. By default, the path administrator service uses the *DijkstraGraphSearch*, which has a mapping of the network as a graph and calculates the path with the Dijkstra algorithm.

Finally, when a path has been found, all flow rules and groups are installed. The method in charge of installing the bidirectional flow rules is used for each device that makes up the found path for a specific flow⁶. A total of four different “device situations” are dealt with (in the “Install bidirectional flow rules in that device” box), including when the device is the source, when the device is directly connected with the destination host, when the device is an intermediate one, and when the device is the destination. Apart from this, as flows need to be distinguished when going through a link so as to know precisely the rate of each of them, a group has been for every flow rule installed in a device using a hash for the group identifier. This hash is derived from two inputs, being these the source and destination MAC addresses of the hosts.

To have this control of which packets going through the link belongs to which flow, the action of every flow rule will have the group identifier to which it points, instead of having the output number directly. This group will be of **indirect** type -the simplest one, as we saw in section 1.2.3- since it reduces considerably the memory usage and, more importantly, because no extra action has to be taken, just outputting the packet through the specified port.

3.3.1.2. Topology listener

The class that is covered in this section is a thread that is always running listening to a plethora of topology events. Even though only two of them are really important for the purpose and scope of this project, they play an important role when it comes to managing flow rules and groups when there is any change in the topology. Fig. 3.7 depicts in detail the following process.

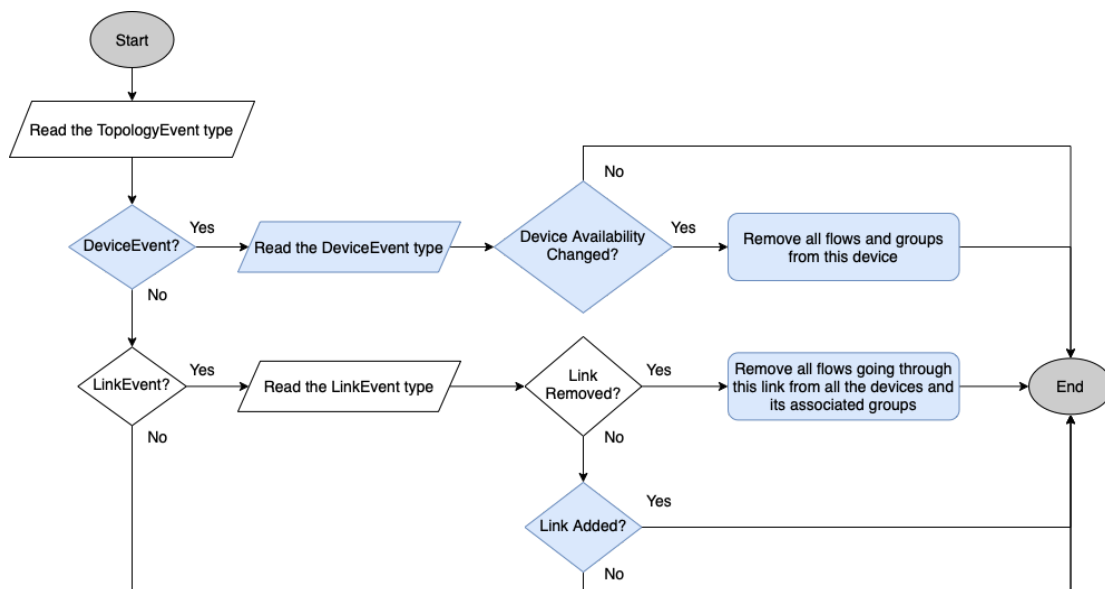


Fig. 3.7 Flowchart of the topology listener using Dijkstra as implemented in ONOS

⁶ From now on, in order to see the details of the code in Java, please refer to the Github repository <https://github.com/SergiVera/onos>, and select the main branch. There you will find all the files and classes mentioned in these chapters.

When the `InternalTopologyListener` class, which is inside as well of the `ReactiveForwarding` Java class, receives a topology event, the method checks whether it is a device event, meaning that there has been a change in one of the devices that forms the network, or a link event, received when any of the links has undergone any modification. Regarding the first event, in case a device is removed from the topology, all flows installed in this device having GROUP as the action, as well as all the belonging groups of this flow will be removed. In relation to the latter event, if a link is removed all flow rules and groups of the flows going through this link will be removed from all the devices, as it is taken for granted that the old path will no longer be valid. The main reason to delete the flow rules manually instead of waiting for the flow timeout to be consumed, is to accelerate the process of finding a new path, considering the few seconds it might take to install all the flow rules.

3.3.1.3. Flow rule listener

Similar to the previous section, this thread defines the second out of the three listeners present in the `ReactiveForwarding` Java class. The **`InternalFlowRuleListener`** class takes different actions based on whether a flow rule has been added or removed, as depicted in Fig 3.8.

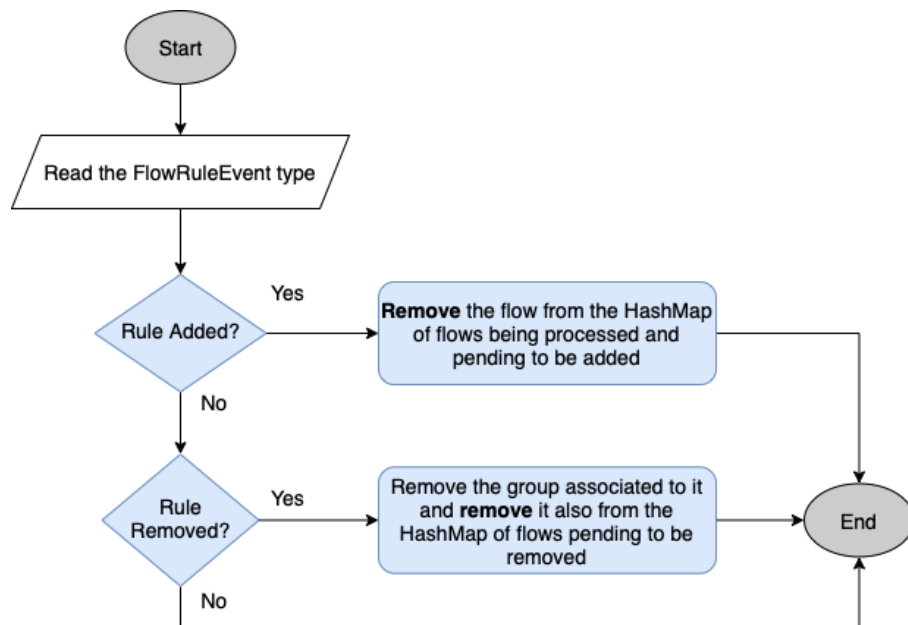


Fig. 3.8 Flowchart of the flow rule listener using Dijkstra as implemented in ONOS

When the main class receives a flow rule event, it processes the type of the event. On the one hand, if a flow rule has been added to a device, this forwarding objective will be removed from the `HashMap`⁷ of flows being

⁷ In Java, a `HashMap` is the representation of a hash table, which stores a list of key-value pairs.

processed and from the list of flows pending to be added. As we have briefly mentioned in section 3.3.1.2, the application might spend a few seconds to find a new path and install all the flow rules. This means that when sending a new flow from a device that is already a source for another flow, our application due to its functioning will cause ONOS to stop counting the bytes received in an interface of a device. This count will not be resumed until the new path is established. Due to the fact that ONOS updates the port statistics every poll interval of **5 seconds**, using the **DefaultLoad** Java class, it will be necessary to store the time taken by our application so it can be used as the interval the first time this rate at the interface is calculated. In this way, a value multiple times greater than expected is avoided.

On the other hand, if a flow rule has been removed, the group associated with it will also be deleted. The way to derive the **group key** -necessary to delete the group- from the group identifier is by hashing again the source and destination MAC addresses of the flow that belonged to. As we saw in section 3.3.1.1, every time a new flow enters the network, the pair of MAC addresses are hashed. The reason behind is due to how the hash algorithm is implemented. Since it is not possible to get the input given a digest (output), the only way we have to get the group key is by hashing again the pair of addresses of the rule that needs to be removed.

3.3.1.4. Group listener

The last of the three listeners located in the main class of our application is the simplest of the three, but no less important, and the inner class is given the name **InternalGroupListener**. This class only processes one type of group event, which is the **removal** of a group. When it is removed, it will also be removed from the HashMap of pending groups to be removed. Bear in mind that this data structure in conjunction with the flow rules pending to be added and to be removed, develop an important role when deciding which flows to process and which ones not. Fig 3.9 depicts the process followed by this listener.

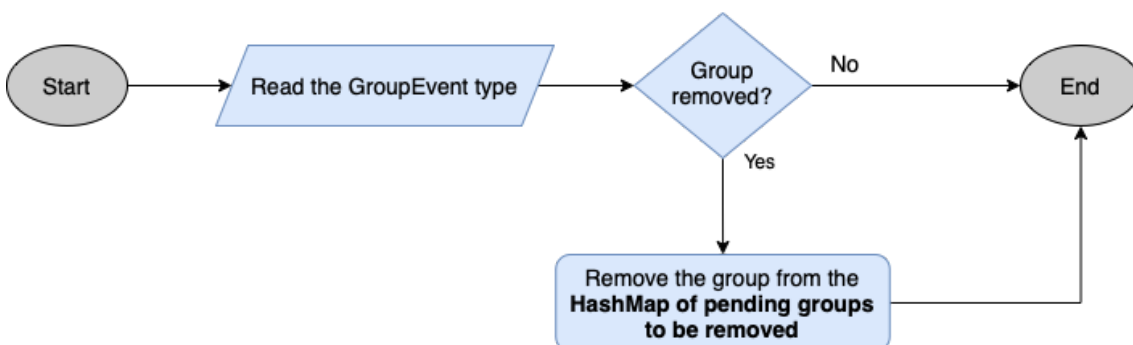


Fig. 3.9 Flowchart of the group listener using Dijkstra as implemented in ONOS

3.3.1.5. CustomLinkWeight Java class

Last but not least, this class that we are going to explain is, undoubtedly, one of the most important. This class is already in the core folder along with the other classes in charge of defining the weighting logic -*MetricLinkWeight*, *HopCountLinkWeigher*, and *GeoDistanceLinkWeigher*, as we saw in section 3.2.- Every time a new path is requested, the default graph path search algorithm used will be **Dijkstra** and the class in charge of giving the weight values of the links to the *DijkstraGraphSearch* class will be the **CustomLinkWeight**. These two classes are set at the initialisation of the *ReactivePacketProcessor* inner class, and we will use the topology service. Dijkstra will request the cost of as many links as needed to find the desired path, replicating the implementation of the algorithm. It is worth mentioning that the behavior of the routing algorithm is not being changed when it comes to the weights assigned in comparison to the default implementation. We are adding this manually since we have modified the way flow rules are installed, as we have seen in section 3.3.1.1. This change in the installation process makes us adapt the *CustomLinkWeight* class.

The potential of this class is that weights of the different links can be modified to meet our requirements, so the algorithm will not always find the shortest path. For now, the behavior of the class is the default one, but in next sections we will change it. In this case, there are two situations in which the value of the weights is chosen in a different way, as depicted in Fig 3.10:

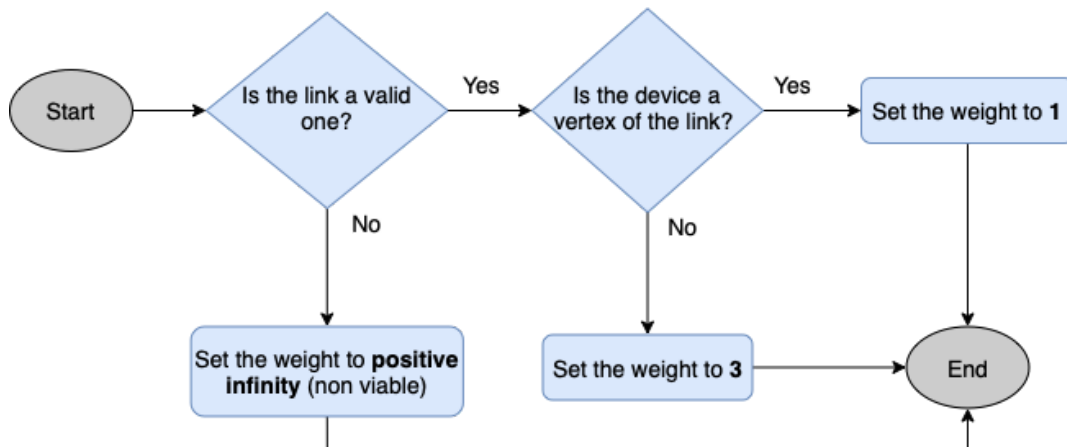


Fig. 3.10 Flowchart of the link weight decision in the CustomLinkWeight class using Dijkstra as implemented in ONOS

To start with, if the link is not valid, meaning that is erroneous or does not exist, the assigned value will be **infinite** -the maximum possible value held by the Java Double class, which is big enough-. Likewise, if the link is valid but the device does not belong to either vertex of the link, the weight set will be **3**. Lastly, if both criteria are met (diamond figures in Fig 3.10), the assigned weight will be the lowest possible, being **1**.

3.3.2. Experimental results

Having seen all the components of the application, now we will analyze the results obtained with a specific network topology to see which paths are chosen when the Dijkstra algorithm with default weights by ONOS is used. Fig 3.11 depicts the representation of the network using the GUI provided by the controller's software. As can be observed, the network has a total of **seven switches**, **eight bidirectional links**, and **ten hosts**. As far as hosts are concerned, these are connected to the switches S1 and S7, with five in each switch. The topology has been generated running a network configuration file in Python using the Mininet software (for more details about its implementation, please refer to Appendix C.1.1).

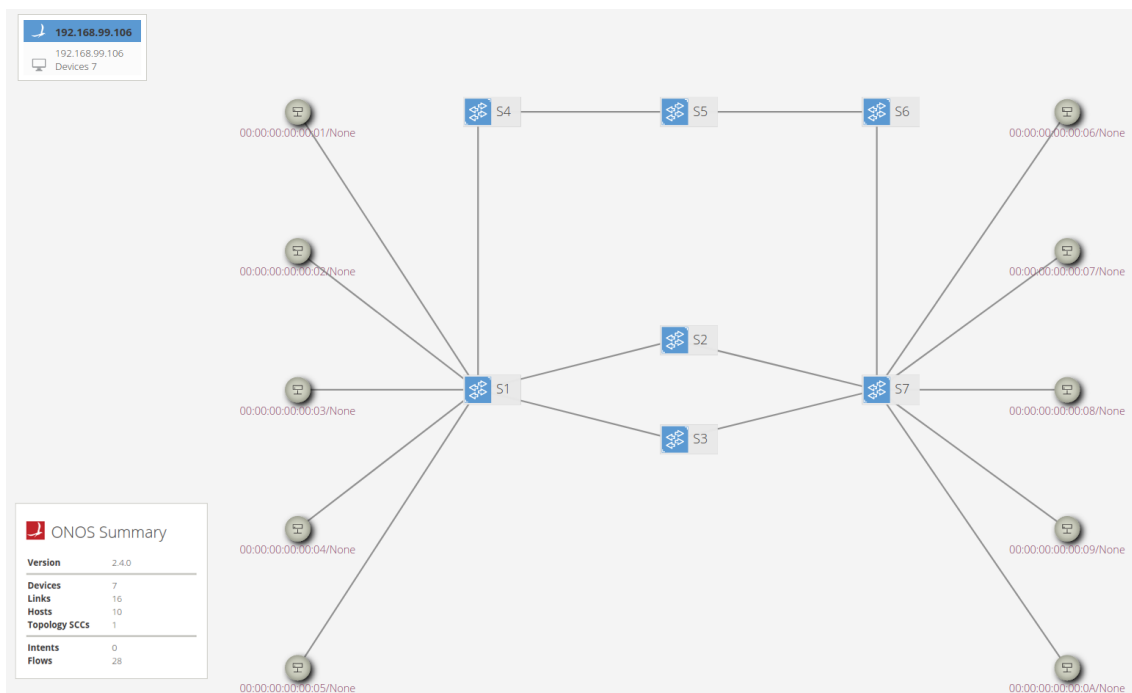


Fig. 3.11 Visualization in ONOS GUI of the topology used for the experimental results for Dijkstra as implemented in ONOS and the implementation using a custom routing based on link occupancies

To get the results for this first approach using Dijkstra as implemented by ONOS, we have used two different traffic generator tools, being the **iPerf** and **MGEN**, respectively. With regards to the visualization of the data, as we saw in section 3.2.1.1, we have made use of InfluxDB to store the data and Grafana to visualize it across the different links and devices of the network. All in all, it is important to mention that the type of flows generated with MGEN and iPerf are exactly the same, although we have used iPerf as well to depict better the BW allocated for each flow. When it comes to the BW of the links, we have decided to work with links of **1 Mbps** for all our scenarios from this point onwards, given the limitations we encountered when generating flows. When working with tools such as Mininet, the hosts are not able to handle Gbps and thus we had to

transmit less data. Nevertheless, since what we want is to exhibit the behavior of the algorithms in a particular scenario, the fact of working with Mbps does not modify it whatsoever.

Having said that, for this scenario we have generated three different flows of type UDP of **1 Mbps** each, between the hosts h1-h6, h2-h7, and h3-h8, stepped one after another. The reason to generate UDP traffic instead of TCP is to avoid the control mechanisms, in terms of congestion, present in this protocol, as well as traffic sent by the receiver in terms of ACKs to acknowledge the packet has been received properly, as our main focus is only in analyzing how the routing is decided when sending packets only from a source to a given destination. If it is true that MGEN can generate many types of different flows, the sender will start with a periodic pattern at a fixed packet size and rate. At the same time, the receiver will listen for UDP messages at ports 5000-5001. To see the commands to generate the flows, please refer to appendix C.3.1. Table 3.2 exhibits the results obtained when transmitting the three different flows. In this case, the overall results are stored in a text file for every traffic generated between a source and a destination. Nevertheless, the output is not parsed and the QoS parameters that are shown in Table 3.2 are obtained with a parser that we have created for that purpose (for more information about this file, please refer to Appendix C.3). As we can see, the obtained results in terms of throughput⁸ **are cut to one third** of the transmission speed, and we also get extra parameters such as the average latency, the loss rate, and the jitter. When it comes to the latency, the value is high (between 1 and 2 seconds) because the packets are waiting in the queue of the switch to be transmitted, due to the fact that we are trying to transmit a total of 3 Mbps in a path that only has links that support 1 Mbps.

Table. 3.2 QoS parameters obtained using Dijkstra as implemented in ONOS⁹

	Flow h1-h6	Flow h2-h7	Flow h3-h8
Average throughput (Mbps)	0.37	0.34	0.37
Average latency (ms)	1,851	2,153	1,967
Loss rate (%)	0	0	0
Jitter (ms)	2.82	3.94	2.42

⁸ The throughput, unlike BW, only takes into account the data that has been transmitted successfully. BW does not care about unsuccessful packets.

⁹ From this table onwards, the punctuation of all numerical results is done using the American notation. Dots to separate numbers and partial numbers, and commas to separate thousands.

When it comes to the results seen in iPerf, we can observe in Fig 3.12 that, for the first ten seconds, the Bandwidth (BW)¹⁰ of the traffic sent between h1 and h6 is of 1 Mbps, meaning that the link still has its full capacity. Nonetheless, when a second flow of data is generated between h2 and h7 (second 10 in Fig 3.12), we observe that the BW is cut in half. Likewise, when a third flow is generated between h3 and h8, as can be seen in Fig 3.12 (second 20) the BW is cut to a third part of the original 1 Mbps, happening the same for the other two flows, giving a BW of **333 Kbps**, on average, when the three flows are being transmitted at the same time. In order to understand why we have obtained these results, we need to analyze what Grafana has captured in the different links and devices.

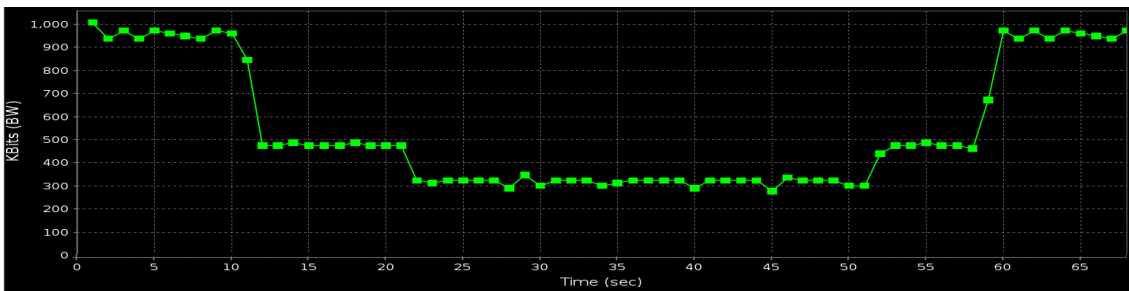


Fig 3.12 BW representation from the perspective of h6 depicted in jPerf using Dijkstra as implemented in ONOS

When the first packet of the flow h1-h8 arrives at the switch, this asks the controller as it does not know what to do with it. The controller, as we saw in section 3.3.1.1, will ask the *TopologyService* to find the shortest path using Dijkstra. As it is implemented by default, all the link weights will be one and the number of shortest possible paths will be **S1-S2-S7** and **S1-S3-S7** in either order. In these situations, when having multiple paths with the same total cost, the topology service will always return the first of the obtained possibilities in the list. It is important to mention that this is not a bad behavior of Dijkstra but about how ONOS logic is implemented. As a result, the 100% of the data traffic will be routed through S3, using the path **S1-S3-S7**, being this the reason why we see the BW in iPerf cut in one third. On the contrary, the links S1-S2 and S1-S4 do not see any traffic going through them, in either direction. To see these results in Grafana, please refer to section C.2.

One last aspect to remark on is how the flow rules are installed in the different devices, in percentage terms (Fig 3.13). ONOS, by default, installs in each device four basic rules: one for BDDP¹¹, ARP, LLDP, and IPv4. On top of these four rules, the devices that forward the data to one of its neighbors will install two more rules per flow - one for the outbound flow and another for the inbound flow. Since there are a total of three flows being transmitted simultaneously, there will be **ten flows** in S1, S3, and S7. As you can see, not only the QoS in

¹⁰ In networking, the bandwidth is the maximum capacity to transmit data in a given amount of time, typically one second.

¹¹ The BDDP ethertype is defined by ONOS with the hexadecimal 0x8942 and it uses the same representation as for an LLDP packet.

terms of BW is being affected, but the workload these switches need to stand is far greater than the others, and it is needless to say that is not balanced among the different devices.

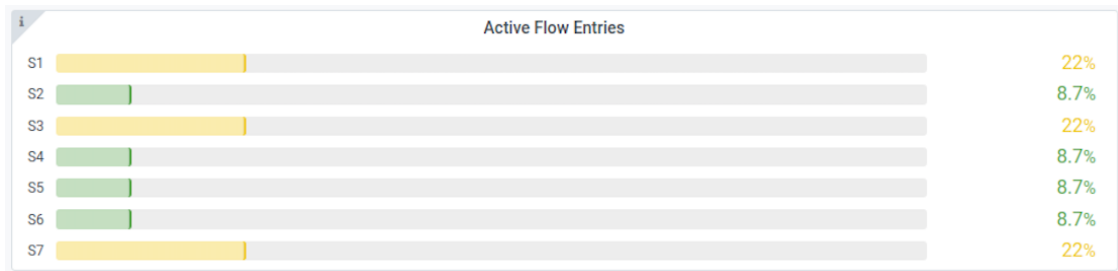


Fig. 3.13 Percentage of flows installed per switch over the total using Dijkstra as implemented in ONOS

3.4. Results using a custom routing based on link occupancies

Having seen the results provided by default by ONOS, using Dijkstra with the default link weights set to one, the network results obtained in terms of QoS are very poor, as packets will always choose the shortest path and if they match, always get the first one in the list. Given this, we will implement a variation to the routing algorithm, assigning the weight depending on the occupancy of the link to then find the shortest path still using Dijkstra. Apart from this, we will also add a control access mechanism to ban from entering the network those flows that cause the link occupancy to go beyond the theoretical bandwidth limit. All together with the objective of improving the QoS.

3.4.1. Flowcharts of each component of the application

As opposed to, now our **ReactiveForwarding** Java class will have a total of five well-differentiated threads working in parallel, as shown in Fig 3.14, being the *InternalGroupCheckerThread* the new one in charge of managing the access control to the network. Let's see in detail how the different threads have changed in comparison to the previous implementation¹² in order to implement the control mechanism logic along with the change in the weight of links.

¹² The threads that are not explained have not suffered any change with respect to the previous section 3.3.

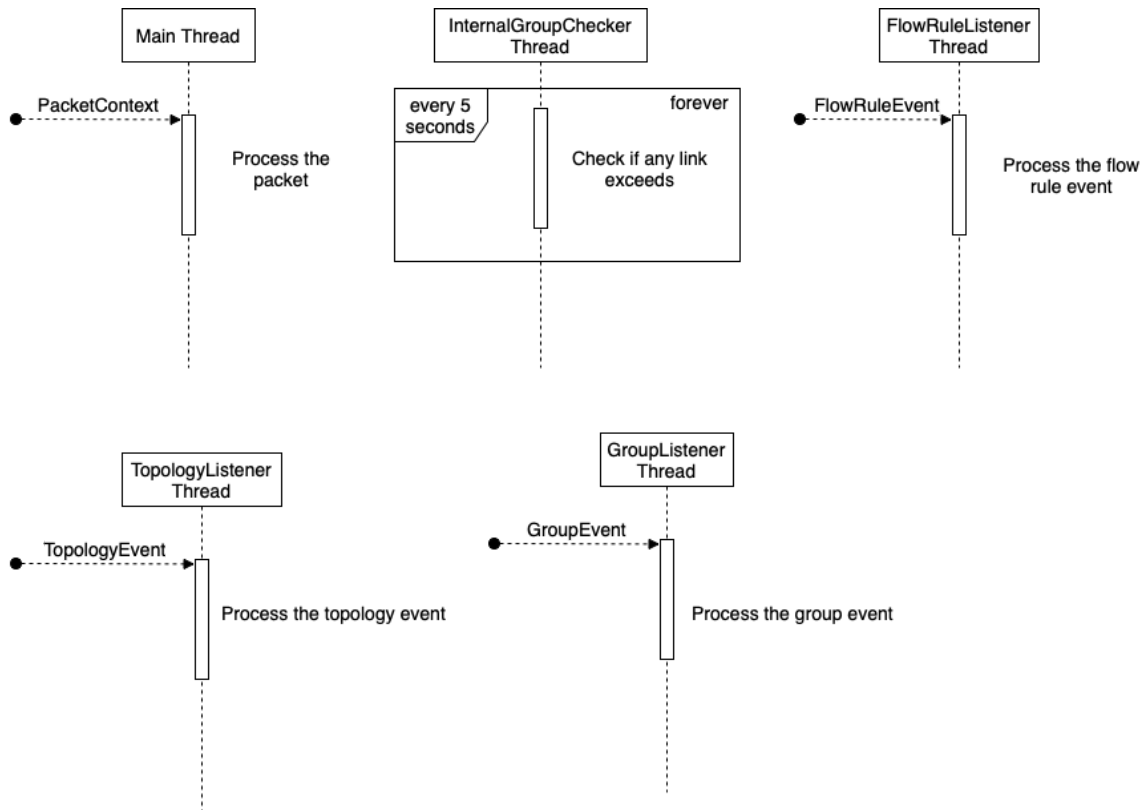


Fig. 3.14 UML diagram of the different threads of the ReactiveForwarding Java class using a custom routing based on link occupancies

3.4.1.1. Packet processing

As depicted in Fig 3.15, apart from not being interested in control packets or the ones of type LLDP, as well as assessing whether there is any forwarding objective pending to be added or removed, we will now also check if a packet belongs to a banned flow, which means that it will not have the right to enter the network. Since we are now granting or rejecting access to flows, we will need to assess if the packet we have just received belongs to a flow to be reallocated in the network. If so, we will inform the class in charge of managing the weights (*CustomLinkWeight*) about it. Otherwise, we will find the shortest path with the current link weight configuration.

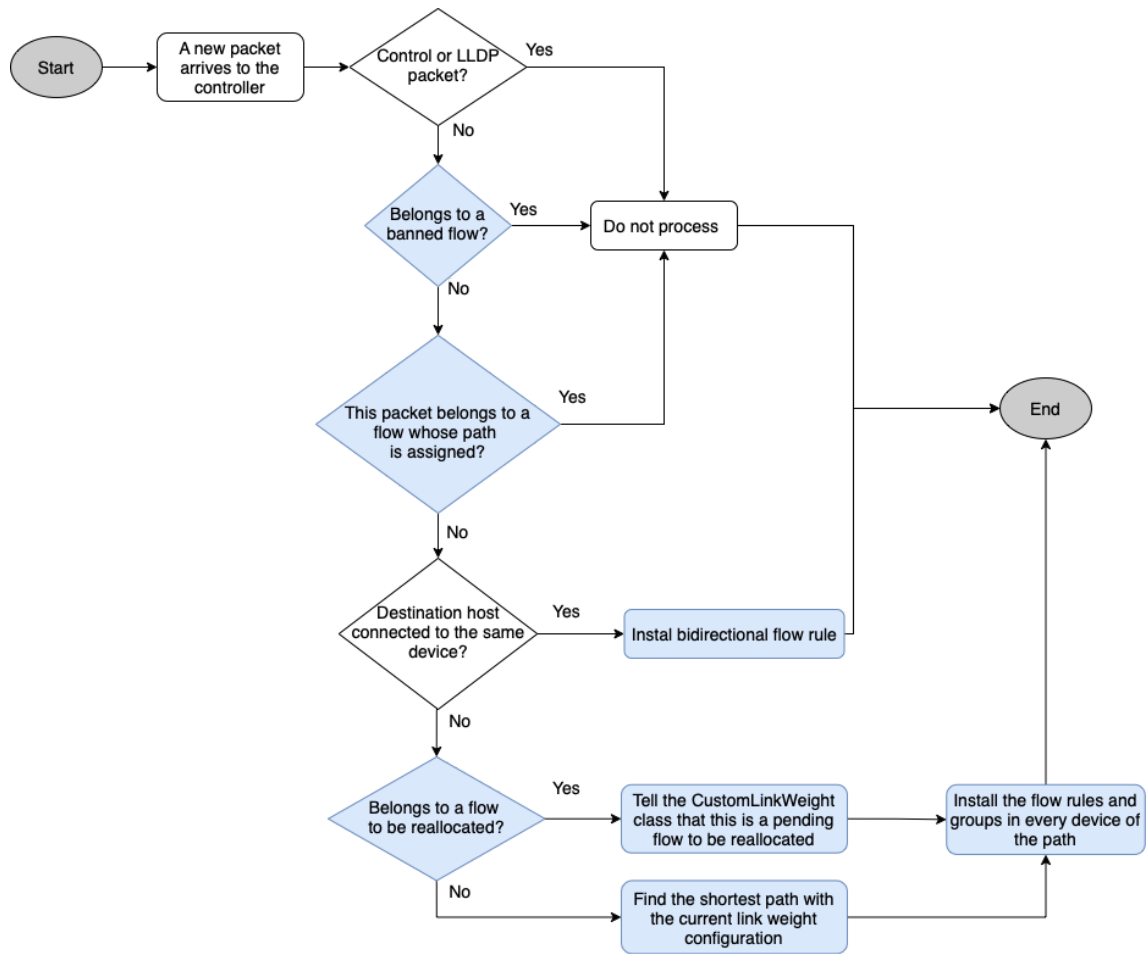


Fig. 3.15 Flowchart of the processing of the packets using a custom routing based on link occupancies

3.4.1.2. Links exceeding group checker

Once the path has been established, it is of extreme importance to oversee all the links of the network, especially the ones constructing the paths, to check whether they exceed its capacity or not. For this reason, in addition to what we had in section 3.3, a schedule thread that is executed every **5 seconds** has been created. The Fig 3.16 shows this procedure:

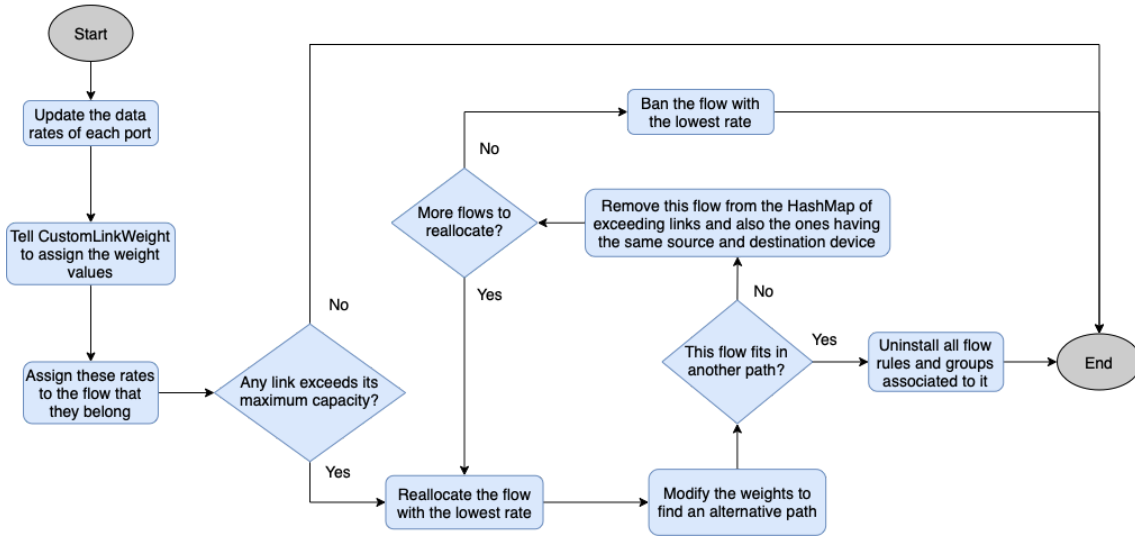


Fig. 3.16 Flowchart of the links exceeding group checker using a custom routing based on link occupancies

In order to keep track of the different flows that go through a link, it will be needed to assign the data rates detected at each port -defined in ONOS as **ConnectPoint**-. Prior to the assignment of these rates, a simple weighted moving average of window size equal to **4** will be applied to the obtained value. The main reason to use this window is to minimize the impact of possible fluctuations that occur from time to time, which have been detected during the coding and testing of this project. The formula used for deciding the rate is as follows:

$$\overline{Rate} = \frac{\sum_{i=0}^N Rate_i \cdot (N-1)}{2 \cdot N \cdot (N+1)}, \text{ where } N = \text{window size} \quad (\text{C. 1})$$

Once the average weighted rate has been calculated, it is assigned to the group that they belong to. The structure used to store them is a double HashMap, more precisely in java notation: *HashMap<ConnectPoint, HashMap<GroupId, Long>>*. Then, when updating the values, the rate detected at the source port -the one directly connected to the host- and at the destination port will be mirrored along the ports for that path, as depicted in Fig. 3.17. It is important to point out that this in conjunction with the group identifiers is the only way we found in ONOS to distinguish flows when they enter the network.

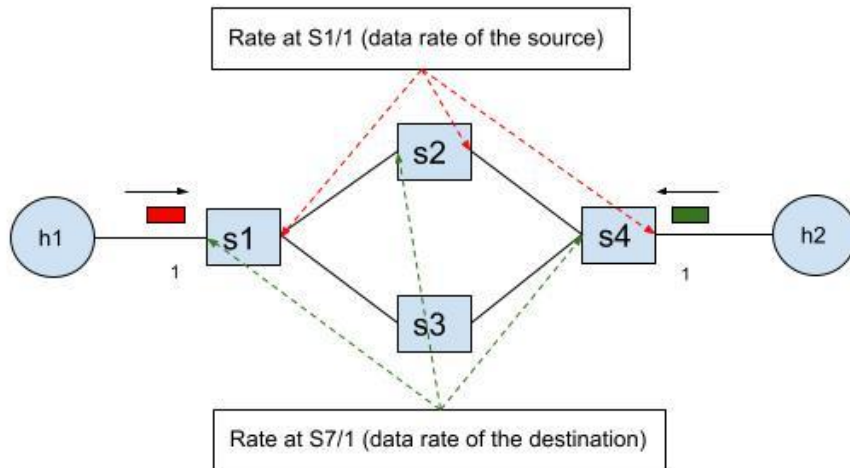


Fig. 3.17 Data rate mirroring along the established path between h1 and h2

Once each of the rates is calculated, it is time to check whether the detected rate exceeds the maximum capacity of that link. If so, a double HashMap with the information of the flows going through this link is drawn up. Firstly, among the exceeding links, it is tried to reallocate in the network the flow with the **lowest rate**. The reason to move the lowest one is to evade the situation where the link that is causing the problem is the highest one. In that scenario we would be moving the problem along all possible alternative paths, with all that this implies in terms of resources and time. The way of achieving these alternative paths is by modifying the links manually, telling *CustomLinkWeight* to update them.

As shown in Fig. 3.16, if the link fits in another path the process is unambiguous; all flow rules and groups associated with this flow are uninstalled, and the link weights are configured in such a way that when using Dijkstra again, the chosen path is the desired one. Conversely, if the flow does not have a possible alternative path, it is removed from the HashMap of exceeding links as well as all the flows having the same source and destination device, drawing from the premise that the same alternative paths will be found again albeit being a different flow. Next, this process is repeated again but this time with the flow with the second lowest rate, hence repeating this process until all flows going through the exceeding links have been checked. If, at its conclusion, no alternative path has been found, the associated flow rules and groups of the most recent flow -the last that entered the network- will be removed. Lastly, the latter will be banned from entering the network.

3.4.1.3. Topology listener

For this section, since now there is a thread that checks the exceeding links, this listener will have to be updated as well. In addition to all previously seen in section 3.3.1.2, now when a link is removed it will also be removed from the HashMap that monitors the exceeding links. On the contrary, when a link is added to the network, it will be needed to add it in the HashMap to monitor the exceeding links. The diagram of Fig 3.18 depicts this new implementation:

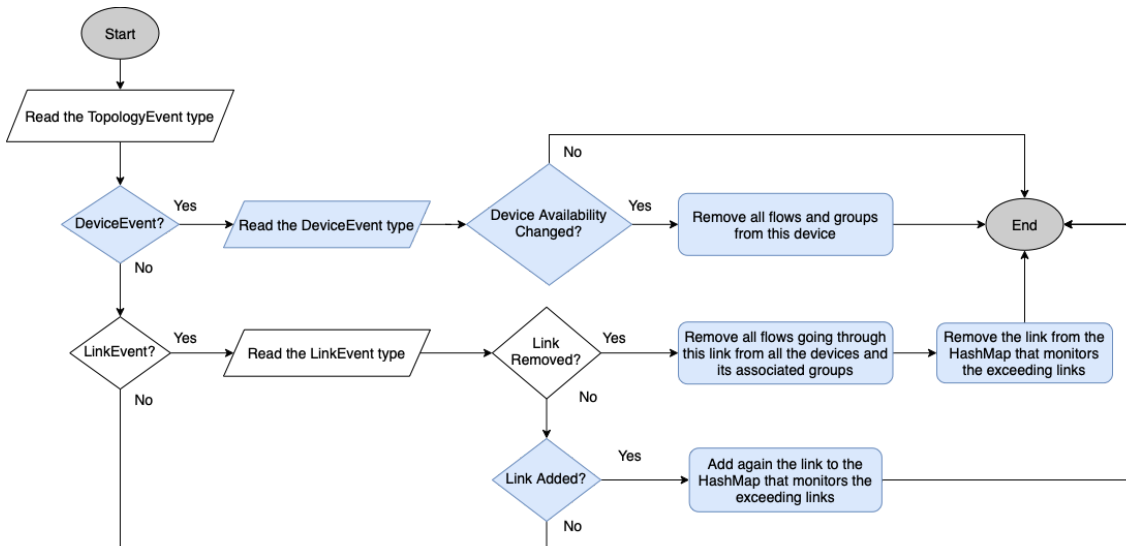


Fig. 3.18 Flowchart of the topology listener using a custom routing based on link occupancies

3.4.1.4. Flow rule listener

When it comes to the listener of flow rules, for this implementation when the flow rules has been added to a device, apart from removing the forwarding objective from the HashMap of flows being processed and pending to be added, if this flow rule belongs to a flow pending to be reallocated -meaning that it had previously exceeded a link and has been moved-, *CustomLinkWeight* will be notified so it updates the weights accordingly (we will see it in the next section). On the other hand, if a flow rule has been removed, in addition to deleting the group associated with it (as we saw in section 3.3.1.4), it will also be checked if there is any banned flow. In case it is, in order to let it enter the network again, it will need to match these two criteria:

- The flow rule removed belonged to a flow whose source and destination device were the **same** as theirs.
- The flow rule removed did not belong to the banned flow **itself**.

Taking this process into account, Fig 3.19 illustrates this new implementation for this listener:

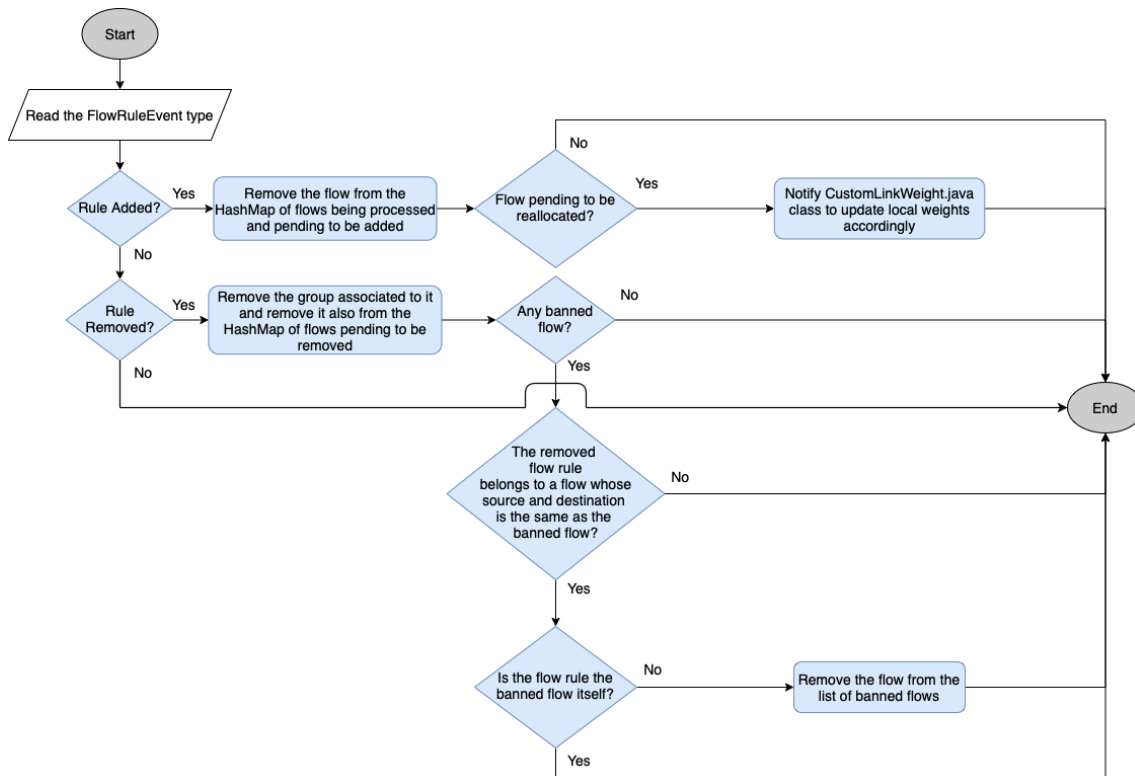


Fig. 3.19 Flowchart of the flow rule listener using a custom routing based on link occupancies

3.4.1.5. CustomLinkWeight Java class

The last class we are going to analyze in this section is the *CustomLinkWeight*. Unlike in section 3.3.1.5 that there was not any flow control access nor a continuous monitoring of the capacity of the links, the weights of the links have changed noticeably as depicted in Fig 3.20.

To start with, if the link is erroneous or does not exist, the assigned value is **infinite** -the maximum possible value held by the java *Double* class, which is big enough-. Assuming the link is valid, if there is not any pending flow to be reallocated, the values are taken based on the current configuration. When explaining the Packet processing section we saw how the main class talks with the *CustomLinkWeight* class to specify the weights (see Fig 3.15). In this case, the weight of the link is set based on the following criteria: if the occupancy is up to 50% (included), the weight assigned is **one**; if the occupancy is greater than 50% and up to 90% (included), the weight assigned is **two**; finally, for occupancies greater than 90%, the weight is **three**. On the contrary, if there are flows pending to be reallocated, and the requested link by *DijkstraGraphSearch* is exceeding its capacity, the value set will be the maximum viable number, which is **32767**. Finally, in the scenario where there are flows to be reallocated and the requested link is not exceeding its capacity, the assigned weight will be **one**. As can be seen, the logic added with respect to the previous implementation is not that difficult, though it is true that now there are more parameters to take into account.

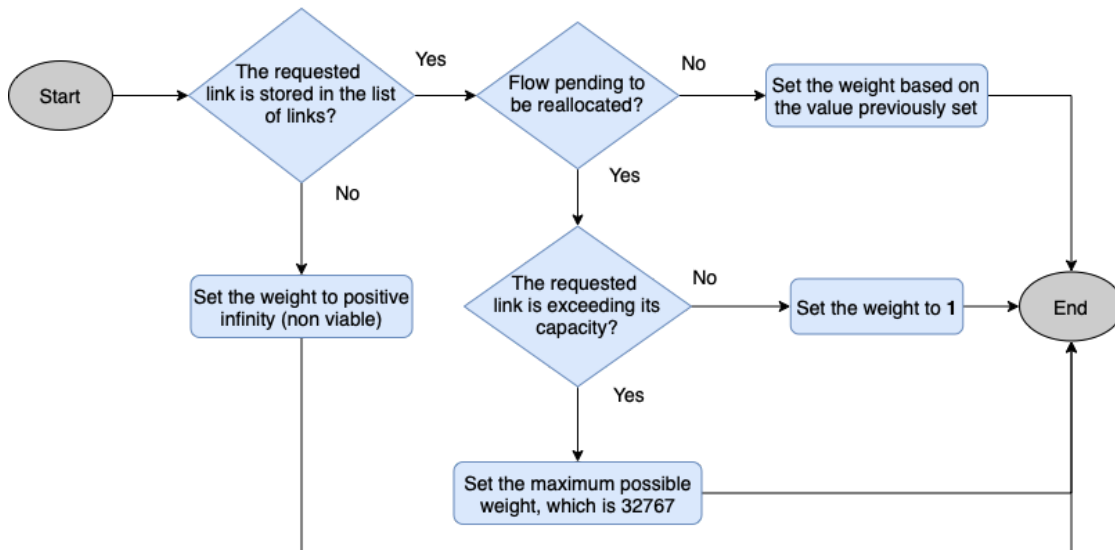


Fig. 3.20 Flowchart of the link weight decision in the CustomLinkWeight class using a custom routing based on link occupancies

3.4.2. Experimental results

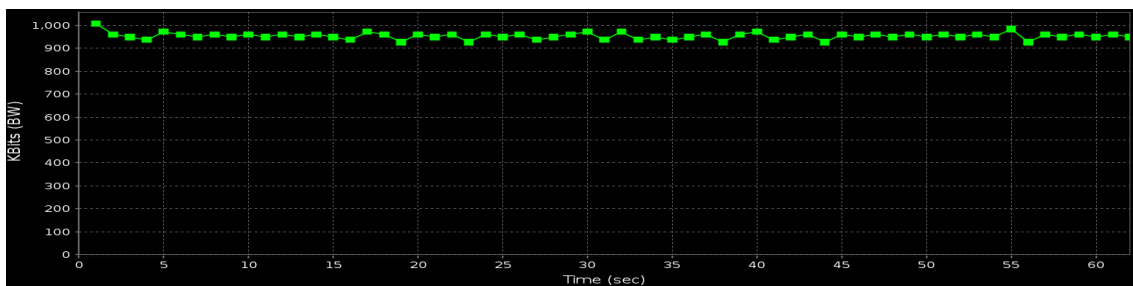
Having seen all the components of the adapted application, now we will analyze the results obtained with the same network topology as in section 3.3.2, with exactly the same configuration of switches, links, and hosts. This means that the flows generated will be the same ones, with the same source and destination and the same rate. The difference now is that we should observe a difference when it comes to the MGEN parameters, as we are making decisions based on the occupancy of the links. As before, we will generate the flows using two different tools but iPerf will only be for BW depiction reasons.

For the results using MGEN, Table 3.3 exhibits the results obtained when transmitting the three different flows using the same parser explained in Appendix C.3. As can be observed, the obtained results in terms of throughput are, on average, **900 Kbps** for each transmission. When it comes to the latency, in this scenario the value has been dramatically reduced because the packets now are not waiting in the queue of the switch to be transmitted. We are not trying to transmit a total of 3 Mbps distributed in three paths of 1 Mbps each. Furthermore, it is worth-knowing that the residual loss rate we see in each of these transmissions is due to the fact that now the time taken to find the path using Dijkstra given these conditions takes some more milliseconds. As we saw when explaining the packet processing workflow, the incoming packets are not treated when we are looking for a new path, and MGEN counts them as a loss as they have not been received.

Table. 3.3 QoS parameters obtained using a custom routing based on link occupancies

	Flow h1-h6	Flow h2-h7	Flow h3-h8
Average throughput (Mbps)	0.90	0.90	0.90
Average latency (ms)	17.68	2.43	18.48
Loss rate (%)	0.07	0.04	0.08
Jitter (ms)	0.07	0.04	0.02

When it comes to the results seen in iPerf, we can observe in Fig 3.21 that, for the first ten seconds, the BW of the traffic sent between h1 and h6 is of 1 Mbps (the maximum possible as the links are of 1 Mb). However, and this is the first striking difference that we observe at first sight, in this case we introduce a second flow of data between h2 and h7, at second 10 in Fig 3.21 we do not see any decrease in the BW at all. As a matter of fact, we observe that it continuously is the same all the time, meaning that the addition into the network of the next two flows, did not have any impact on the first one. Similarly, if we take a look at Appendix C.2.2 to see the links in Grafana, we will see that the BW of the flows is constant throughout the time. Hence, on average, the BW is of **1 Mbps** when the three flows are being transmitted at the same time. So as to understand why we have obtained this improvement in the results, we need to analyze what Grafana has captured in the different links and devices.

**Fig. 3.21** BW representation from the perspective of h6 depicted in jPerf using a custom routing based on link occupancies

When the first packet of the flow h1-h8 arrives at the switch, this will do the same as with the default implementation, which is to ask the controller what to do with it. The controller will find the shortest possible path taking into account now the occupancy of the links, which at the beginning is zero, so the chosen path will be **S1-S3-S7**. Once the second flow wants to enter the network, the links S1-S3 and S3-S7 that are fully occupied, will get a weight of three each, so

now we make sure that this path is not chosen again while the first flow is being transmitted. Based on the total cost of the remaining paths, *DijkstraGraphPathSearch* will now choose the path **S1-S2-S7**. Finally, when the third flow wants to enter the network, both paths will now have a total cost of six, as the occupancies are almost 100%. Therefore, even though the cost of the remaining path **S1-S4-S5-S6-S7** has a cost of four and it implies a total of three hops instead of two, we avoid losing BW.

One last aspect to emphasize is the installation of the different flow entries across the devices. Unlike in the default implementation using Dijkstra with always weights of value one, now we observe that the flows are distributed better and there are flow rules installed in each device, reducing the workload in the devices (see Fig 3.22). As an example, the switches carrying the most installed flows only have 20% of the total installed in the network, in contrast to the 22% we observed in Fig 3.13.

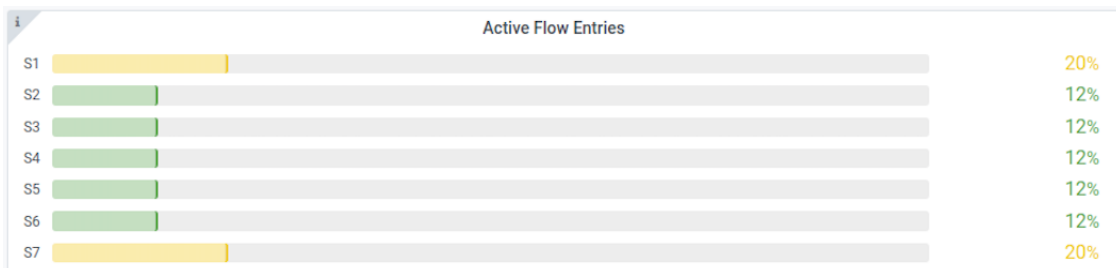


Fig 3.22 Percentage of flows installed per switch over the total using a custom routing based on link occupancies

3.5. Comparison results between both implementations

Having seen the results for the Dijkstra algorithm with default weights by ONOS and for the weights modified based on link occupancies, this section shows the final comparison, at a glance, between both. For it, we have decided to take only the results for MGEN, primarily because it provides us far more information than iPerf, and also because throughput takes into account lost packets, those that have not been received by the destination host. As can be seen in Table 3.4, after implementing the weight of the links in the network based on the link occupancy, the results are noticeably better, especially when it comes to throughput and latency. The flows transmitted between hosts are of 1 Mbps, as we have done for sections 3.3.2 and 3.4.2.

Table. 3.4 Comparison of the QoS parameters obtained using Dijkstra as implemented in ONOS and using a custom routing based on link occupancies

	Routing implementation using default weights			Routing implementation based on link occupancies		
	Flow h1-h6	Flow h2-h7	Flow h3-h8	Flow h1-h6	Flow h2-h7	Flow h3-h8
Average throughput (Mbps)	0.37	0.34	0.37	0.90	0.90	0.90
Average latency (ms)	1,851	2,153	1,967	17.68	2.43	18.48
Loss rate (%)	0	0	0	0.07	0.04	0.08
Jitter (ms)	2.82	3.94	2.42	0.07	0.04	0.02

With these figures in place, it could be thought that there is nothing left to implement if the flows are routed in the network based on the available BW in the links, but there are two reasons so as to improve the performance. Firstly, taking the routing decisions manually based only on one parameter is prone to errors if the network topology escalates rapidly, with more and more devices and links in the network. Secondly, if these decisions are taken arbitrarily, meaning the weights are decided based on different occupancy thresholds set by us, we would not be taking into account how this flow allocation could impact if more flows are going to enter the network in the future. Let's see a scenario in which the allocation of flows really matters for upcoming flows.

As can be seen in Fig 3.23, let's suppose we want to transmit, in this order, flows from h1 to h6 of rate 300 kbps, from h2 to h7 of rate 300 kbps, from h3 to h8 of rate 550 kbps, from h4 to h9 of rate 600 kbps, and from h5 to h10 of rate 600 kbps, totalling for **2,35 Mbps**, with each possible path having 1 Mbps of BW. First of all, based on our link occupancies weighting logic, the first flow gets assigned the path **S1-S2-S7**, and now this path will have an occupancy of 30%. The second flow will get assigned the same path **S1-S2-S7**, leaving the path with an occupancy of 60%. The third flow will get assigned the shortest empty path **S1-S3-S7**, leaving the path with an occupancy of 55%. When the fourth flow h4-h9 wants to enter the network, based on the current occupancies, the path **S1-S4-S5-S6-S7** will be assigned, leaving the path with an occupancy of 60%. Lastly, the last flow will not be able to enter the network as there is no possible way to allocate it unless we start moving flows, which will have a great impact in losses and computational consumption of the switches. Alternatively, if we had initially allocated the flow h2-h7 in a different path, it would have been possible to allocate all the flows.

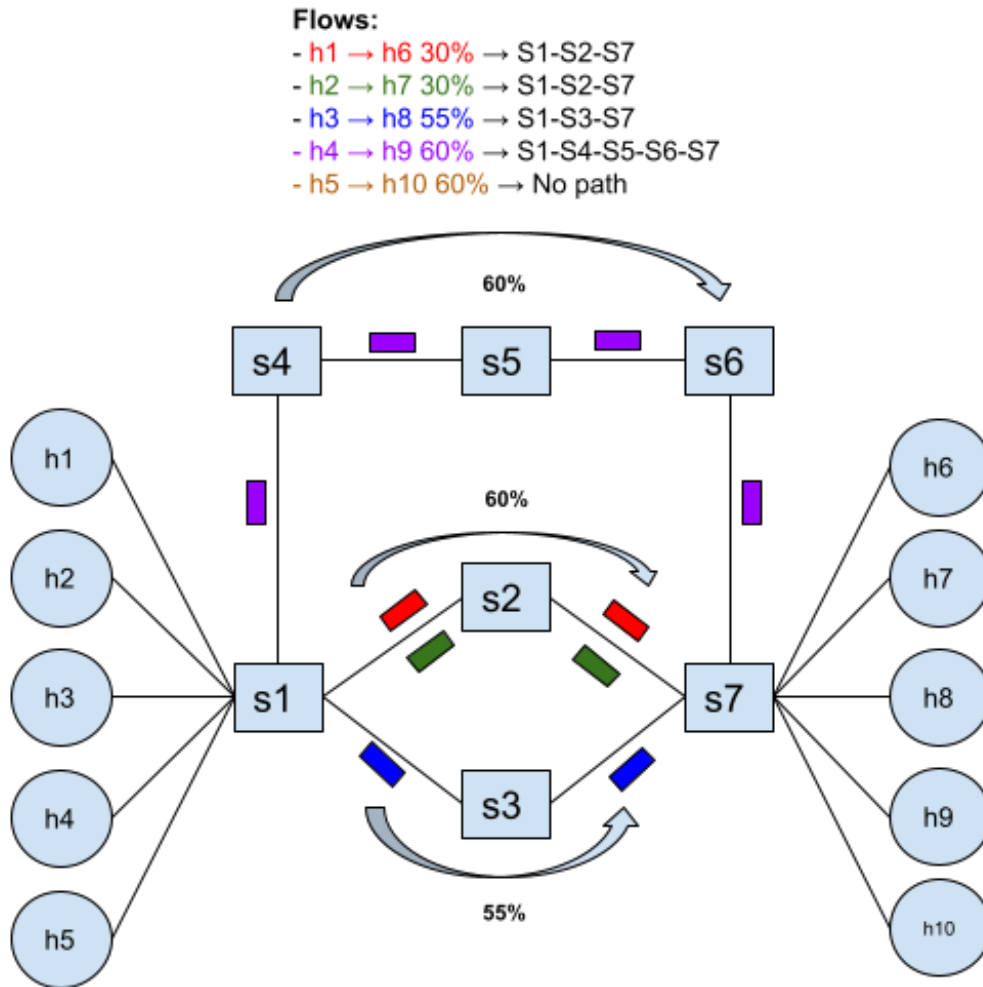


Fig. 3.23 Representation of the allocation problem using conventional routing algorithms

Having seen this quandary, it is when the necessity of RL comes into play, with the objective of taking these routing decisions based on a previously trained model to give us the best possible path also thinking in future network conditions. In the next chapter, we are going to see in detail how RL has been implemented for this project.

CHAPTER 4. REINFORCEMENT LEARNING

Having seen the experimental results for the routing using Dijkstra as implemented in ONOS and with the link weights based on link occupancies, now it is time to find a workaround to substantially improve the parameters we obtained in the last chapter. Taking this into account, in this fourth chapter we will see the basics of AI and RL to get familiar with them and see how they have been applied to this project. Furthermore, we will see all the processes from the definition and creation of the RL model to its adaptation to the final network topology. Finally, we will see a thorough comparison between RL, a LL approach based on the link occupancies, and SP. Before delving into the details of this chapter, it is important to succinctly explain why, as we will see, we have not trained the model in ONOS but in a simulated local environment. When we initially started working with RL and had a first model prototype, we started generating flows manually with MGEN to train the model, but it was virtually impossible given the resources available in our machine and the specifications thereof. Furthermore, the number of samples we needed to make the model learn was so high that it could have taken us years to train it.

4.1. Artificial Intelligence and Reinforcement Learning

The use of AI has seen an exponential growth over the last decade, as shown by this research about the number of patents registered [18]. Nowadays, it is being used in almost every sector we can imagine, being the telematic sector one of them. Based on its purpose, AI could be defined as the intelligence demonstrated by a machine capable of carrying out human tasks, such as perceiving and inferring information, by means of different techniques.

As the concept of AI is very broad, let's drill down to see where RL falls into this hierarchy. As can be seen in Fig 4.1, Machine Learning (ML) is a subcategory of AI and inside it contains three more subcategories, named Unsupervised Learning (UL), Supervised Learning (SL), and Reinforcement Learning.

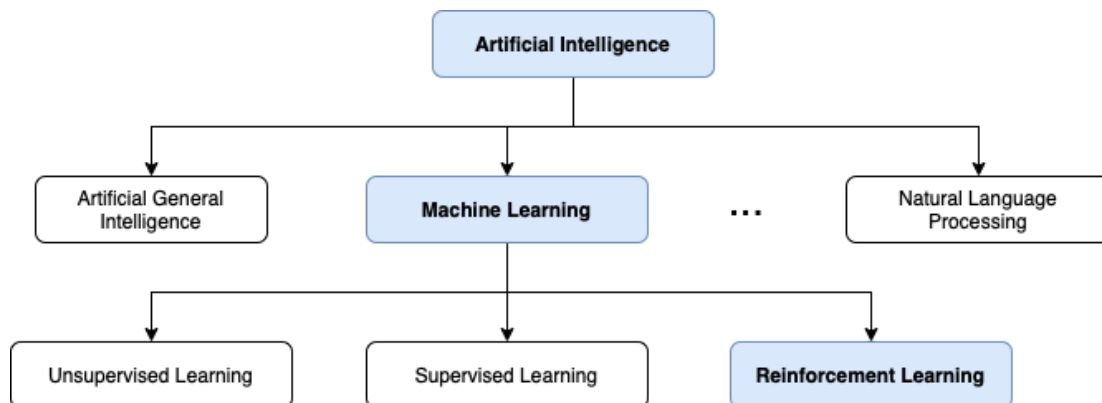


Fig. 4.1 AI and its different subcategories

Taking into consideration our objective of predicting which path to choose given an incoming flow based on the network state at that moment in time -this is the occupancy of the links-, we will need to use a technique that based on training data makes a prediction to get the most accurate result. This is why we have decided to use ML instead of the other techniques, which are meant to be used for other purposes. Analyzing each of the ML techniques, both UL and SL are very similar with the exception that they learn by trying to find patterns from unlabelled data (UL) and labeled data¹³ (SL) to train and test the model. The main setback of these techniques is the necessity of having a great amount of data from the very beginning in order to get great results. On the other hand, and simply put, RL does not need any beginning data as it learns from mistakes. Given that **in our scenario it is impossible to get thousands of samples about the state of the flows in the network**, we have decided to use RL.

Now that we have seen the reasoning behind the choosing of RL as our ML technique, let's explain thoroughly what RL is and the elements that it is composed of, which we will later translate into our environment in section 4.2. To begin with, RL can be defined as a ML technique in which the model that makes decisions, also known as **agent**, learns by its mistakes and its correct guesses while getting a positive result, known as **reward**, or a punishment. Drawing from this premise, we argue that the agent learns by interacting with the **environment** in different situations, known as the **states**. The decisions that are made by the agent at different moments in time are called **actions** and they will depend on the environment. In summary, the agent starts like a newborn and it will learn over time using a predefined **policy**. Even though RL is used widely in videogames and robotics, its dynamism and flexibility will allow us to adapt the agent to our network, as it does not need previously collected data for training purposes. Nevertheless, and as we will see in section 4.3, the experience the agent needs to achieve great results is staggering, following an exponential learning curve as the environment gets more and more difficult. Having explained this, let's now analyze in detail each of the highlighted terms aforementioned and some related (some of these terms are depicted in the agent process in Fig 4.2):

- First of all, the **agent** is the model that will interact with the environment, and has defined a series of neuronal layers and mathematical equations to train the data, being undoubtedly the most difficult part, as we will see in the next subsections.
- We understand the **environment** as the place in which the agent will take all of its actions and obtain the rewards from. It can be a real, physical world or a simulated environment.
- The **state** is defined as the snapshot of the current environment the agent is in. Sometimes it is not possible to represent all the possible states given the complexity to train the agent, and the granularity of the states will depend on the definition of the model. The complete list or set

¹³ In ML, labeled data is the one that has tags associated with it. For instance, if we want to predict whether the image contains a cat or a dog, the labels would be "cat" and "dog".

of states is known as **state space**, and the initial state of the environment is known as the **idle state**.

- The **action** is the act of making the conscious decision of which action to take, while actually executing it is called **step**. Likewise to the states, the list or set of actions is known as the **action space**, and will be the result based on the current state.
- The **reward** is the feedback obtained from the environment based on the action taken. Usually, but it might change, the correct guesses are rewarded with a positive value while the mistakes are punished with negative values.
- The **policy** is the methodology and algorithms used to translate the current state into an action. This is defined by the agent, and, as we will see, there are policies for taking actions and for training the model.
- Finally, when it comes to training the agent, an **episode** is considered all the interactions from the initial to the final state. This will be seen in more detail in section 4.3.

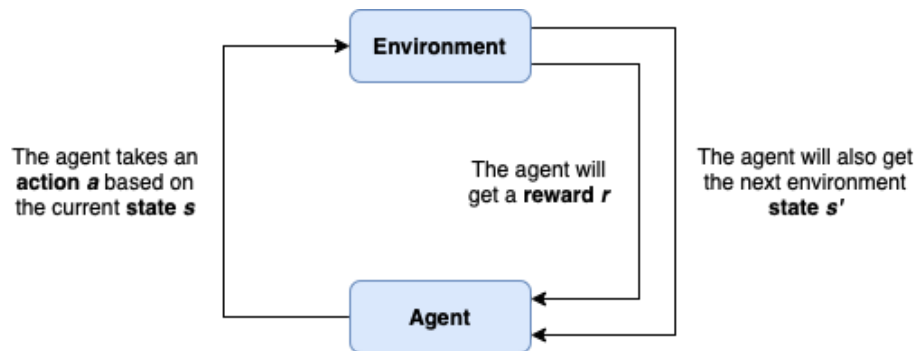


Fig. 4.2 RL agent and environment diagram

4.1.1. Epsilon-Greedy algorithm

When it comes to the methodology used for taking actions, it is always complex to determine which action to take and when. To better understand this intricacy, let's imagine an agent that begins from the scratch, and it does not have any experience and the state of the environment is the idle one. Firstly, based on the current state, the agent will have to take an action, and it will try to predict the correct action albeit its lack of experience, getting thus a reward. Next, the agent will follow the same flow of taking an action and getting a reward, and so on and so forth. As we can see, and especially at the beginning of the training, the agent needs to take random actions to see how they affect the environment, as well as to know all the actions there are at its disposal in order not to get stuck always in the same state. The decision of when to make a prediction based on the current experience is by means of the epsilon-greedy algorithm. Although there are alternative algorithms such as *soft epsilon-greedy* [19], this has proved to be the most efficient one for the majority of environments.

As we can see in Fig 4.3, the agent will choose a random option with probability ϵ , meaning that it will **explore** the environment, and choose the option which,

based on its current experience, seems the best one with probability $1 - \epsilon$, which is known as **exploiting** the environment. Along with the epsilon value, it is assigned an exponential epsilon decay, in order to almost always explore at the beginning and start exploiting the environment with an exponential probability as the number of training samples increases throughout the episodes, as can be seen in Fig 4.4.

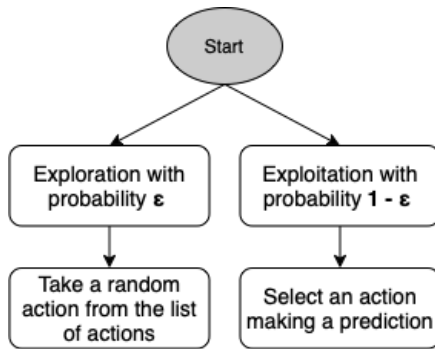


Fig. 4.3 Epsilon-greedy diagram

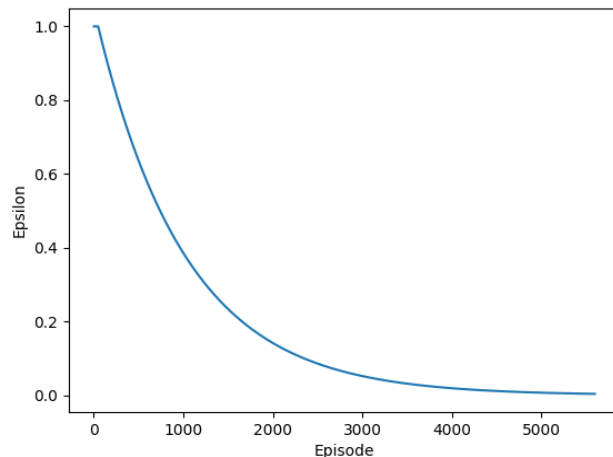


Fig. 4.4 Evolution of epsilon value throughout the episodes

4.1.2. Q-Learning algorithm

The goal of a RL model when training is to find the action that will yield the best reward given the actual state. For this, there are different algorithms such as SARSA [20] (one of the best alternatives right now) and Q-learning, but we have decided to use the second one mainly because right now is the most used and the complexity to implement it is less.

The goal of Q-learning is to find the optimal policy by learning the **optimal Q-value** for each state-action pair. This algorithm defines a table for each state-action pair and sets a Q-value, which will be zero at the beginning for all pairs. After each step (taking the action and getting a reward) the Q-values of this table will be updated. The agent, when trying to take an action from its experience, will actually look for the action with the highest Q-value in the table, which in the end is the most valuable state-action pair. Next, the Table 4.1 represents these previously mentioned pairs:

Table. 4.1 Q-Table with the Q-value for all the state-action pairs

	Action 1	Action 2	...	Action N
State 1	$Q(S_1, A_1)$	$Q(S_1, A_2)$...	$Q(S_1, A_N)$
State 2	$Q(S_2, A_1)$	$Q(S_2, A_2)$...	$Q(S_2, A_N)$
...
State M	$Q(S_M, A_1)$	$Q(S_M, A_2)$..	$Q(S_M, A_N)$

In order to understand how the agent updates them, we will need first to understand how the **Bellman equation** works (shown in 4.1). It tells us that the value of an action 'a' in a state 's', is the immediate reward the agent gets plus the maximum expected reward we can get in the next state. The importance of this expected reward is determined by the gamma factor γ (in section 4.3 we will see value for all these constants).

$$Q(s, a) = r + \gamma \cdot \max_{a'} Q(s', a') \quad (4.1)$$

Once we have calculated the value of the resulting Bellman equation, the algorithm will calculate the new Q-value, using the following equation:

$$Q_{new}(s, a) = (1 - \alpha) \cdot Q_{old}(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a')) \quad (4.2)$$

The **learning rate**, represented with the letter α , is a value between zero and one, and determines how quickly the agent discards the previous Q value to calculate a new one given a state-action pair. In other words, it states how much of the actual Q value information we keep for future Q values. Therefore, the higher the learning rate the quicker the agent will adopt the newly computed Q value.

4.2. Definition of the RL in our environment

Having seen all the theory of what RL is and the most common algorithms used with this technology, now it is time to see how we have adapted all the terms and values we have seen in the previous section to the environment we want to work with. First of all, let's explain the topology shown in Fig 4.5.

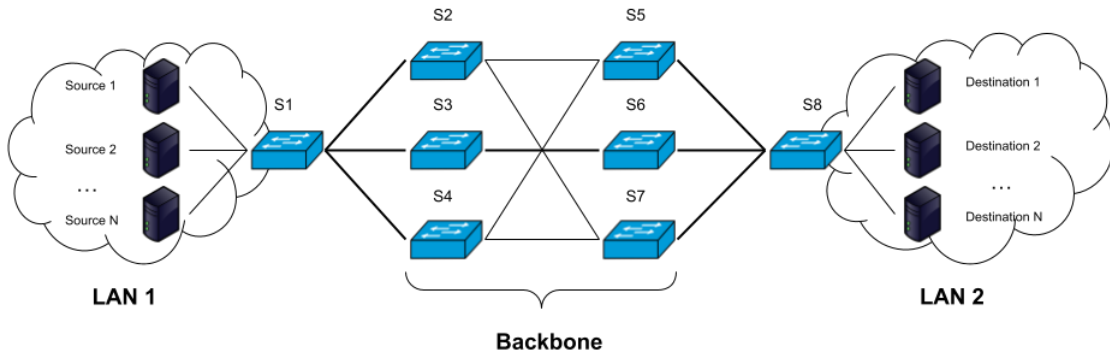


Fig. 4.5 RL topology used to train the agent

Our **environment** will have two LANs, being **LAN 1 the source** and **LAN 2 the destination**, with N servers connected to one and the other. The servers from LAN 1 will send flows of information with varying rates and not following a pattern between them, when it comes to the distribution in time of the flows, including its duration. There will be a total of **eight switches**, and a total of **eleven bidirectional links**. They will have a theoretical BW of 2 Mbps, except the ones with a thinner line in the image (S2-S5, S2-S7, S4-S5, and S4-S7), which will have a BW of 1 Mbps.

Having explained the topology for the scenario, let's see how we have defined the agent. In the field of AI, especially for ML, it is very common to build the agent we want to train using a **Deep Neural Network (DNN)**. These structures are formed by a series of layers with different neurons each, that are connected mimicking the way biological neurons signal to one another. DNNs have an input layer, an output layer, and a series of multiple hidden layers in between. Each neuron of the layer is connected to one or multiple neurons of the neighboring layers and has an associated weight. As you can imagine, the structure of these networks can be adapted to almost any kind of environment, no matter how difficult it is to train. Let's see how we have defined the structure of the agent in our scenario by explaining thoroughly the Fig 4.6.

As can be seen and as we saw in the previous section, from left to right the first thing we need to define is our state space so the agent can train with that data. In our scenario, it will be of **size 23** and will have only 1 dimension, resulting from the sum of the 22 different links of the network (11 bidirectional links) plus the rate of the incoming flow in the network. In summary, the RL agent will take into account the capacity of the links and the incoming rate to properly allocate the flows in the network. Having defined the state space, we find what is known as the **input layer**, which is defined as the outer incoming layer in a DNN. To define them, we have made use of the **Keras** library in Python, which allows us to create and train a DNN. If it is true that there are many different kinds of layers such as Convolutional, Flatten, Dropout, etc, we have decided to use the one that is regarded as the regular NN layer. The **dense** layer is defined by a series of neurons in which each of them is connected to each and every neuron of the next layer, forming a fully connected network. Furthermore, this layer uses what is known as an **activation function**, which determines the output

value of a neuron that will be communicated to the next layer. In our case, since we are defining multiple layers, it is recommended to use a **Rectified Linear Unit (ReLU)**, which converts every negative incoming value to zero and conserves the positive part of the argument, as shown in Fig 4.7.

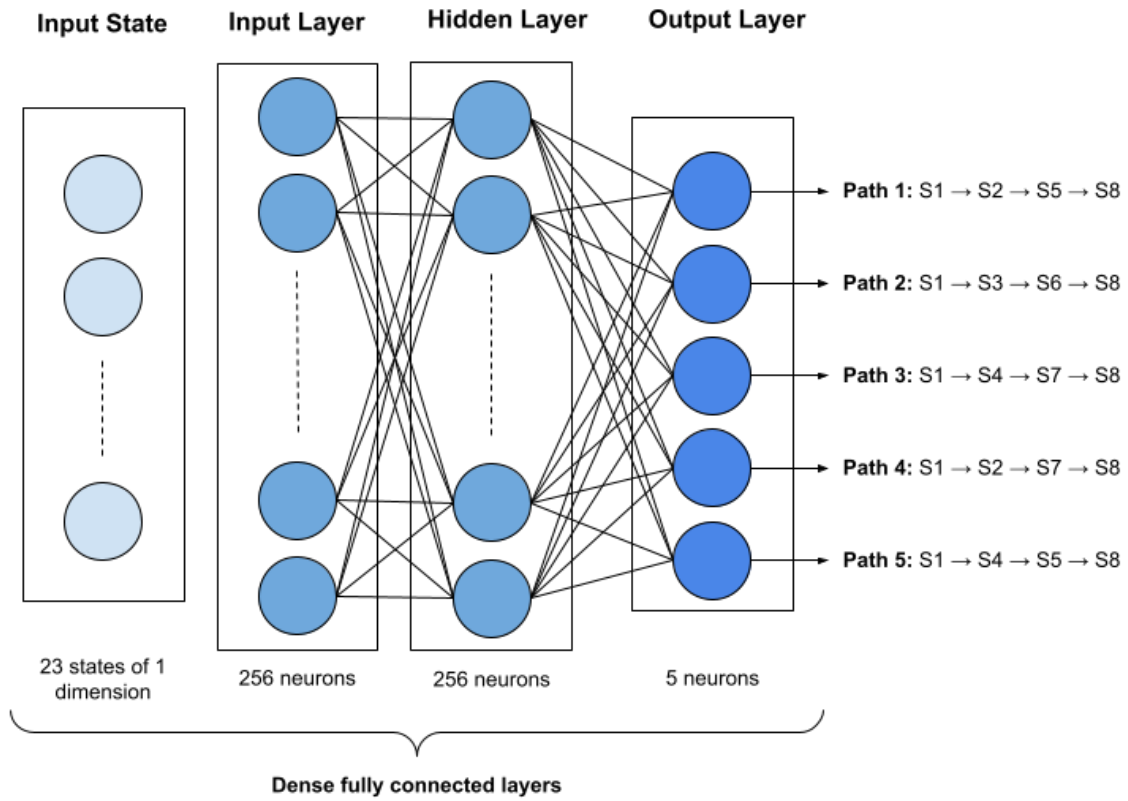


Fig. 4.6 DNN representation of the RL agent

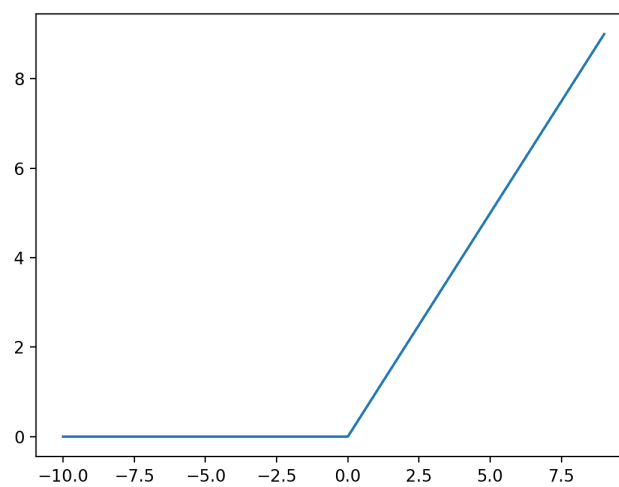


Fig. 4.7 Representation of the ReLU activation function

When it comes to the configuration of the second layer, we will again define **256 neurons** and a **ReLU** activation function. Finally, the output layer will determine the different outputs of the NN, which in our case will be the different paths between the LAN 1 and LAN 2. If it is true that there are more paths to go from S1 to S8, we have decided to just take the five most significant ones, being S1-S2-S5-S8, S1-S3-S6-S8, S1-S4-S7-S8, S1-S2-S7-S8, and S1-S4-S5-S8. Therefore, our **action space** will be of **size five**. Unlike the first two layers, the activation function for the output layer is of type **linear**, meaning that the output value will not be confined between any range. The reason to use this function at the output is because we are trying to predict a numerical variable (known as regression problems) and not a classification or categorical variable [21]. When it comes to the reward of the actions taken, we will compensate the model with a **one** if the allocated flow does not overflow any link, and a **zero** if it does (all of this will be seen in detail in section 4.3).

The last parameter of the agent that we are going to explain is the optimizer. They help improve the overall performance of a RL model. Every time the model is trained, it yields accuracy and loss values which are very indicative of the agent's performance. In summary, the optimizer has the objective of minimizing this loss using different mathematical equations. For this project, we will use the **Adam** optimizer [22], setting the learning rate (α) to 0.00005, as we will see in the next section.

4.3. Training of the RL model

Having seen what RL is and the adaptation to our environment, in this section we will show how the model has been trained to properly allocate the incoming flows in the network. First of all, it is important to mention that given the complexity to train these models when it comes to the staggering number of iterations to get decent results [23], we have decided to simulate the environment using Python instead of generating the flows directly with MGEN in ONOS.

Before getting into the essence of training, we need to understand first how a RL agent is trained. In every RL scenario, no matter which one, is needed to narrow down the environment into some kind of a game, defining not only an initial state as we saw in section 4.1 but also a final state, considered for the agent to be as a win. All the actions taken from the initial to the final state are called **episodes**, and are of high importance as far as training is concerned.

4.3.1. Training parameters

Having said this, we have generated a JavaScript Object Notation (JSON) training file containing a total of **1,000 flows** distributed randomly in a spatial time of **40,000 seconds** (on average a new flow every 40 seconds) and with a varying duration. Furthermore, in order for the agent to learn from as many different situations as possible and to allocate many different flows in one single link, the flow rate is limited between **80 and 248 Kbps** and is multiple of eight,

thus having a total of 22 different rates. The JSON file will contain all the necessary information to simulate real flows entering and exiting the network: the **hash**, the **time** the flow enters or leaves the network, whether it is **ingress or egress**, the **source** and **destination**, the **start** and **end** times, and finally the **rate**. Doing the adaptation of an episode in this context, we will set that when the flow exceeds the capacity of a link, the reward will be zero and thus the episode will end, and the environment will be reset to the initial state, and we will begin introducing the flows again in the same order. The final objective is that the agent learns how to properly allocate the 1,000 flows consecutively. In Table 4.2 is shown the structure with real values of the JSON file that contains the flows. The figures 4.8 and 4.9 show us the information of the flows that we have used to train the model. The average rate during the whole time frame of 40,000 seconds is of roughly **3,000 Kbps**, meaning that, on average, the network will be at **50% of its capacity** at all times. This value is obtained by considering that the three links of the switch S1 with S2, S3, and S4 are of 2 Mbps each, totalling 6 Mbps. We can also observe that the flows are very well distributed in terms of rate and the quantity of flows for each possible rate is fairly well distributed as well (Fig 4.9).

Table. 4.2 Excerpt of the JSON file containing the flows to train the RL agent

```
[
  {
    "hash": -3075843607397821400,
    "time": 0,
    "ingress": {
      "source": 1,
      "destination": 8,
      "start": 0,
      "end": 327,
      "rate": 144
    }
  },
  .....,
  {
    "hash": -3075843607397821400,
    "time": 327,
    "egress": {
      "source": 1,
      "destination": 8,
      "start": 0,
      "end": 327,
      "rate": 144
    }
  },
  .....
]
```

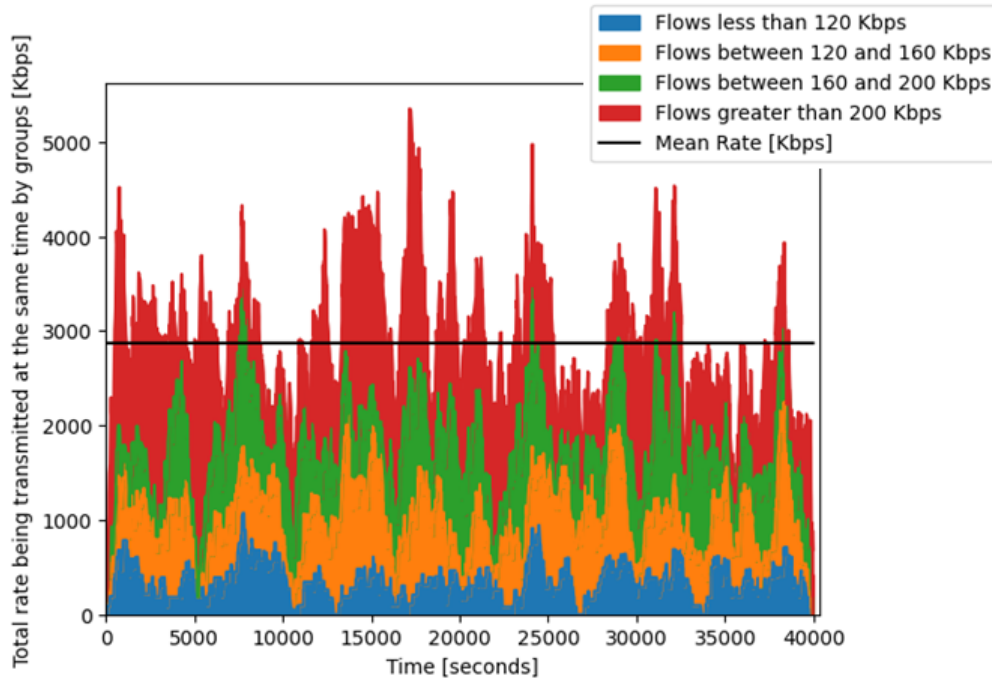


Fig. 4.8 Distribution in time of training flows grouped by rate

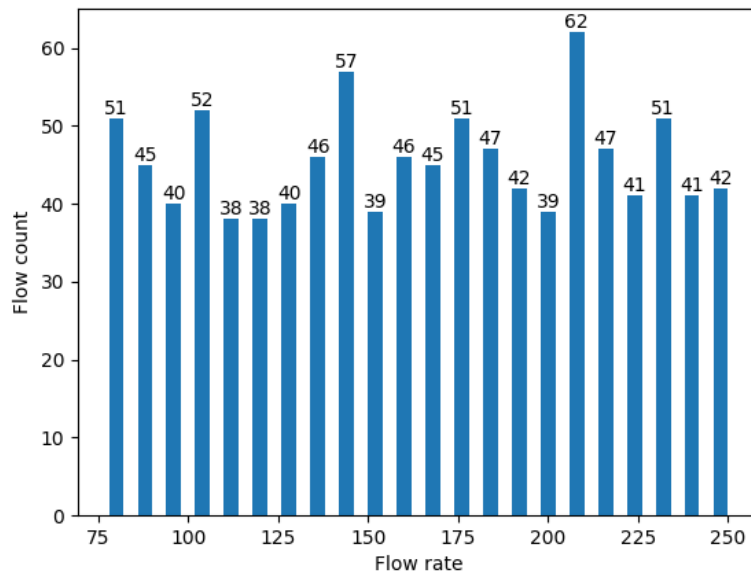


Fig. 4.9 Number of training flows grouped by rate

Now that we have a file of flows, we need to define all the parameters necessary to train a Deep Q-Learning (DQN) agent, and a couple extra related to the training process. In RL, these parameters are called **hyperparameters**, which means that they are used to control the learning process of the agent. Even though there are some techniques, for instance, to find an appropriate learning rate for a specific environment, it is extremely difficult to know in advance which are the right values to use. This is the main setback we have

faced during the training of the agent, turning the training into a trial-error exercise in most cases (in Appendix D there are some examples, not all, with different values to show this complexity). Knowing this difficulty and after trying many different combinations, we have concluded that the values set in Table 4.3 yield very promising results, as we will see shortly.

Table. 4.3 Hyperparameters used to train the RL model

State Space	23
Action Space	5
Learning Rate	0.00005
Epsilon Decay	0.999
Min Epsilon	0.001
Gamma	0.999
Replay Memory Size	10,000
Batch Size	128
Replay Start Learning Size	1,000
Maximum Number of Steps per Episode	1,000
Maximum Number of Episodes	10,000

- As we saw in section 4.2, the **state space** and **action space** will be 23 and 5, respectively.
- When it comes to the **learning rate**, we have decided to set a very low value of 0.00005, so as not to learn too quickly without gathering a lot of training data.
- When it comes to the epsilon-greedy algorithm parameters, we will start with an epsilon of one, and we will be decaying it at the end of each episode by a factor of 0.999, following the equation: 0.999^N , where N is the number of the episode. Apart from this, we will never set the epsilon to zero but leaving it at 0.001, to always, on average, explore the environment once every thousand iterations to see if there are some unseen states that the agent can learn from.
- In line with the learning rate, we will set a **gamma** value very close to 1 (0.999), so the agent can focus much more on future rewards and continue learning.
- When it comes specifically to the training parameters, we will only store a maximum of **10,000 training samples**. Out of these, we will randomly get **128** and train them using Q-learning (known as batch size). These samples will be composed of the *current state*, the *action taken*, the

reward, the *next state*, and whether the *episode has finished or not*. We will store them in a First in, First out (FIFO) mode, with the aim of not training with erroneous initial exploration samples after many iterations. Otherwise, if we had stored a million samples, the agent most likely would have never learnt.

- In order to have a minimum of samples before starting the training, we will explore the environment **1,000 times** (replay start learning size).
- Finally, there will be a total of **10,000 episodes**, and **1,000 iterations** (flows in our scenario) per episode.

In Fig 4.10, we can see the complete diagram of the process followed to train the agent. First of all, we will read all the parameters of the previous Table 4.3 from the `DQN_config.json` file, and generate the environment setting it to the initial state. After that, we will **create the DQN agent** with all its layers using Keras. From this point until the last episode, we will be reading each flow from the `flows_training.json` file in the order they appear, simulating the entrance in the network of real flows. If the flow is exiting the network (*“Is the flow entering the network?”* diamond in the diagram), we will update the network parameters related to the occupancy of the links. On the other hand, if the flow is entering the network, we will **explore or exploit based on the epsilon value** at that moment, and allocate the flow based on the chosen path (action). If it is possible to allocate the flow (*“Is it possible to allocate the flow?”* diamond in the diagram), we will update the occupancy of the links, give the agent a **reward of one**, and set the next state to the current state. Otherwise, if the agent has made a mistake and there is no option to allocate the flow, it will be given a **reward of zero**, and the episode will end.

Regardless of whether the flow can be allocated or not, we will have completed what is known as an **iteration**, and we will store the training sample we have aforementioned. In parallel with these iterations, once we have a total of **1,000 samples** the training process will begin. At the end of each episode get **128 samples** and **get the new Q-parameters** using the equation (4.2) seen in section 4.1. We will decrease the epsilon value by epsilon decay and start a new episode. To conclude, once we have gone through all episodes, the training will end and we will **save the DQN agent** so later we can use it to make predictions.

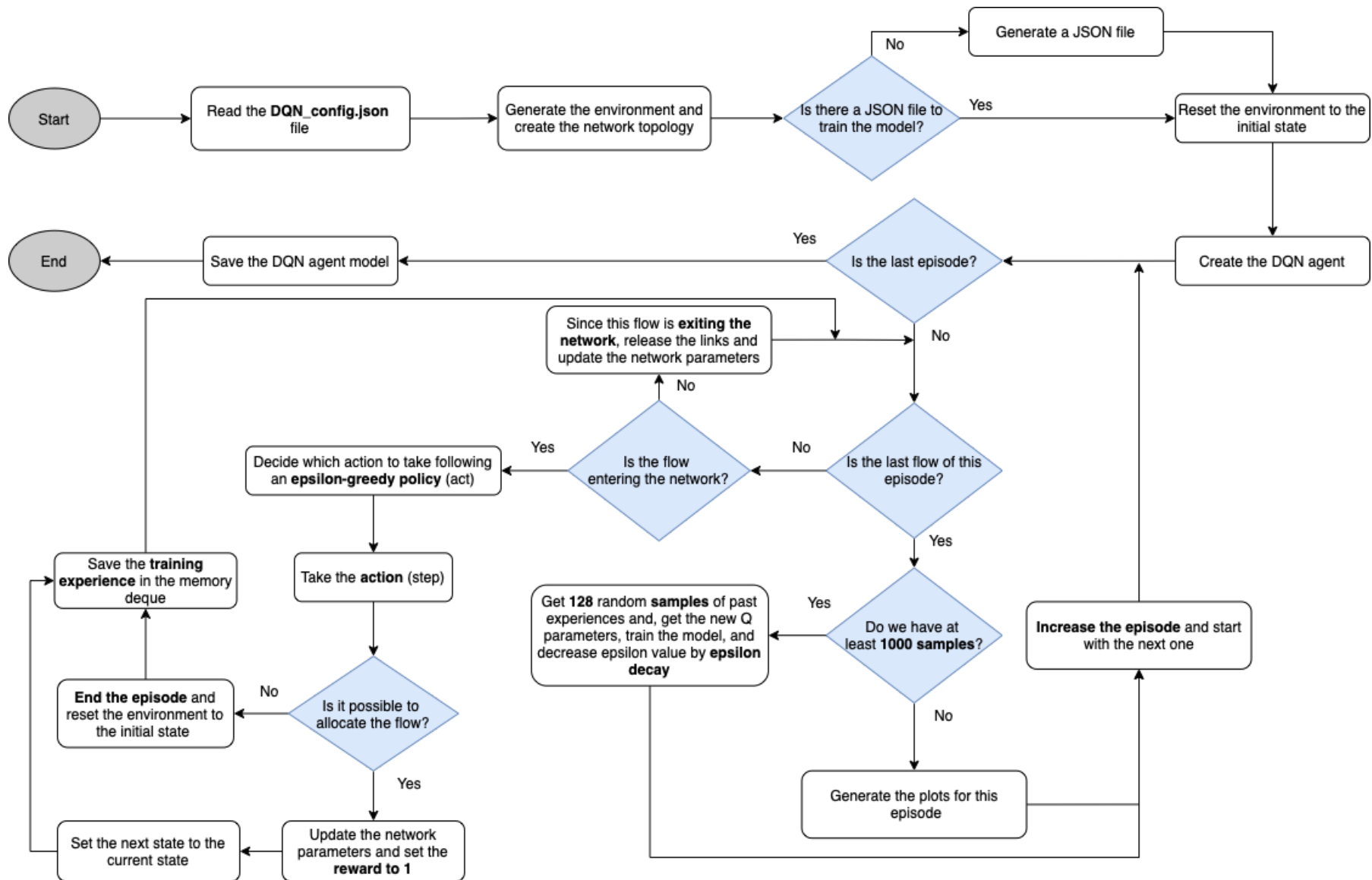


Fig. 4.10 Diagram to train the RL agent

4.3.2. Training results

Having seen the values set to the different parameters and the process of how the agent is trained, let's now see the results obtained. In order to thoroughly analyze the quality of the results, we have generated a total of three different graphs. The first one (Fig 4.11) shows us the total reward on average for the whole episode, which is obtained from summing all the rewards. Therefore, this chart is telling, from out of the 1,000 possible flows to allocate, how many the RL agent has routed properly in the network, without causing any link to exceed its capacity. To avoid very big fluctuations in the representation, we have smoothed the rewards using moving averages of 50, 100, 150, and 200, respectively. As can be seen, while the episodes start to increase and thus the epsilon value decreases, the obtained episode rewards are greater and greater. Thanks to the epsilon-greedy algorithm, the agent initially explores many different allocation of flows and can, therefore, learn progressively as the episodes go by. In the graph we can see that, approximately, at **episode 5,100 we get the best reward**, with an average in 50 episodes of **960 flows properly allocated**. We manually stopped the training earlier before finishing the episodes at 10,000 in order to avoid overfitting, something that we can see appearing as a tail in the blue 50 moving average, dropping the average reward substantially. In RL, if the agent exploits the environment excessively, it can end up with an excess of trained data, especially after having almost reached the maximum reward for an episode many consecutive times. In reference [24] this behavior is explained in detail.

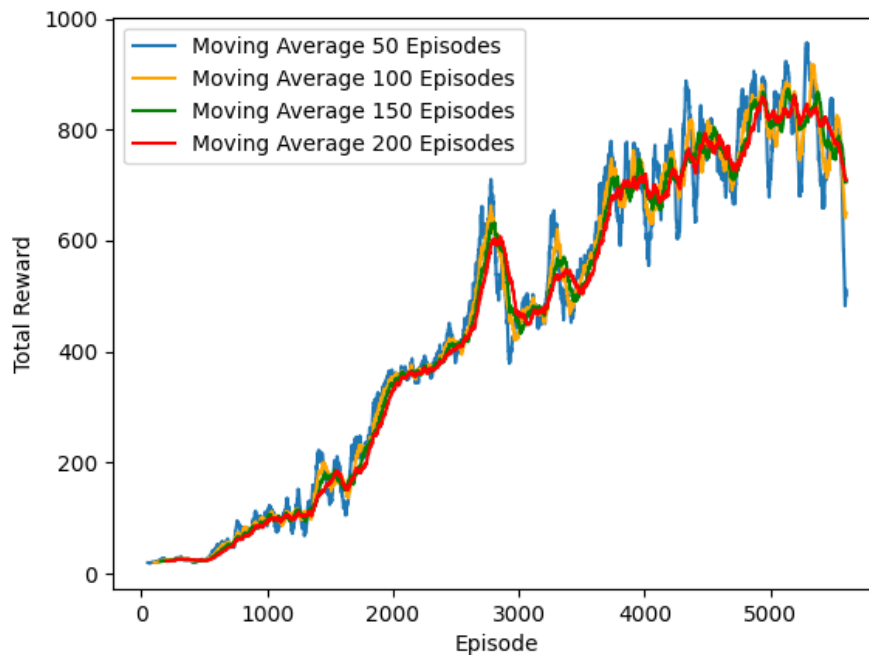


Fig. 4.11 Total episode reward with moving averages throughout the episodes

When it comes to the different paths taken and the occupancy of the links during the training process, we observe that the agent is doing very well as a load balancer. In Fig 4.12 we observe that **65% of the time** is taking the **path S1-S3-S6-S8**, mainly because all the links have a BW of 2 Mbps. The rest of

the time, the agent is choosing the other four paths almost the same number of times.

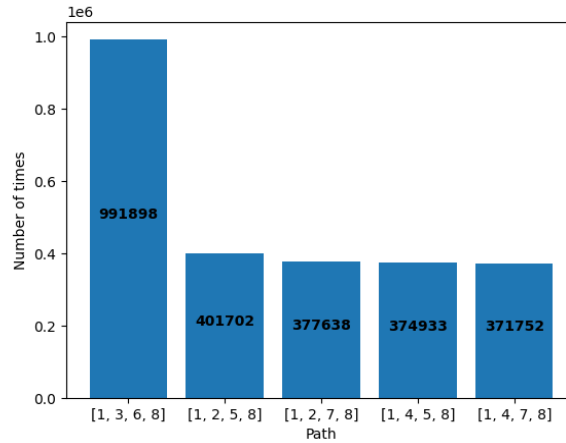


Fig. 4.12 Total number of times a path is chosen by the RL agent

In Fig 4.13, we can see what we have aforementioned. The flows are distributed almost as if we were working with a load balancer based on the occupancy of the links. This is very meaningful for the learning process of the agent, as it has been able to solve the dichotomy at switches S2 and S4 with two possible links to choose 1 Mbps each.

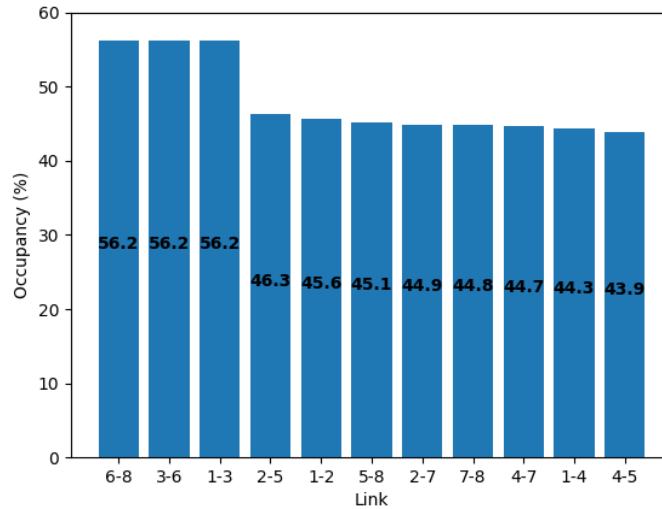


Fig. 4.13 Average link occupancy during the whole training period

4.4. Comparison between routing algorithms

Now that we have seen that the agent has learnt how to correctly allocate the flows, it is time to compare it against other routing algorithms and with different files of flows to see the quality of the RL model. Before entering into the details of how we have made the comparison, we will explain the two other algorithms, and the adaptation of RL for this comparison.

The diagram of Fig 4.14 shows how we have modified the RL training process seen in section 4.3 to compare it with other algorithms. One of the differences

now is that we will **load the already trained DQN model**, instead of creating a new one. Furthermore, if the flow is entering the network, **the agent will not use the epsilon-greedy algorithm**, instead it will always make a prediction (exploit) to decide which is the best path, rather than randomly select a path from time to time (explore). As expected, we will not train the agent anymore.

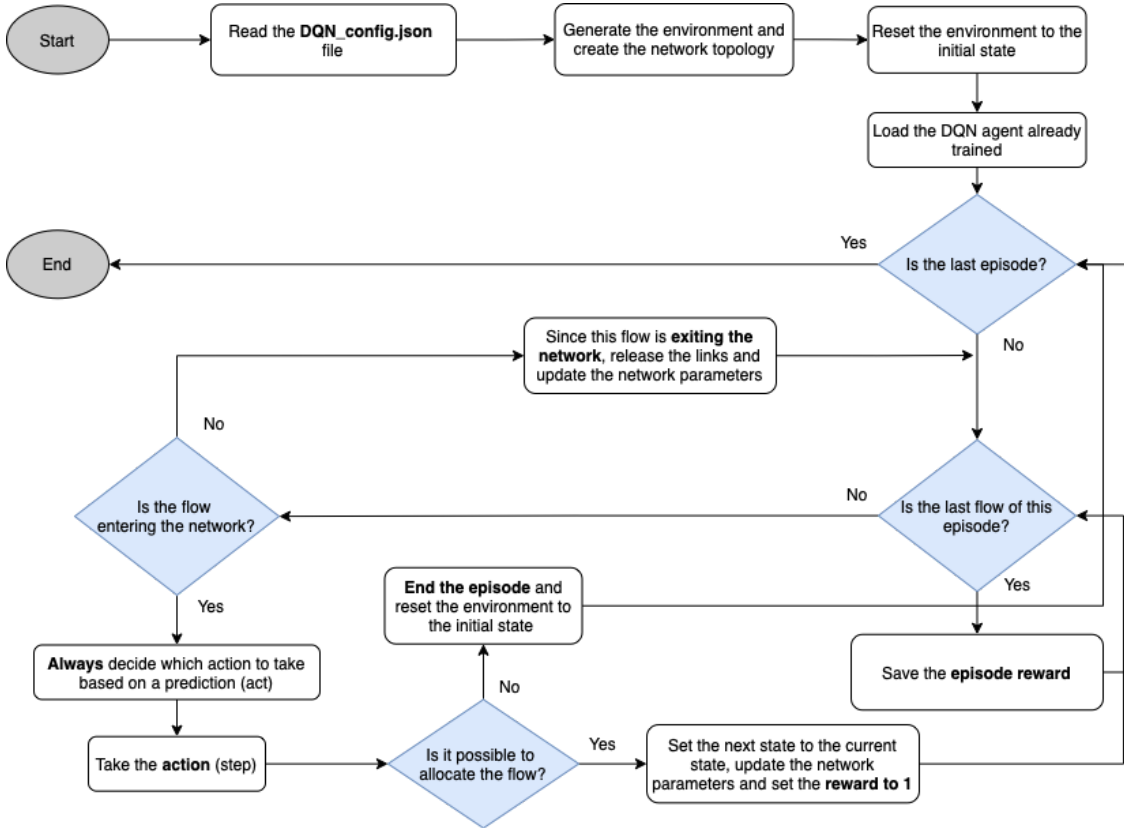


Fig. 4.14 Diagram to adapt RL to the comparison with other routing algorithms

The diagram of Fig 4.15 shows how we have implemented the LL algorithm based on the occupancy of the links. This time we will not use the RL agent to make any prediction, rather we will use a conventional routing algorithm. Once the environment is generated and the state is set to what would be an equivalent of an initial state in RL (all links empty), we will allocate the flows using LL. First of all, as we did for RL, if the flow is exiting the network we will just update the occupancy of the links. On the other hand, if the flow is entering the network we will **assign the weights of the links based on the occupancy in base one**. With this configuration, Dijkstra will return the least-loaded path, as the total path weight will be greater for those links with some data already going through them. In case there is more than one possible path (they have the same average occupancy), we will shuffle them and get a random one. This process will be repeated for every flow that enters the network.

Finally, the diagram of Fig 4.16 shows the implementation of the last algorithm, the SP. For this algorithm, the process will be the same as for the LL but we will adapt the algorithm part to find a path. We will always **set the weights of the links to one** and use Dijkstra to find the shortest path. Likewise, in the event of multiple shortest paths, we will mix them and select an arbitrary one.

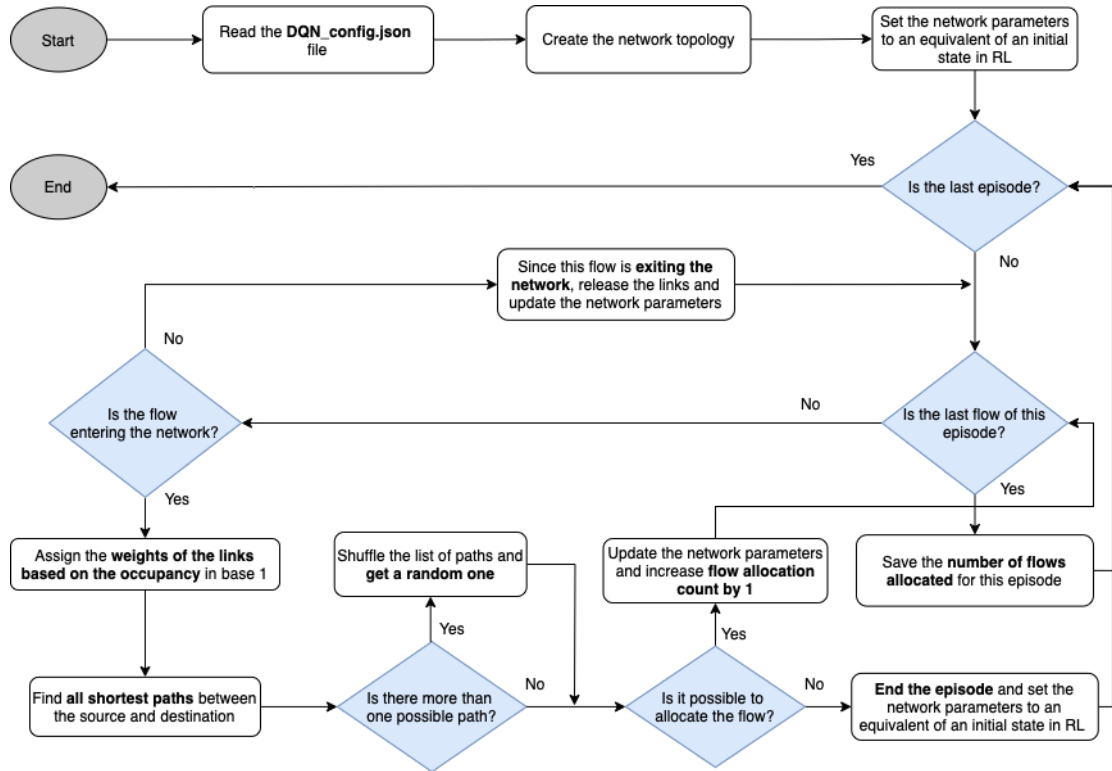


Fig. 4.15 Diagram to use the LL algorithm for the comparison

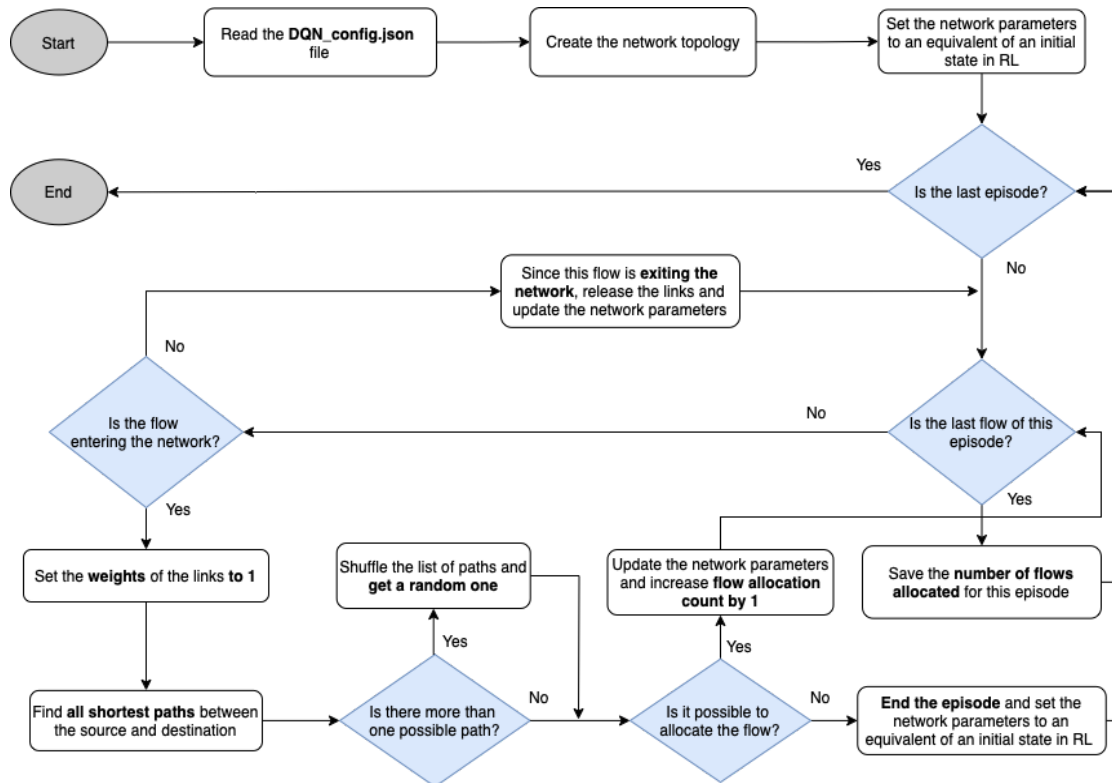


Fig. 4.16 Diagram to use the SP algorithm for the comparison

4.4.1. Comparison results

Having seen the adaption of RL and the different implementation diagrams of LL and SP, now we will see how we have made this comparison and the results thereof. We have generated a total of **5,000 comparison files with 1,000 flows each**, in the same way we generated the training file. Therefore, there will be a total of **five million of different potential flows** to allocate, a number high enough to avoid any kind of false good or bad result in terms of the figure of flows properly allocated (to see the tool in charge of creating these comparison files, please refer to Table C.10 from Appendix C). We have generated a total of four different graphs to depict the comparison between the algorithms.

In Fig 4.17 we can see the average number of flows properly allocated in the network for each of the three algorithms, after going through all the 5,000 files. Just as a reminder, the process followed to get these figures is the same as we did for the training of the agent. Every flow allocated properly without causing any link to overflow will count as one; otherwise, when the very first one exceeds, we will not introduce any more flow from that file and the count will stop. Therefore, the maximum theoretical average will be 1,000 once again. We can clearly see that our RL agent has managed to clearly beat the LL algorithm by a hefty **15.87%**, and the SP algorithm by **1,865.38%**. Even though we are far from the maximum average, since the other algorithms also have to allocate the same flows in the same conditions as RL, the number obtained is accurate and valid. In order to discard any outlier values that could have made this average to spike substantially, we have decided to plot a moving average of 50 comparison files. What we observe in Fig 4.18 is that almost every time the RL values are above the LL ones, or at most having the same moving average, but **never below**. This proves that not only the RL agent is better, on average, but also is very constant across the different files, yielding very promising results.

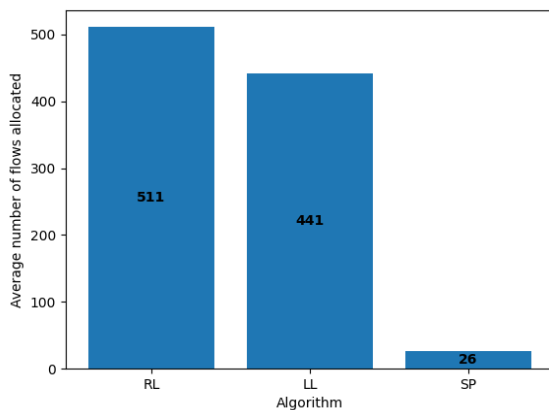


Fig. 4.17 Average number of flows allocated for the comparison of the algorithms

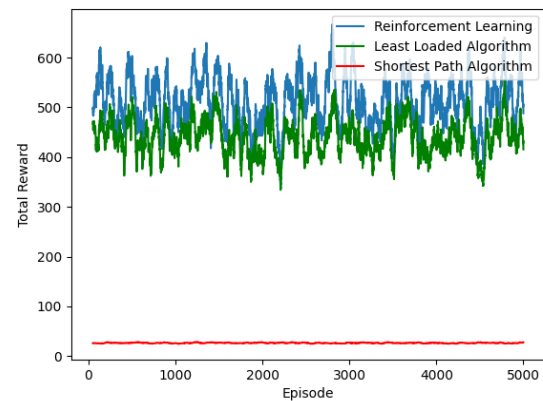


Fig. 4.18 Moving average of 50 comparison files for the comparison of the algorithms

When it comes to the different paths taken and the occupancy of the links during the comparison process, we observe in Fig 4.19 that the agent is taking the **path S1-S3-S6-S8 39% of the time**, mainly because all the links have a BW of 2 Mbps. The rest of the time, the agent is choosing the other four paths almost the same number of times, except for the path S1-S4-S7-S8 that is taken 12% of the time. Similarly, in Fig 4.20 we can see what was initially introduced in this project. The agent is doing better as a load balancer than the LL algorithm, due to the fact that the RL agent already learned when training to allocate the flows thinking of the future impact these decisions might have.

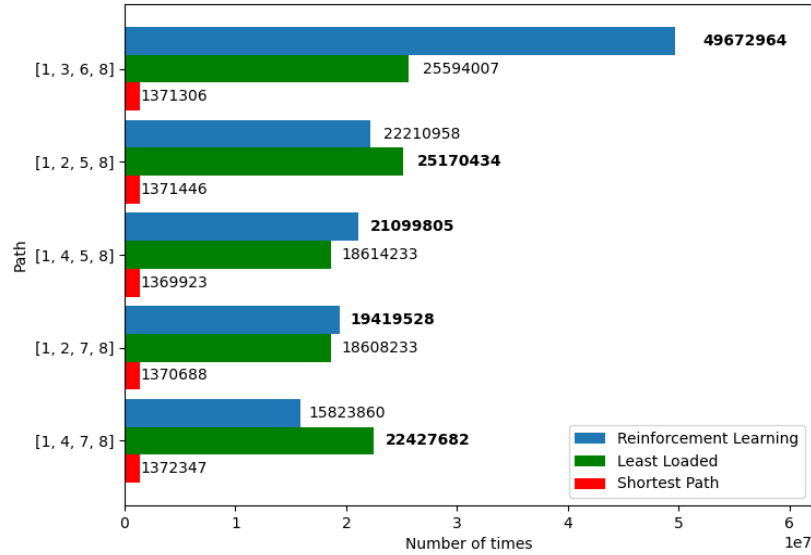


Fig. 4.19 Total number of times a path is chosen by each comparison algorithm

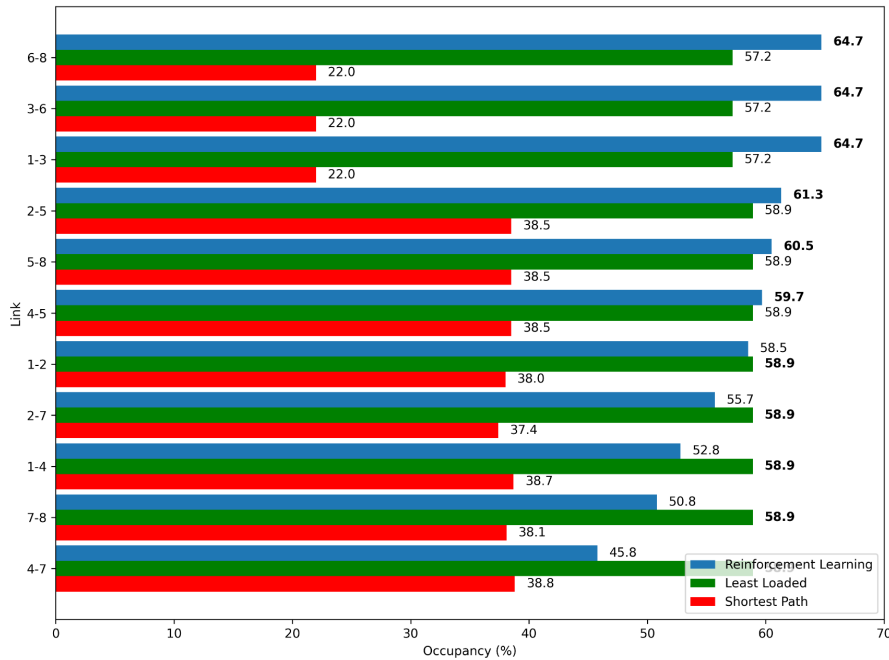


Fig. 4.20 Average link occupancy during the whole comparison period by each algorithm

CONCLUSIONS

It is undeniable that the use of the Internet is still growing almost exponentially across the world, especially when it comes to the use of AI and all its subcategories, and RL is not an exception. Throughout this project we have seen what SDN is, how the most used open-source controller (ONOS) is adapted and how it behaves to work with SDN, and we have also seen that is possible, training a RL model, that we can consistently beat other traditional routing algorithms such as LL based on link occupancies and SP.

The main objective of this project was to train a Neural Network model using RL to make routing decisions when new flows want to enter the network. This way, the model would not take into account only the rate of the incoming traffic but also it will analyze how future traffic might be affected by this decision. As we have seen in the fourth section of this project, we have successfully achieved our initial goal of automating routing decisions and creating a model that is better when compared to other routing algorithms.

First of all, we defined the architecture of the RL agent using a DNN and then later trained it using the Q-learning algorithm. With this training we accomplished that at least the agent was able to properly allocate one thousand flows, on average, proving that the agent can learn to assign a path to flows. Secondly, to actually demonstrate that the trained model adapts to different combinations of flows, we compared it against two well-known routing algorithms, such as LL and SP. We proved that RL was routing better traffic by roughly 16% when compared against its direct competitor, the LL.

I am very honored to have worked on this project, especially because I learnt lots of different concepts about not only RL but also SDN and ONOS. Furthermore, I would like to outline what could be the next steps, as I firmly believe the agent could be improved more:

- Firstly, I would load this already trained RL agent into ONOS, just calling it from the *ReactivePacketProcessor* to get a path once the flow wants to enter the network. As was introduced at the beginning of chapter 4, for the aforementioned circumstances we had to work with a simulated environment. Now that the model is ready, it could be used by our routing-app ONOS application.
- Secondly, I would try to improve the learning process of the agent by increasing the nodes per NN layer or maybe using a higher batch size to use more training samples when training. Another option could be to train the model using more than one training file, and randomly switching the set of files (preferably just about ten of them) throughout the episodes.
- Thirdly, I would try to compare it to another routing algorithm, to see how well it behaves against more algorithms.
- Finally, as we have seen in chapter 4, the training of the RL algorithm has been made always having the same source and destination, which could limit the implementation of more real scenarios. The next step

would be to first adapt the model to consider flows from N sources to only one destination. In this case, we would need to somehow punish the model if the chosen path is not a feasible one. Other options could involve increasing the complexity of the NN, adding more nodes per layer or even adding an additional hidden layer, so the model can work with far more different combinations and work out more different scenarios. However, the learning curve would not be lineal and we would need even more samples to properly train the model. This could be achieved by spending more time exploring the environment (epsilon-greedy) and giving more importance to future rewards.

REFERENCES

- [1] DataReportal, "Digital Around the World — DataReportal – Global Digital Insights," *DataReportal – Global Digital Insights*, 2023.
<https://datareportal.com/global-digital-overview>
- [2] Market Research Report, "Software-Defined Networking Market by Component (SDN Infrastructure, Software, and Services), SDN Type (Open SDN, SDN via Overlay, and SDN via API), End User, Organization Size, Enterprise Vertical, and Region - Global Forecast to 2025," *MarketsandMarkets*, Jul. 2020.
<https://www.marketsandmarkets.com/Market-Reports/software-defined-networking-sdn-market-655.html>
- [3] Imperva, "What is OSI Model," *Imperva Inc.*
<https://www.imperva.com/learn/application-security/osi-model/>
- [4] Open Networking Foundation, "OpenFlow Switch Specification Version 1.5.1," Mar. 2015. [Online]. Available:
<https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf>
- [5] Sophos, "DSCP Value," *Sophos Firewall Manager Help*, 2018.
<https://docs.sophos.com/nsg/sophos-firewallmanager/v17.0.0/Help/en-us/webhelp/onlinehelp/index.html#page/onlinehelp/DSCPValue.html>
- [6] NOX Repo, "The NOX Controller," *GitHub*. <https://github.com/noxrepo/nox>
- [7] POX Repo, "The POX network software platform," *GitHub*.
<https://github.com/noxrepo/pox>
- [8] Atlassian, "Floodlight Controller," *Project Floodlight*.
<https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview?homepageld=1343545>
- [9] The Linux Foundation Projects, "Home," *OpenDaylight*.
<https://www.opendaylight.org/>
- [10] Open Networking Foundation, "Open Network Operating System (ONOS) SDN Controller for SDN/NFV Solutions," *Open Networking Foundation*.
<https://opennetworking.org/onos/>

- [11] Karanatsios, "Full-Stack OpenFlow Framework in Ruby," *GitHub*.
<https://github.com/trema/trema>
- [12] ONOS, "ONOS Developers," *Google Groups*.
<https://groups.google.com/u/1/a/onosproject.org/g/onos-dev>
- [13] Oracle, "What Is an Interface? (The Java™ Tutorials > Learning the Java Language > Object-Oriented Programming Concepts)." Learning
<https://docs.oracle.com/javase/tutorial/java/concepts/interface.html>
- [14] ONOS Project, "TrafficSelector.Builder (ONOS Java API (2.4.0))," Jun. 05, 2020.
<https://api.onosproject.org/2.4.0/apidocs/org/onosproject/net/flow/TrafficSelector.Builder.html>
- [15] ONOS Project, "TrafficTreatment.Builder (ONOS Java API (2.4.0))," Jun. 05, 2020.
<https://api.onosproject.org/2.4.0/apidocs/org/onosproject/net/flow/TrafficTreatment.Builder.html>
- [16] Geeks for Geeks, "Tarjan's Algorithm to find Strongly Connected Components," *GeeksforGeeks*, Aug. 20, 2014. [Online]. Available:
<https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/>
- [17] Burning Bright, "Suurballe's algorithm," *BurningBright*, Jan. 31, 2018.
<https://www.linchenguang.com/2018/01/31/Suurballe-s-algorithm/>
- [18] Our World in Data, "Annual patent filings for artificial intelligence technologies globally," *Our World in Data*.
<https://ourworldindata.org/grapher/number-artificial-intelligence-patent-filings>
- [19] Lazy Programmer, "Difference between epsilon-greedy and epsilon-soft policies," *Lazy Programmer*, Feb. 27, 2020.
<https://lazyprogrammer.me/what-is-the-difference-between-epsilon-greedy-and-epsilon-soft-policies/>
- [20] D. S, "Reinforcement Learning with SARSA — A Good Alternative to Q-Learning Algorithm," *Towards Data Science*, Oct. 19, 2022. [Online]. Available:
<https://towardsdatascience.com/reinforcement-learning-with-sarsa-a-good-alternative-to-q-learning-algorithm-bf35b209e1c>

- [21] J. Brownlee, "How to Choose an Activation Function for Deep Learning," *MachineLearningMastery.com*, Jan. 18, 2021.
<https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/>
- [22] J. Brownlee, "Gentle Introduction to the Adam Optimization Algorithm for Deep Learning," *MachineLearningMastery.com*, Jul. 03, 2017.
<https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>
- [23] OpenAI, "OpenAI Baselines: DQN," May 24, 2017.
<https://openai.com/research/openai-baselines-dqn>
- [24] C. Zhang, O. Vinyals, R. Munos, and S. Bengio, "A Study on Overfitting in Deep Reinforcement Learning," Apr. 2018.

APPENDIX A. ENVIRONMENT CONFIGURATION

In this appendix, it is explained thoroughly how to install each and every tool necessary to run ONOS as well as the configuration of some environment variables. Besides that, it is also explained how to install complementary tools to make the coding and understanding of the controller easier.

A.1. Installing Maven (3.6.3) and Karaf Runtime (4.2.9)

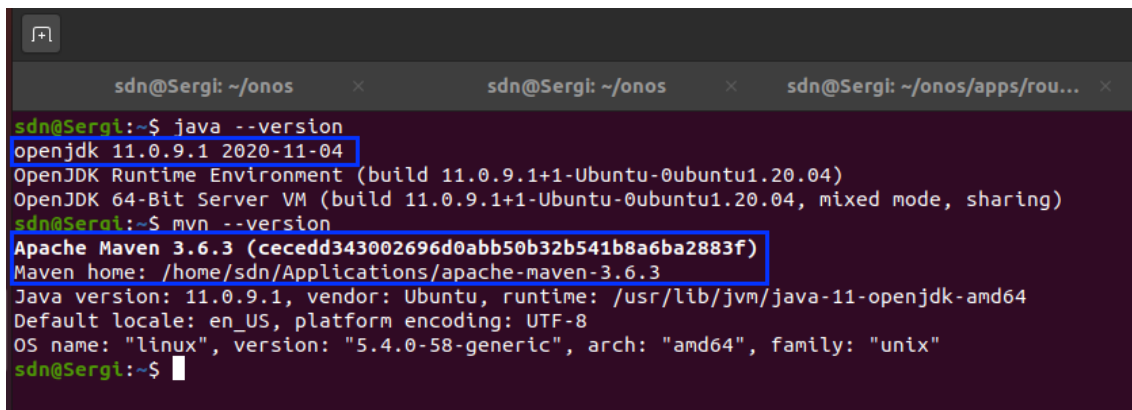
```
$ cd ~
$ mkdir Applications
$ cd Downloads
$ wget http://archive.apache.org/dist/karaf/4.2.9/apache-karaf-4.2.9.tar.gz
$ wget
http://archive.apache.org/dist/maven/maven-3/3.6.3/binaries/apache-maven-3.6
.3-bin.tar.gz
$ tar -zxvf apache-karaf-4.2.9.tar.gz -C ../Applications
$ tar -zxvf apache-maven-3.6.3-bin.tar.gz -C ../Applications
$ cd Applications
$ chmod +x apache-karaf-4.2.9
$ chmod +x apache-maven-3.6.3
```

A.2. Installing Oracle Java 11

```
$ cd ~
$ sudo apt update
$ sudo apt install openjdk-11-jdk
$ sudo nano .bashrc
-   export JAVA_HOME=/usr/lib/jvm/java-1.11.0-openjdk-amd64
$ source .bashrc
```

A.2.1. Making *mvn* command global

```
$ cd ~
$ sudo nano /etc/profile.d/maven.sh
-   export JAVA_HOME=/usr/lib/jvm/java-1.11.0-openjdk-amd64
-   export M2_HOME=/home/sdn/Applications/apache-maven-3.6.3
-   export MAVEN_HOME=/home/sdn/Applications/apache-maven-3.6.3
-   export PATH=${M2_HOME}/bin:${PATH}
$ source /etc/profile.d/maven.sh
$ sudo ln -s /home/sdn/Applications/apache-maven-3.6.3
/home/sdn/Applications/maven
```



```

sdn@Sergi: ~/onos
sdn@Sergi:~$ java --version
openjdk 11.0.9.1 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.04)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-Ubuntu-0ubuntu1.20.04, mixed mode, sharing)
sdn@Sergi:~$ mvn --version
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: /home/sdn/Applications/apache-maven-3.6.3
Java version: 11.0.9.1, vendor: Ubuntu, runtime: /usr/lib/jvm/java-11-openjdk-amd64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "5.4.0-58-generic", arch: "amd64", family: "unix"
sdn@Sergi:~$

```

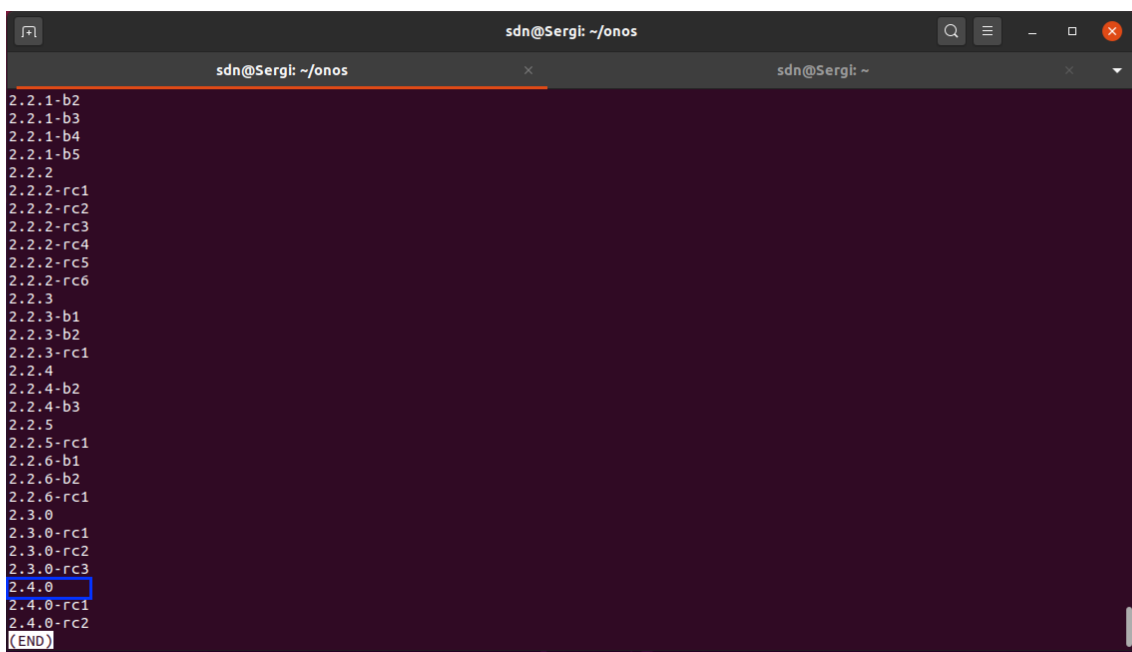
Fig. A.1 Maven installation verification

A.3. Installing ONOS Uguisu (2.4.0)

```

$ cd ~
$ git clone https://gerrit.onosproject.org/onos
$ cd onos
$ git tag

```



```

sdn@Sergi: ~/onos
sdn@Sergi:~$ git tag
2.2.1-b2
2.2.1-b3
2.2.1-b4
2.2.1-b5
2.2.2
2.2.2-rc1
2.2.2-rc2
2.2.2-rc3
2.2.2-rc4
2.2.2-rc5
2.2.2-rc6
2.2.3
2.2.3-b1
2.2.3-b2
2.2.3-rc1
2.2.4
2.2.4-b2
2.2.4-b3
2.2.5
2.2.5-rc1
2.2.6-b1
2.2.6-b2
2.2.6-rc1
2.3.0
2.3.0-rc1
2.3.0-rc2
2.3.0-rc3
2.4.0
2.4.0-rc1
2.4.0-rc2
(END)

```

Fig. A.2 List of ONOS versions

```

$ git checkout -b 2.4.0 2.4.0
$ cd ..
$ sudo nano .bashrc

```

```

- export ONOS_ROOT=~/.onos
- ~/.ONOS_ROOT/tolos/dev/bash_profile
$ source .bashrc

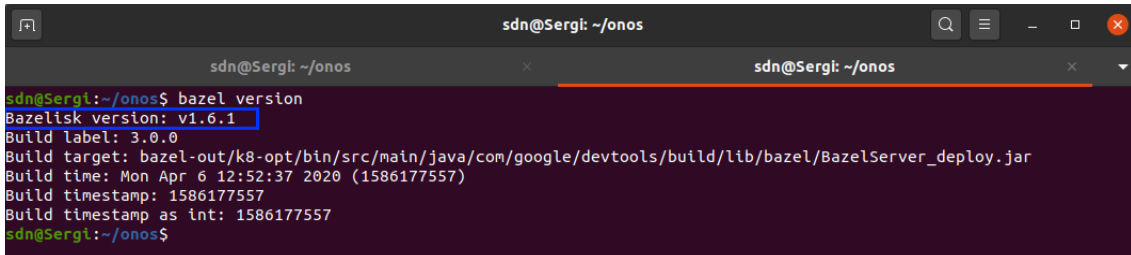
```

A.4. Building ONOS using Bazel (1.6.1)

```

$ cd ~
$ wget
https://github.com/bazelbuild/bazelisk/releases/download/v1.6.1/bazelisk-linux-a
md64
$ chmod +x bazelisk-linux-amd64
$ sudo mv bazelisk-linux-amd64 /usr/local/bin/bazel

```



```

sdn@Sergi: ~/.onos
sdn@Sergi: ~/.onos
sdn@Sergi:~/.onos$ bazel version
Bazelisk version: v1.6.1
Build label: 3.0.0
Build target: bazel-out/k8-opt/bin/src/main/java/com/google/devtools/build/lib/bazel/BazelServer_deploy.jar
Build time: Mon Apr 6 12:52:37 2020 (1586177557)
Build timestamp: 1586177557
Build timestamp as int: 1586177557
sdn@Sergi:~/.onos$

```

Fig. A.3 Bazel installation verification

A.4.1. Installing gcc compiler (9.3.0), python2, python3, and pip

In order to build and run ONOS correctly, we will need to install the gcc compiler python2, and set python3 as the default version.

```

$ cd ~
$ sudo apt update
$ sudo apt install build-essential
$ sudo apt install manpages-dev
$ sudo apt install python2
$ sudo update-alternatives --install /usr/bin/python python /usr/bin/python2 1
$ sudo update-alternatives --install /usr/bin/python python /usr/bin/python3 2

```

When introducing the below command, we will be requested to type a selection number or keep the current choice by pressing the enter keyboard (see Fig. A. 4). In our case, we will choose the number 1 as the first priority.

```

$ sudo update-alternatives --config python
$ sudo add-apt-repository universe
$ sudo apt update
$ curl https://bootstrap.pypa.io/get-pip.py --output get-pip.py
$ sudo python2 get-pip.py
$ sudo apt install python3-pip

```



```

sdn@Sergi:~$ gcc --version
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

sdn@Sergi:~$ sudo update-alternatives --config python
There are 2 choices for the alternative python (providing /usr/bin/python).

  Selection    Path                        Priority  Status
  -----
  0            /usr/bin/python3            2        auto mode
* 1            /usr/bin/python2            1        manual mode
  2            /usr/bin/python3            2        manual mode

Press <enter> to keep the current choice[*], or type selection number:
sdn@Sergi:~$ pip --version
pip 20.2.4 from /usr/local/lib/python2.7/dist-packages/pip (python 2.7)
sdn@Sergi:~$

```

Fig. A.4 Bazel's complementary tools verification

After having executed all the above commands, by using the python *pip* tool, all the necessary packages will be installed to run both the RL scripts and those of the monitoring tools. Since future versions after the writing of this project could have changed or deleted part of their methods, it is advisable to install the exact versions, especially *keras*, *tensorflow*, and *networkx*. Under the home directory, run the following commands.

```

$ cd ~
$ pip3 install h5py==2.10.0 Keras==2.4.3 networkx==2.5.1 numpy==1.19.5
scipy==1.4.1 tensorflow==2.4.1 scikit-learn==0.24.1 pandas==0.25.3
influxdb==5.3.1 mininet==2.3.0dev6

```

A.4.2. Changing ONOS IP localhost for out private address

By default, ONOS is configured to run in the localhost address (127.0.0.1). In our case, since the virtual machine has two network adapters, one attached to Network Address Translator (NAT) and the other attached to *Host-only Adapter*, we will need to set the address used by the latter adapter (see Fig. A. 5). Knowing the private address, we will need to set this address in the **onos-run-karaf** file (see Fig. A. 6).

```

sdn@Sergi: ~
sdn@Sergi:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255
    inet6 fe80::4c9b:f80e:7d0e:b86c prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:d1:fb:ce txqueuelen 1000 (Ethernet)
    RX packets 225770 bytes 319214345 (319.2 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 16859 bytes 1936502 (1.9 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.99.106 netmask 255.255.255.0 broadcast 192.168.99.255
    inet6 fe80::feea:49e5:ff00:6f75 prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:20:8e:d7 txqueuelen 1000 (Ethernet)
    RX packets 12 bytes 4142 (4.1 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 108 bytes 12079 (12.0 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 1352 bytes 141581 (141.5 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1352 bytes 141581 (141.5 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

sdn@Sergi:~$

```

Fig. A.5 Network interfaces

```

$ cd ~
$ sudo nano $ONOS_ROOT/tools/package/onos-run-karaf
- IP=${ONOS_IP:-192.168.99.106}

```

```

sdn@Sergi: ~
GNU nano 4.8 onos/tools/package/onos-run-karaf
# Blitz previously unrolled onos- directory
rm -fr $ONOS_DIR

# Unroll new image from the specified tar file
[ -f $ONOS_TAR ] && tar xzf $ONOS_TAR -C /tmp

# Write out this installation's MD5 checksum
echo "$newMD5" > $ONOS_MD5

# Run using the secure SSH client
[ ! -f ~/.ssh/id_rsa.pub ] && ssh-keygen -t rsa -f ~/.ssh/id_rsa -P '' -q
$ONOS_DIR/bin/onos-user-key $(id -un) "$(cut -d\ -f2 ~/.ssh/id_rsa.pub)"
$ONOS_DIR/bin/onos-user-password onos rocks

# Create config/cluster.json (cluster metadata)
IP=${ONOS_IP:-192.168.99.106}
echo "Creating local cluster configs for IP $IP..."
[ -d $ONOS_DIR/config ] || mkdir -p $ONOS_DIR/config
cat > $ONOS_DIR/config/cluster.json <<-EOF
{
  "name": "default-$RANDOM",
  "node": {
    "id": "$IP",
    "ip": "$IP",
    "port": 9876
  }
}

```

Fig. A.6 Onos-run-karaf configuration file

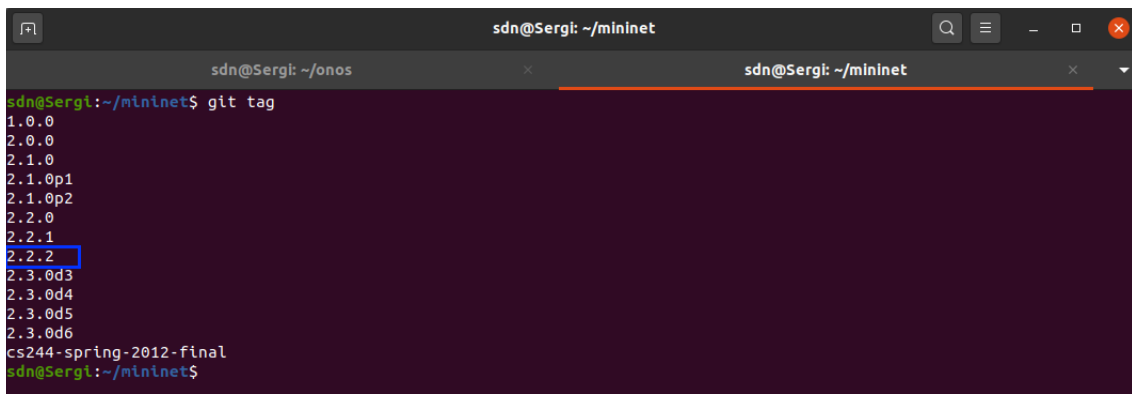
```
$ cd $ONOS_ROOT
```

To build ONOS we need to run the command **bazel build onos** in the `$ONOS_ROOT` directory. This process could take an important amount of time, especially the first time since there are thousands of packages and libraries that need to be installed. There are times that the building could fail, throwing an error of any missing library or something similar. In that case, execute again the command quoted above to build ONOS. That said, if the problem persists, contact with the ONF organization via the discussion group [11].

Apart from this, it is important to mention that if we change some core files related to the ONOS API we will need to publish these files into the API before building, running the command **onos-publish -l** in the `$ONOS_ROOT` directory. With the use of this command, we will have the API updated and ready to use with the latest changes.

A.5. Installing Mininet (2.2.2)

```
$ cd ~  
$ sudo apt update  
$ sudo apt install mininet  
$ sudo mn -c  
$ git clone git://github.com/mininet/mininet  
$ cd mininet  
$ git tag
```



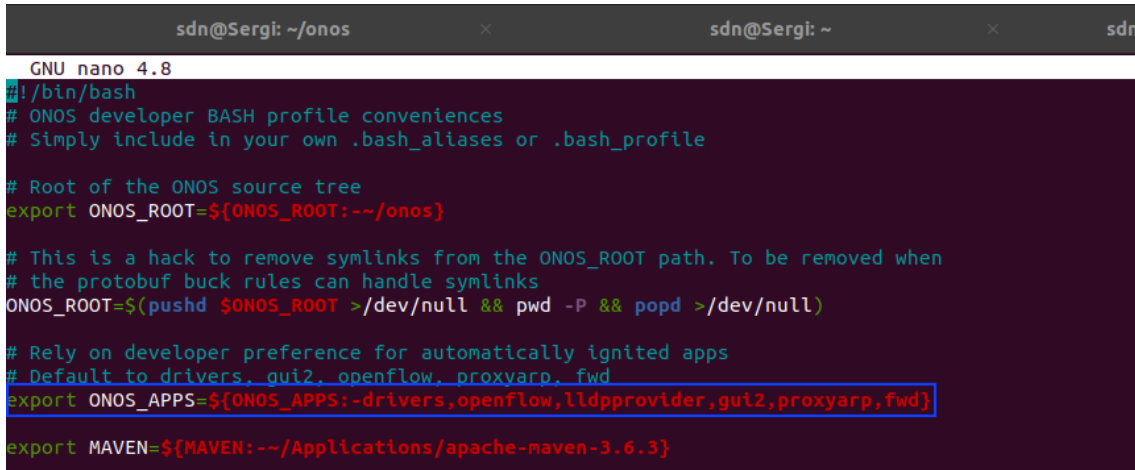
```
sdn@Sergi: ~/mininet  
sdn@Sergi:~/mininet$ git tag  
1.0.0  
2.0.0  
2.1.0  
2.1.0p1  
2.1.0p2  
2.2.0  
2.2.1  
2.2.2  
2.3.0d3  
2.3.0d4  
2.3.0d5  
2.3.0d6  
cs244-spring-2012-final  
sdn@Sergi:~/mininet$
```

Fig. A.7 List of Mininet versions

```
$ git checkout -b 2.2.2 2.2.2  
$ cd ..  
$ sudo mininet/util/install.sh -a
```

A.6. Running Mininet and ONOS

```
$ sudo nano $ONOS_ROOT/tools/dev/bash_profile
- export
  ONOS_APPS=${ONOS_APPS:-drivers,openflow,lldpprovider,gui2,proxy
  arp,fwd}
```



```
GNU nano 4.8
#!/bin/bash
# ONOS developer BASH profile conveniences
# Simply include in your own .bash_aliases or .bash_profile

# Root of the ONOS source tree
export ONOS_ROOT=${ONOS_ROOT:~/onos}

# This is a hack to remove symlinks from the ONOS_ROOT path. To be removed when
# the protobuf buck rules can handle symlinks
ONOS_ROOT=$(pushd $ONOS_ROOT >/dev/null && pwd -P && popd >/dev/null)

# Rely on developer preference for automatically ignited apps
# Default to drivers, gui2, openflow, proxyarp, fwd
export ONOS_APPS=${ONOS_APPS:-drivers,openflow,lldpprovider,gui2,proxyarp,fwd}

export MAVEN=${MAVEN:~/Applications/apache-maven-3.6.3}
```

Fig. A.8 ONOS_APPS variable in bash_profile file

```
$ cd ~
$ source .bashrc
```

Once we have set which applications we want to be activated by default on start, we need to run the controller in the \$ONOS_ROOT directory. To do so, we will run the command **bazel run onos-local** to run it without any extra configuration or **bazel run onos-local clean debug** if we want to do some kind of debugging using the IntelliJ IDEA (this option is strongly recommended so as not to stop and restart the controller to debug).

Eventually, when the controller is up and running, we will need to load our custom Mininet topology. Because we have specified all the attributes of the controller (see *custom-routing.py* on Appendix D), we will only need to run **sudo python custom-routing.py** under the \$ONOS_ROOT/utils/mininet/topologies directory.

A.7. Installing additional tools

As complementary tools, we have decided to install the widely known **Wireshark** software to analyze packets, the Integrated Development Environment (IDE) **IntelliJ IDEA** to code better and the **Jperf** application, which is the GUI of the iPerf command.

A.7.1. Installing Wireshark

```
$ cd ~
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install wireshark
$ sudo dpkg-reconfigure wireshark-common #Select <Yes> when asking
whether non-superusers should be able to capture packets
$ sudo chmod +x /usr/bin/dumpcap
```

A.7.2. Installing IntelliJ IDEA (2019.3.4) and Bazel project configuration

Firstly, it will be necessary to download the IDE to work more efficiently and comfortably. In my case, I have decided to use IntelliJ, which has a free edition called Community Edition. Since the onos project is built using Bazel, we will need to install the corresponding plugin. Even though there are IDE versions from 2020 onwards, the latest one handling with the use of Bazel is 2019.3.4. Next, there are the necessary commands to install IntelliJ. Once the script is executed, it will ask you whether to import settings or create a launcher script. It is strongly recommended to leave everything by default to avoid any future problem.

```
$ cd ~
$ cd Downloads
$ wget https://download.jetbrains.com/idea/ideaIU-2019.3.4.tar.gz
$ tar -zxvf ideaIU-2019.3.4.tar.gz -C ../
$ cd ..
$ sudo mv idea-IU-193.6911.18/ intelliJ
$ cd intelliJ/bin
$ sudo ./idea.sh
```

Secondly, we will install the Bazel plugin using the Marketplace, which can be found on settings under the plugins section. After the installation, in the “welcome” window we will see a new option called **Import Bazel Project**. The following numerical list explains the steps to follow with their corresponding images to understand the process better.

1. Select the Bazel workspace. This will depend on where the onos project was initially downloaded. In our case, the workspace is **/home/sdn/onos** (see Fig. A. 9).
2. Next, select the project view. In order not to import everything wasting time and memory, we will use the one provided by onos. The following commands will generate a temporary project file:
 - `$ cd $ONOS_ROOT/tools/dev/bin`
 - `$./onos-gen-bazel-project > /tmp/onos_bazelproject`
3. Back to the project view, select the **copy external** option, and type the path `/tmp/onos_bazelproject` (see Fig. A. 10).

4. Next, it will be shown a summary including all the information of the previous steps (see Fig. A. 11). Press **finish**.
5. Let the IDE sync all files of the project automatically. If this does not happen, just go to Bazel → Sync → Sync Project with BUILD Files (make sure Expand Sync to Working Set is selected). Syncing the files for the first time it could take up to 10 minutes (depending on the computer). If the synchronisation gives an error, repeat the aforementioned steps but this time deleting previously some cache folders and files, namely `~/.m2/`, `$ONOS_ROOT/.ijwb/`, and `$ONOS_ROOT/bazel-*` (where `*` stands for any file starting with `bazel-`).
6. Specify the bazel binary path, previously established in section A. 4. Go to File → Settings → Bazel Settings and select the configuration file (see Fig. A. 12).
7. In order to import the recommended IntelliJ settings, import the `.jar` file provided by onos. Go to File → Import Settings and select the file (see Fig. A. 13). To make the changes take effect, the IDE needs to be restarted.
8. Configure the Apache 2 license copyright header file to include at the beginning of every new file. Go to File → Settings → Editor → Copyright → Copyright Profiles and create a new one called ONOS (see Fig. A. 14). The copyright text can be found at `$ONOS_ROOT/tools/dev/header.txt`.
9. Mark `onos` directory as **Sources Root** thus all packages are interpreted correctly by the IDE (see Fig. A. 15). Once done, a white dot will appear in the `apps` folder.
10. Once it has finished, the IDE will not detect the files inside the `routing-app` folder as `.java` since it was built with maven (see Fig. A. 16). To solve this problem, it will be necessary to configure the application directory as a Maven project. To do so, right click on the `pom.xml` file and select **add as Maven project** (see Fig. A. 17). Now, IntelliJ will detect the files as `.java` and will find declarations to go (see Fig. A. 18).
11. To avoid maven dependencies in the `pom.xml` file from being detected as *not* OSGi ready, go to File → Project Structure → Modules → `routing-app` and delete the already-existing OSGi framework (see Fig. A. 19).
12. Since the application is built with Maven, in order to include external dependencies in the `routing-app-1.0-SNAPSHOT.oar` file under `$ONOS_ROOT/apps/routing-app/target`, including `commons-net`, `jgrapht-core`, `jheaps`, and `json`, it will be required to create a custom **app.xml** and **features.xml** file under `$ONOS_ROOT/apps/routing-app`. The first one is in charge of converting `.jar` dependencies into `.oar`, whereas the latter loads the bundles at run-time so can be used by the application.
13. Finally -this step is not always necessary as it depends on whether the IDE was previously used for other projects- it will be required to define a project Software Development Kit (SDK). For this purpose, go to File → Project Structure → Project and add a new Java Development Kit (JDK), which, by default, is located at `/usr/lib/jvm/` (see Fig. A. 20 and Fig. A. 21).

Executing all the steps aforesaid will make the IDE ready to work with the *routing-app* application inside the ONOS project. Bear in mind that if all files are synchronized again with Bazel, IntelliJ will not be able to find declarations and imports. Notwithstanding, recall that the custom application is built and installed using maven so doing the synchronization only once at the beginning will be enough.

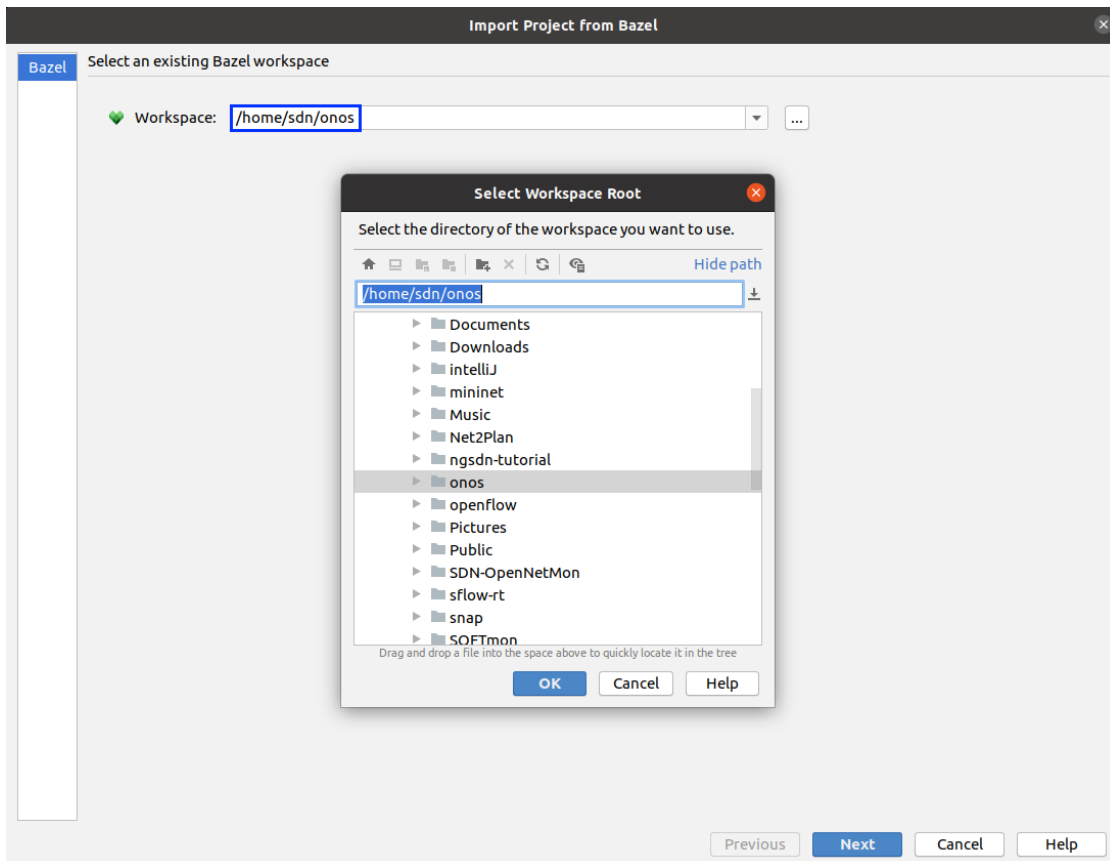


Fig. A.9 Bazel workspace

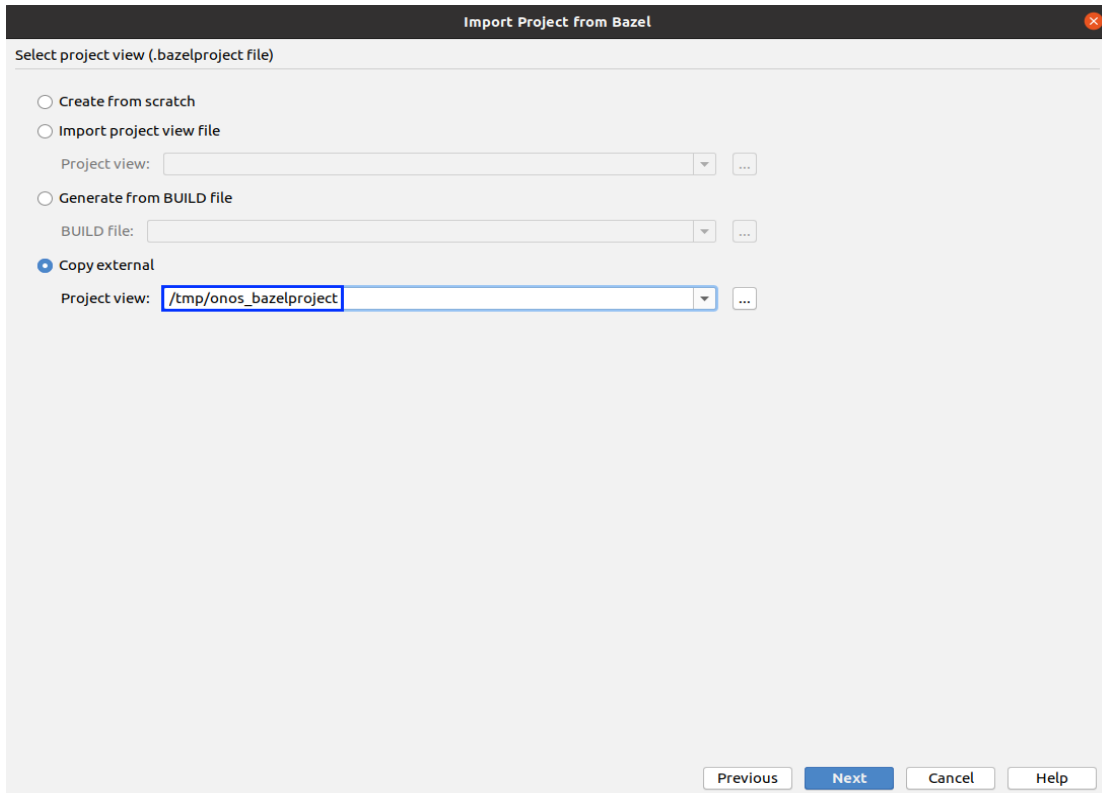


Fig. A.10 Copy external option to select the project view

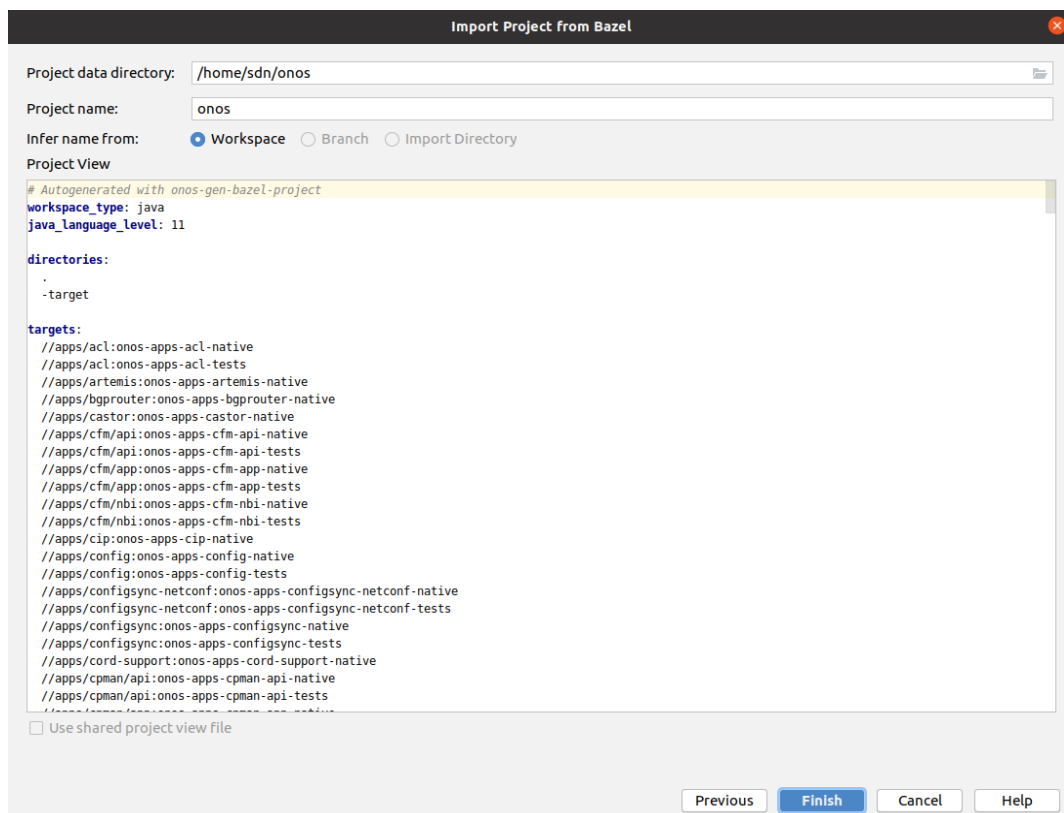


Fig. A.11 Bazel project view summary

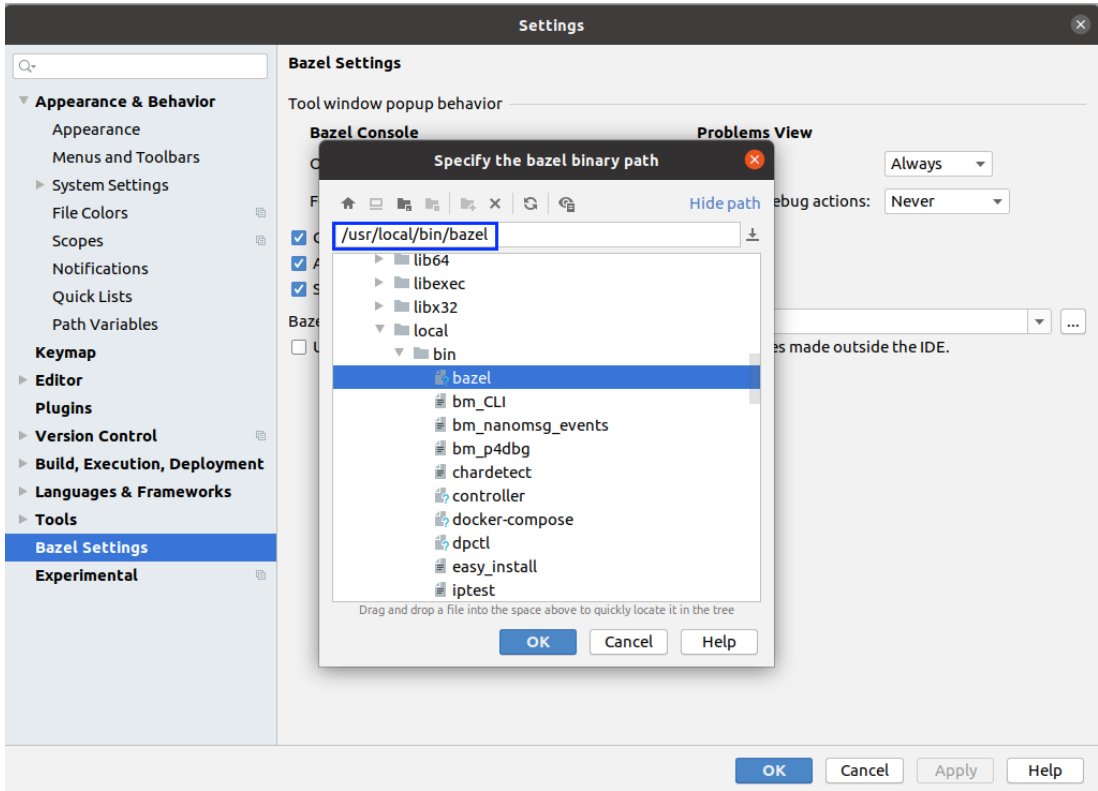


Fig. A.12 Bazel binary path

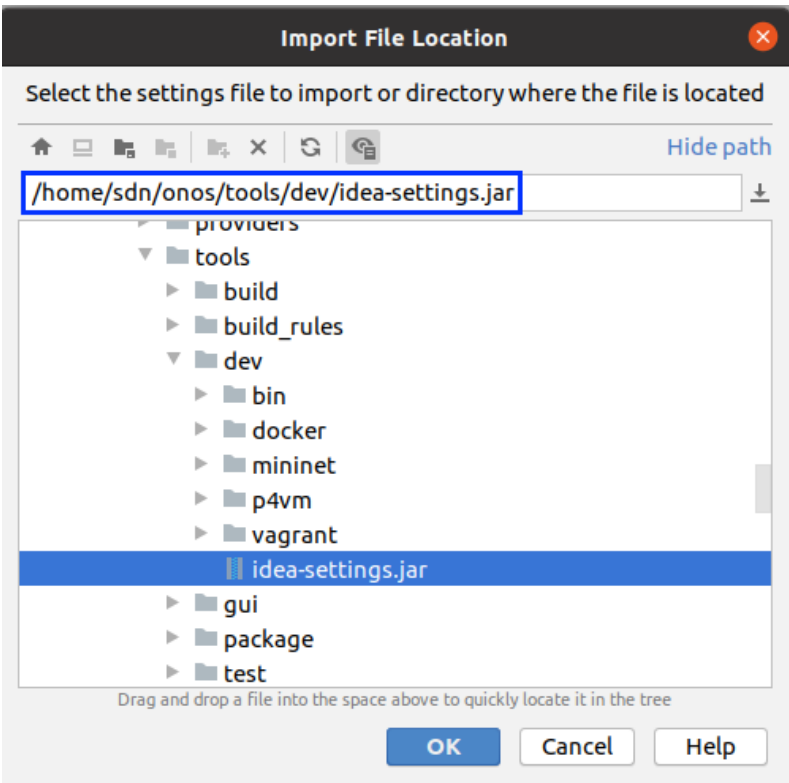


Fig. A.13 IntelliJ settings for ONOS project

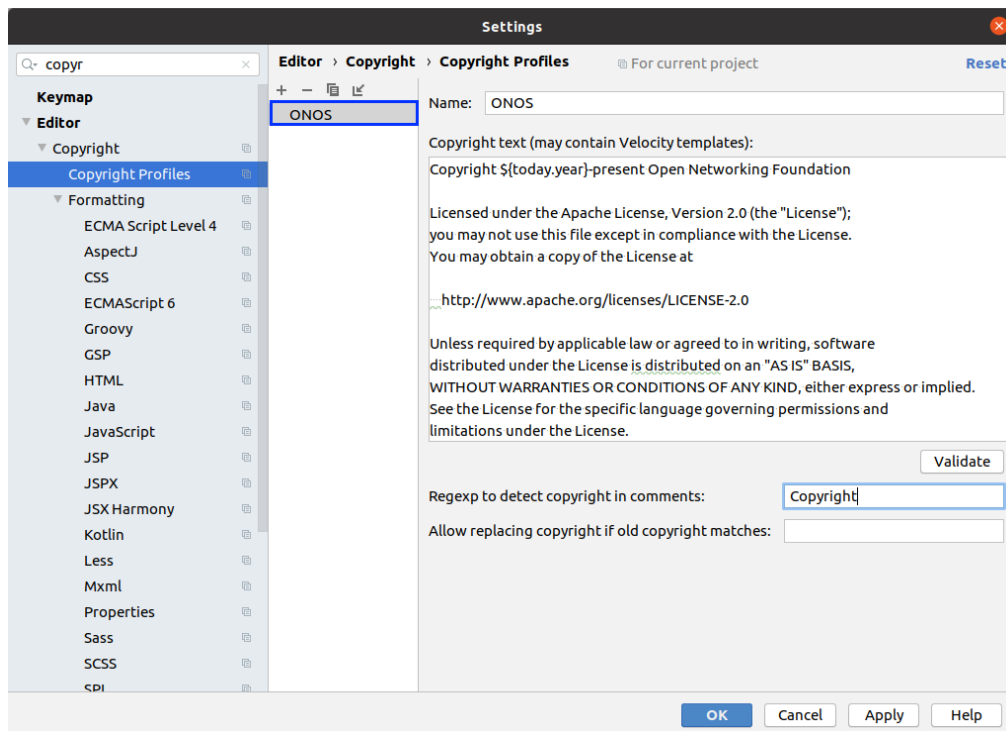


Fig. A.14 ONOS copyright profile for Apache2 license

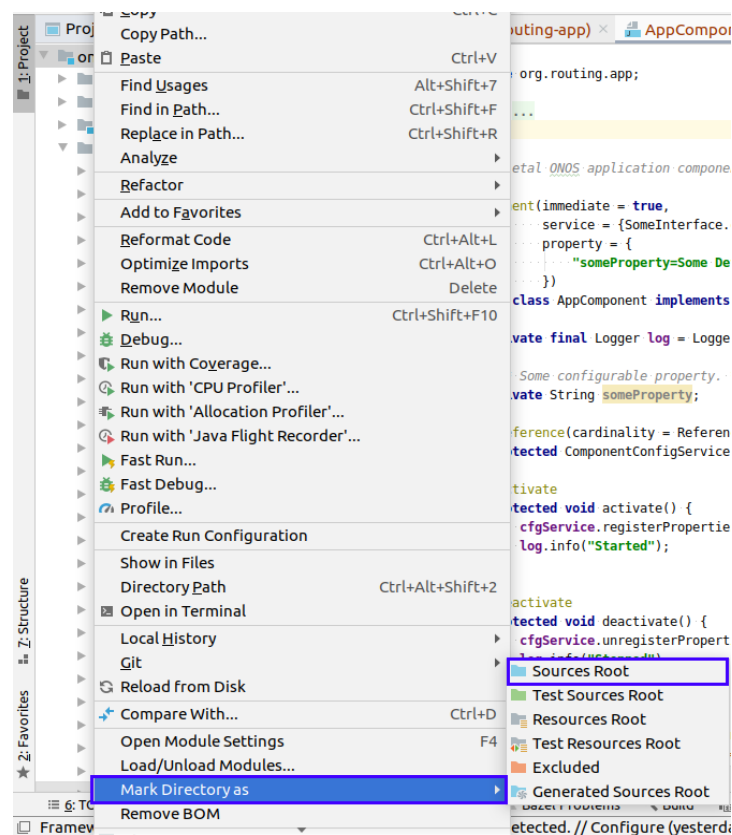


Fig. A.15 Mark onos directory as Sources Root

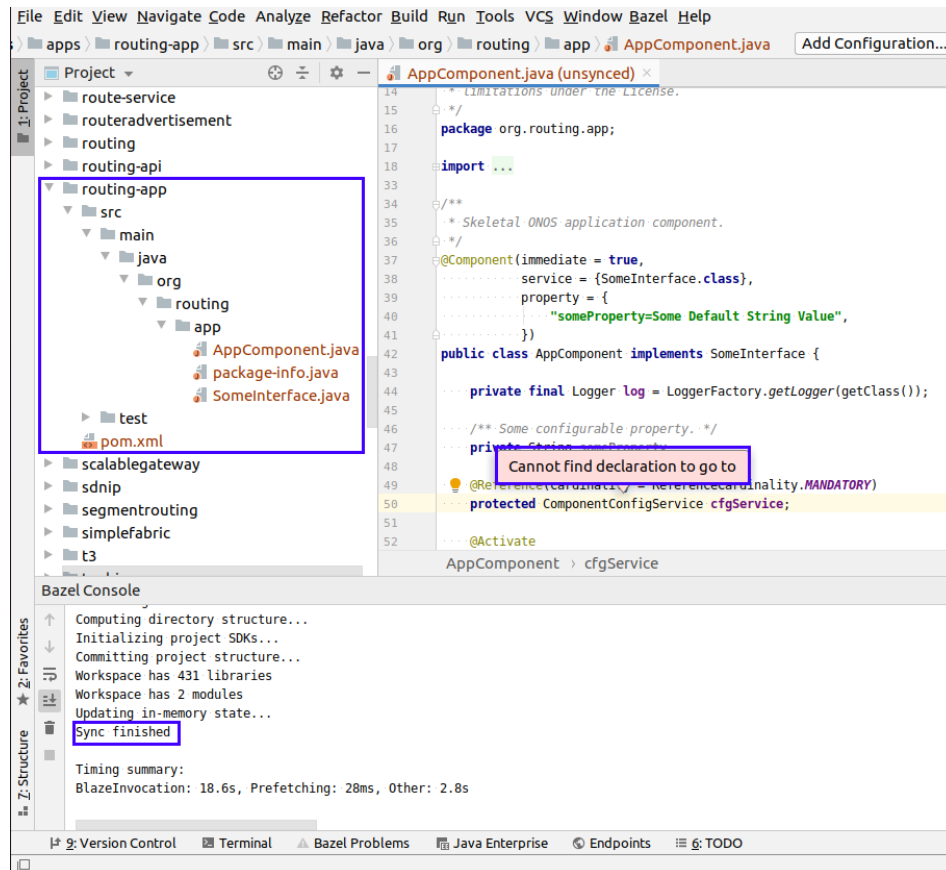
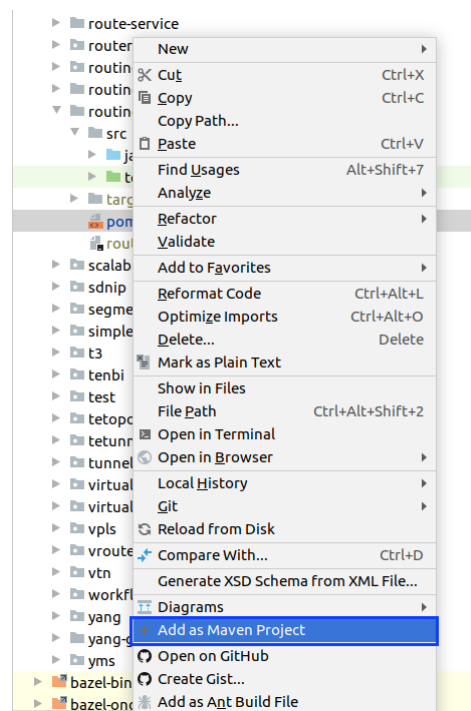


Fig. A.16 Project view errors in IntelliJ

Fig. A.17 Add *routing-app* directory as a Maven project

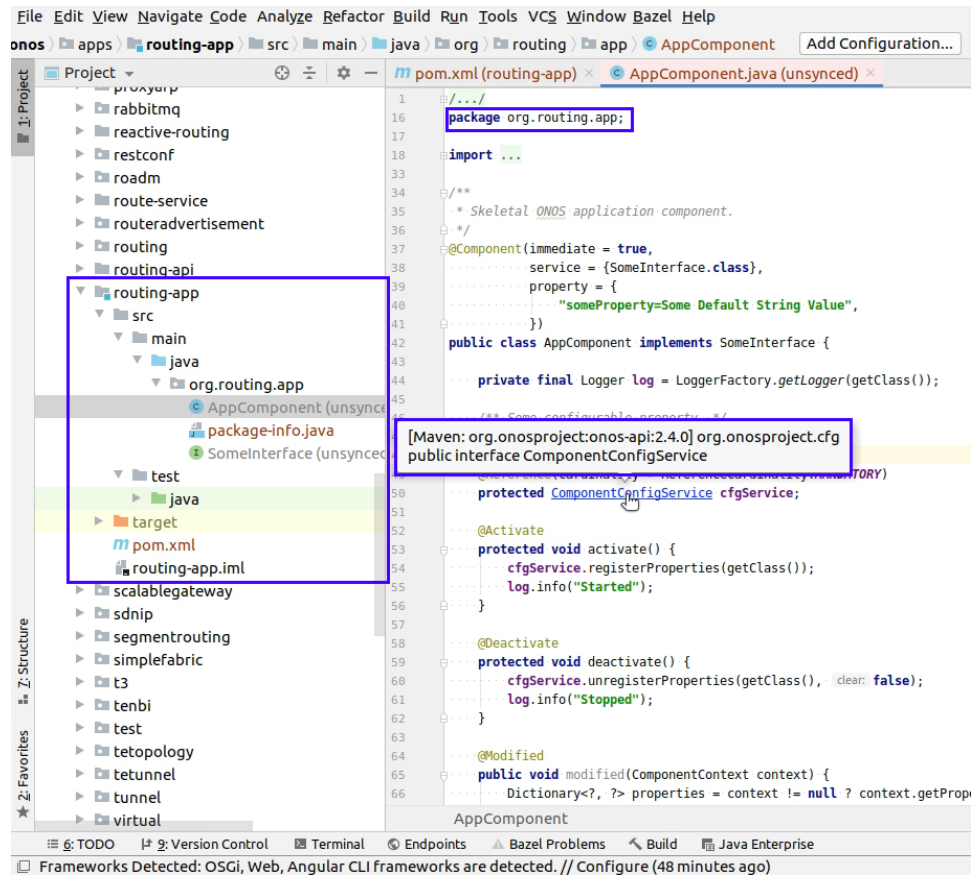
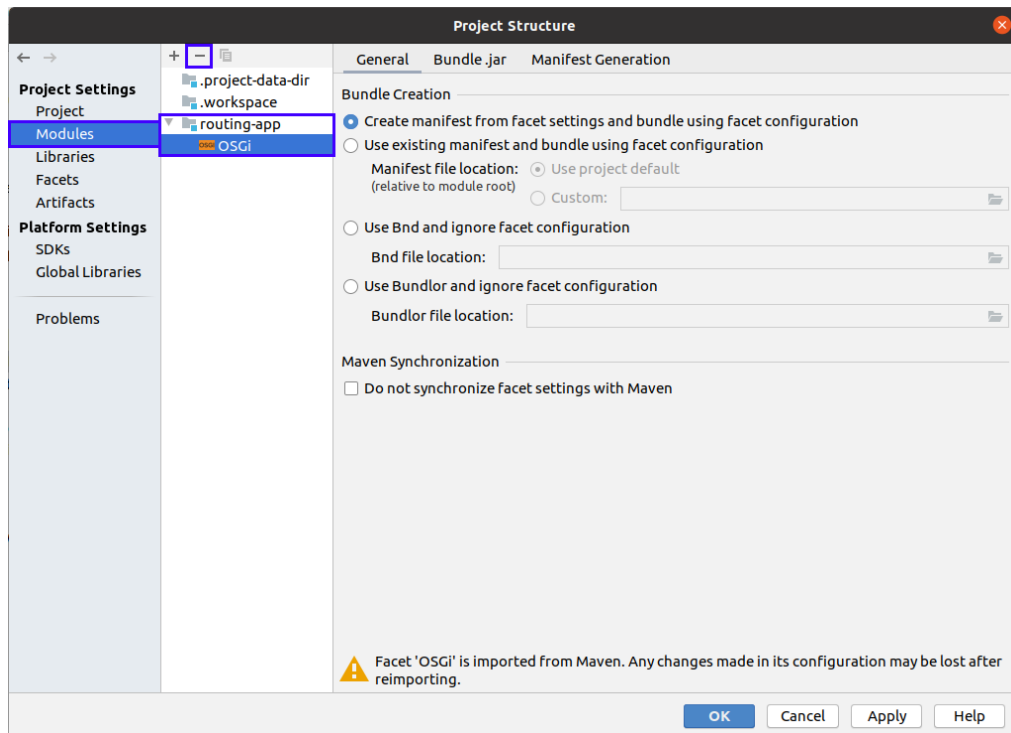


Fig. A.18 Proper project view in IntelliJ

Fig. A.19 Delete the OSGi framework under *routing-app* module

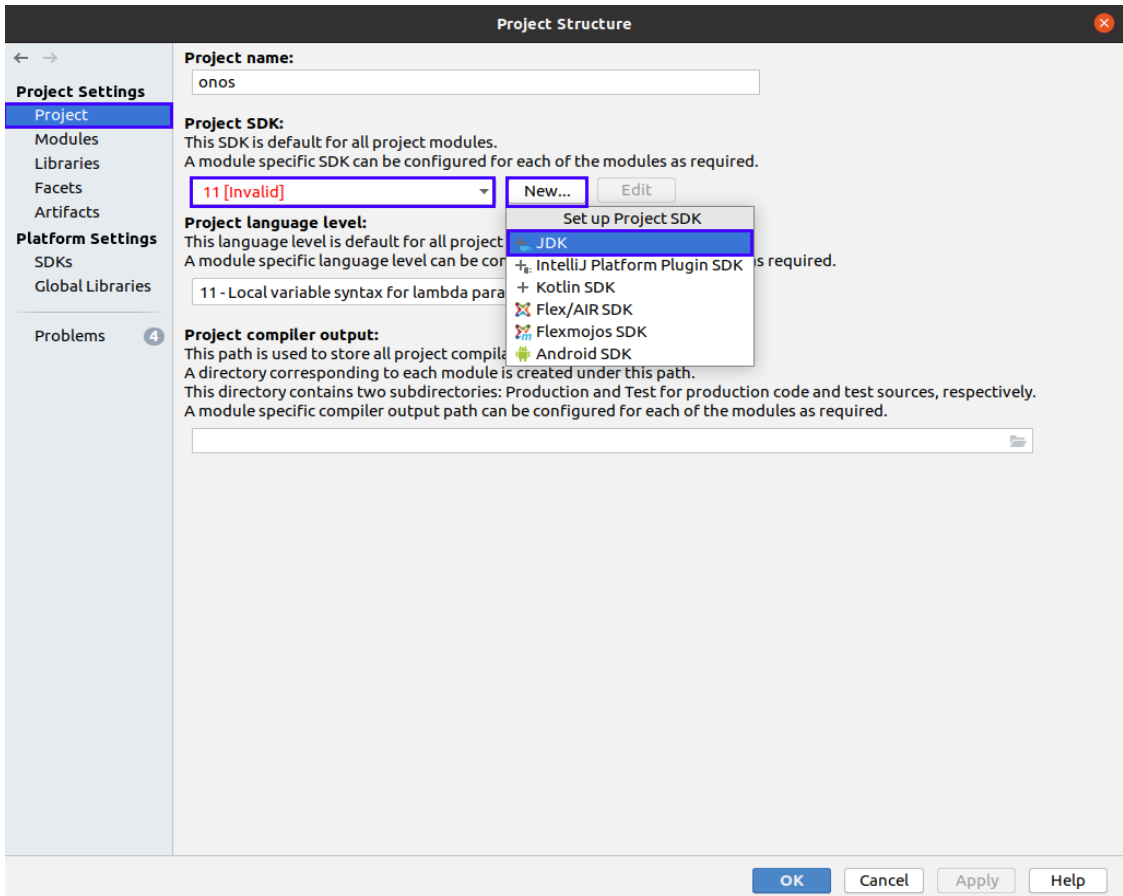


Fig. A.20 Set up a new project SDK

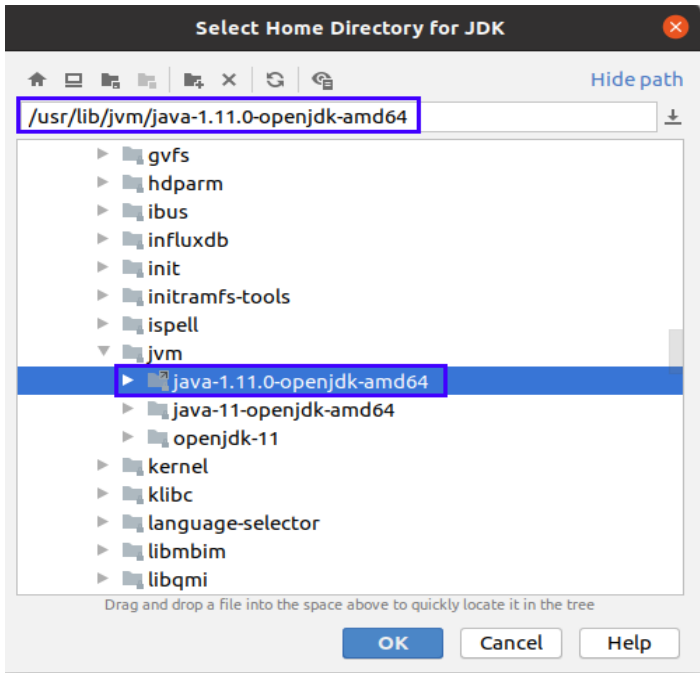


Fig. A.21 Select the Home Directory for JDK

Finally, in order to debug ONOS it will be necessary to create a new debug configuration. To do so, click on **Add Configuration...** in the top right-hand corner, press the plus sign, select *Remote* and leave the host and the port as default, since we will debug ONOS on our machine (see Fig. A. 22). Take into account that in order to debug ONOS, the controller has to be launched with the debug option (refer to section A. 6).

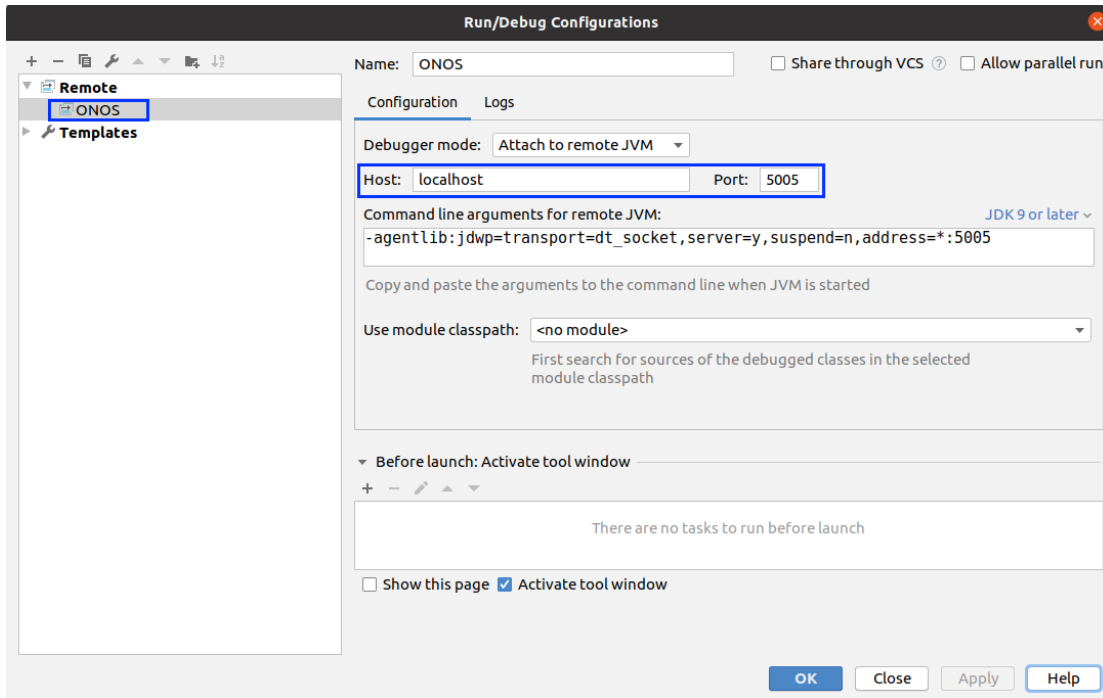


Fig. A.22 IntelliJ debug configuration

A.7.3. Installing Jperf

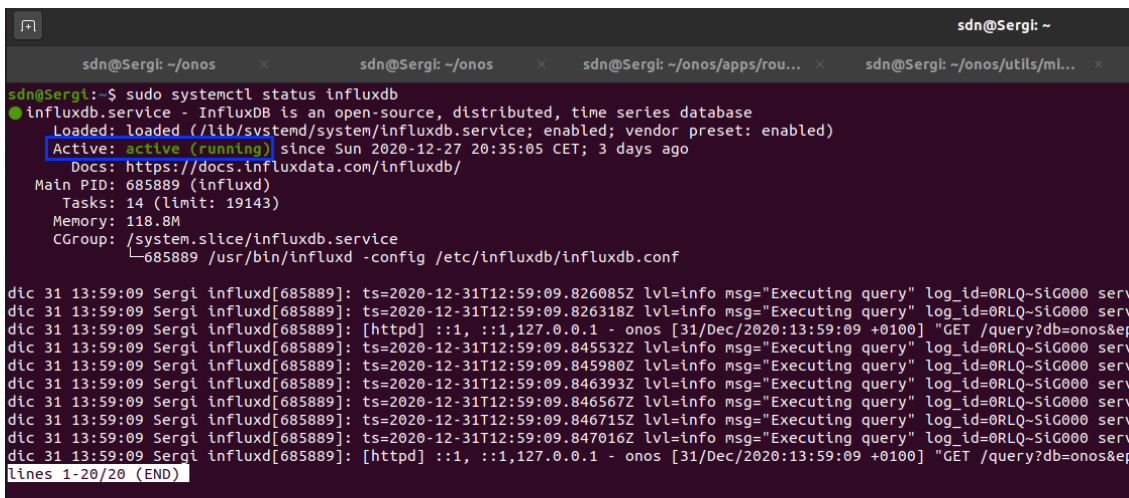
```
$ cd ~
$ cd Downloads
$ wget
https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.
com/xjperf/jperf-2.0.2.zip
$ sudo unzip jperf-2.0.2.zip
$ mv jperf-2.0.2 /home/sdn/onos/utils/mininet/topologies/jperf
$ cd /home/sdn/onos/utils/mininet/topologies/jperf
$ sudo chmod +x jperf.sh
```

APPENDIX B. INSTALLATION AND CONFIGURATION OF MONITORING TOOLS

In this appendix, it is explained thoroughly how to install each and every tool necessary to monitor the desired parameters. **InfluxDB** will be used to store all the measurements and its values whilst **Grafana** is used for monitoring and observability.

B.1. Installing InfluxDB

```
$ cd ~
$ echo "deb https://repos.influxdata.com/ubuntu focal stable" | sudo tee
/etc/apt/sources.list.d/influxdb.list
$ sudo curl -sL https://repos.influxdata.com/influxdb.key | sudo apt-key add -
$ sudo apt update
$ sudo apt install influxdb
$ sudo systemctl enable --now influxdb
$ sudo systemctl status influxdb
```



```
sdn@Sergi: ~
sdn@Sergi: ~/onos
sdn@Sergi: ~/onos/apps/rou...
sdn@Sergi: ~/onos/utis/ml...

sdn@Sergi:~$ sudo systemctl status influxdb
● influxdb.service - InfluxDB is an open-source, distributed, time series database
   Loaded: loaded (/lib/systemd/system/influxdb.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2020-12-27 20:35:05 CET; 3 days ago
     Docs: https://docs.influxdata.com/influxdb/
    Main PID: 685889 (influxd)
      Tasks: 14 (limit: 19143)
     Memory: 118.8M
      CGroup: /system.slice/influxdb.service
             └─685889 /usr/bin/influxd -config /etc/influxdb/influxdb.conf

dic 31 13:59:09 Sergi influxd[685889]: ts=2020-12-31T12:59:09.826085Z lvl=info msg="Executing query" log_id=0RLQ-SiG000 serv
dic 31 13:59:09 Sergi influxd[685889]: ts=2020-12-31T12:59:09.826318Z lvl=info msg="Executing query" log_id=0RLQ-SiG000 serv
dic 31 13:59:09 Sergi influxd[685889]: [httpd] ::1, ::1,127.0.0.1 - onos [31/Dec/2020:13:59:09 +0100] "GET /query?db=onos&eq
dic 31 13:59:09 Sergi influxd[685889]: ts=2020-12-31T12:59:09.845532Z lvl=info msg="Executing query" log_id=0RLQ-SiG000 serv
dic 31 13:59:09 Sergi influxd[685889]: ts=2020-12-31T12:59:09.845980Z lvl=info msg="Executing query" log_id=0RLQ-SiG000 serv
dic 31 13:59:09 Sergi influxd[685889]: ts=2020-12-31T12:59:09.846393Z lvl=info msg="Executing query" log_id=0RLQ-SiG000 serv
dic 31 13:59:09 Sergi influxd[685889]: ts=2020-12-31T12:59:09.846567Z lvl=info msg="Executing query" log_id=0RLQ-SiG000 serv
dic 31 13:59:09 Sergi influxd[685889]: ts=2020-12-31T12:59:09.846715Z lvl=info msg="Executing query" log_id=0RLQ-SiG000 serv
dic 31 13:59:09 Sergi influxd[685889]: ts=2020-12-31T12:59:09.847016Z lvl=info msg="Executing query" log_id=0RLQ-SiG000 serv
dic 31 13:59:09 Sergi influxd[685889]: [httpd] ::1, ::1,127.0.0.1 - onos [31/Dec/2020:13:59:09 +0100] "GET /query?db=onos&eq
lines 1-20/20 (END)
```

Fig. B.1 InfluxDB service up and running

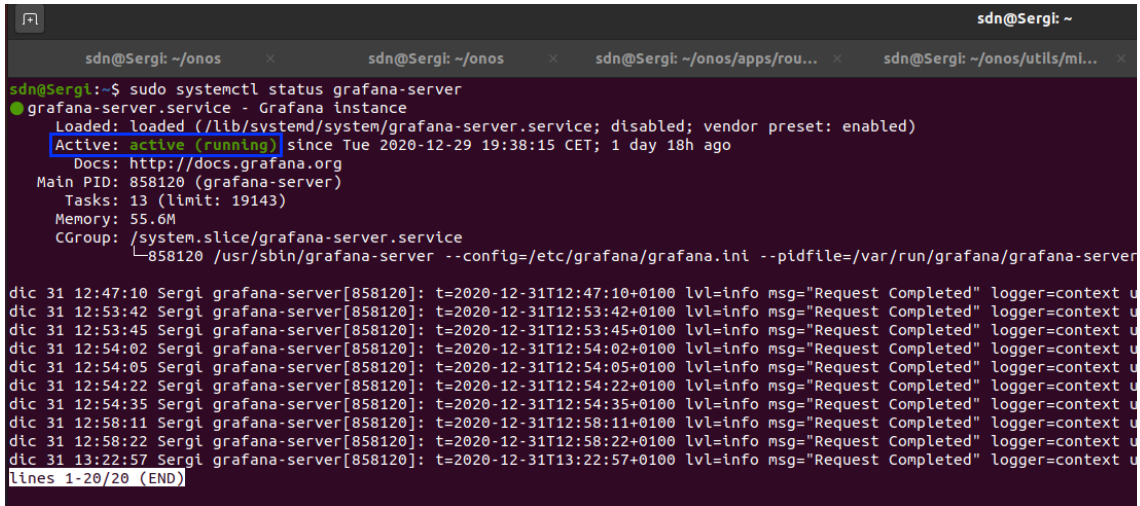
B.1.1. Configuration process

Commands to create the database in InfluxDB to store the data:

```
$ influx
> CREATE DATABASE onos
> USE onos
```

B.2. Installing Grafana

```
$ cd ~
$ sudo apt-get install -y adduser libfontconfig1
$ wget https://dl.grafana.com/oss/release/grafana_7.3.6_amd64.deb
$ sudo dpkg -i grafana_7.3.6_amd64.deb
$ sudo systemctl enable --now grafana-server
$ sudo systemctl status grafana-server
```



```
sdn@Sergi: ~
sdn@Sergi: ~/onos
sdn@Sergi: ~/onos/apps/rou...
sdn@Sergi: ~/onos/utlis/ml...

sdn@Sergi:~$ sudo systemctl status grafana-server
● grafana-server.service - Grafana instance
   Loaded: loaded (/lib/systemd/system/grafana-server.service; disabled; vendor preset: enabled)
   Active: active (running) since Tue 2020-12-29 19:38:15 CET; 1 day 18h ago
     Docs: http://docs.grafana.org
    Main PID: 858120 (grafana-server)
      Tasks: 13 (limit: 19143)
     Memory: 55.6M
    CGroup: /system.slice/grafana-server.service
            └─858120 /usr/sbin/grafana-server --config=/etc/grafana/grafana.ini --pidfile=/var/run/grafana/grafana-server

dic 31 12:47:10 Sergi grafana-server[858120]: t=2020-12-31T12:47:10+0100 lvl=info msg="Request Completed" logger=context u
dic 31 12:53:42 Sergi grafana-server[858120]: t=2020-12-31T12:53:42+0100 lvl=info msg="Request Completed" logger=context u
dic 31 12:53:45 Sergi grafana-server[858120]: t=2020-12-31T12:53:45+0100 lvl=info msg="Request Completed" logger=context u
dic 31 12:54:02 Sergi grafana-server[858120]: t=2020-12-31T12:54:02+0100 lvl=info msg="Request Completed" logger=context u
dic 31 12:54:05 Sergi grafana-server[858120]: t=2020-12-31T12:54:05+0100 lvl=info msg="Request Completed" logger=context u
dic 31 12:54:22 Sergi grafana-server[858120]: t=2020-12-31T12:54:22+0100 lvl=info msg="Request Completed" logger=context u
dic 31 12:54:35 Sergi grafana-server[858120]: t=2020-12-31T12:54:35+0100 lvl=info msg="Request Completed" logger=context u
dic 31 12:58:11 Sergi grafana-server[858120]: t=2020-12-31T12:58:11+0100 lvl=info msg="Request Completed" logger=context u
dic 31 12:58:22 Sergi grafana-server[858120]: t=2020-12-31T12:58:22+0100 lvl=info msg="Request Completed" logger=context u
dic 31 13:22:57 Sergi grafana-server[858120]: t=2020-12-31T13:22:57+0100 lvl=info msg="Request Completed" logger=context u
lines 1-20/20 (END)
```

Fig. B.2 Grafana service up and running

B.2.1. Configuration process

Having installed InfluxDB and Grafana, now we are ready to configure the communication between both.

First of all, we will proceed to go to the Grafana URL, which is **localhost:3000/login**. By default, Grafana credentials are username: **admin** and password: **admin** (see Fig. B. 3) and when logging in we will have the possibility to introduce a new password or just keep the default (it is recommended to use a different password).

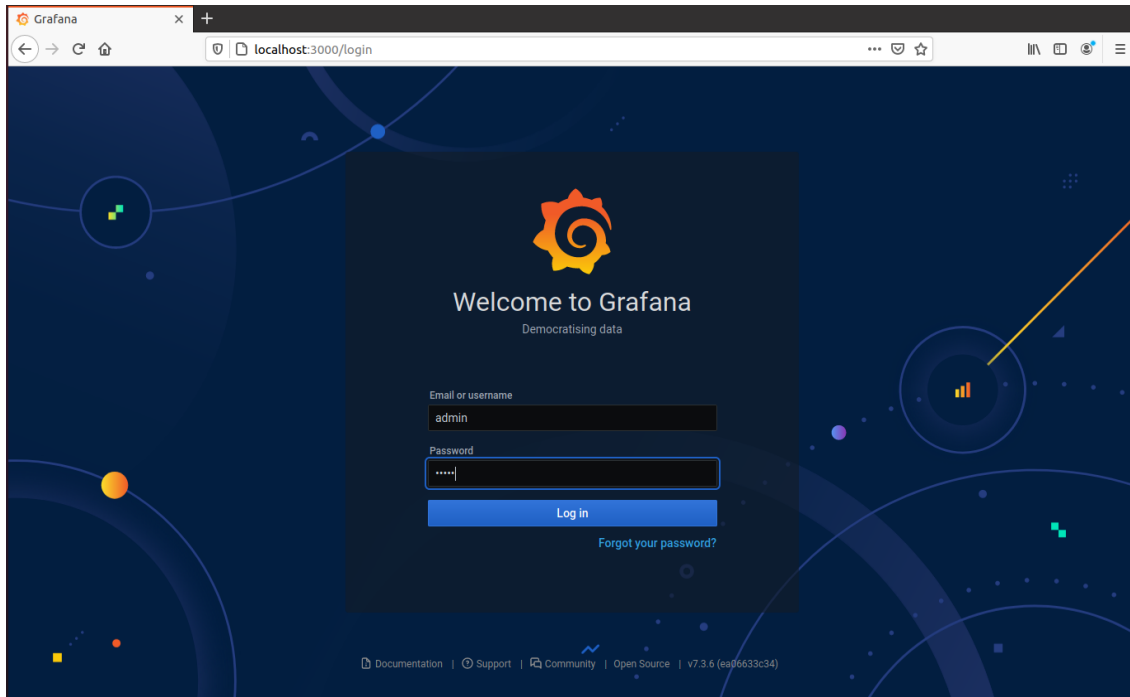


Fig. B.3 Default login screen in Grafana

Secondly, we will need to use InfluxDB as the data source. To do so, we will go to Configuration → Data Sources → Add data source (see Fig. B. 4). We will have to select InfluxDB under the Time series databases section (see Fig. B. 5).

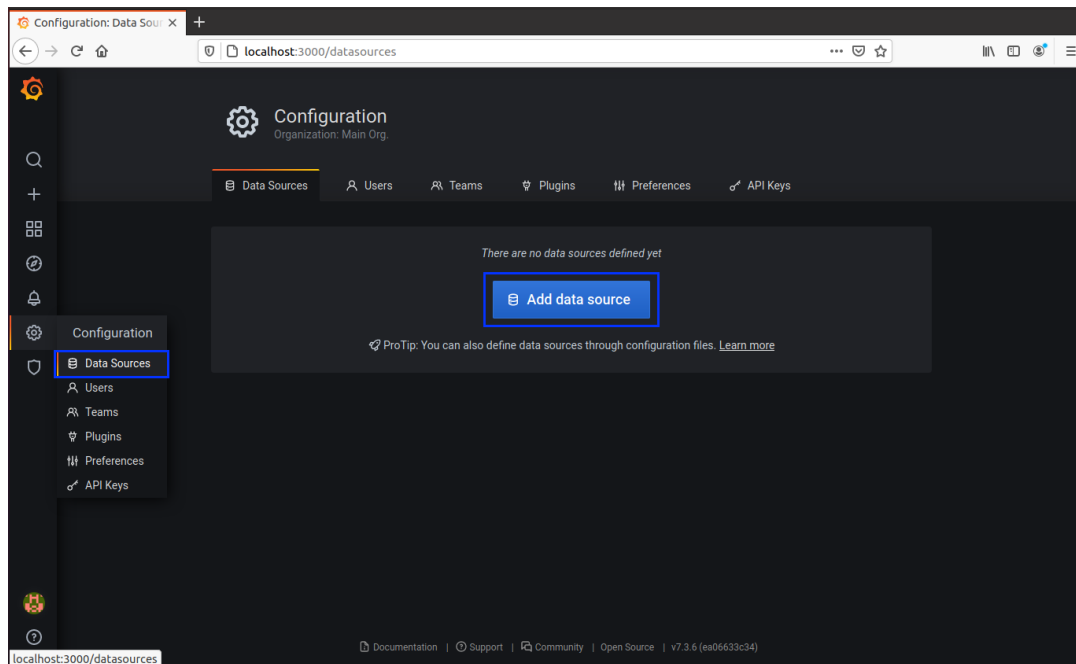


Fig. B.4 Add data source in Grafana

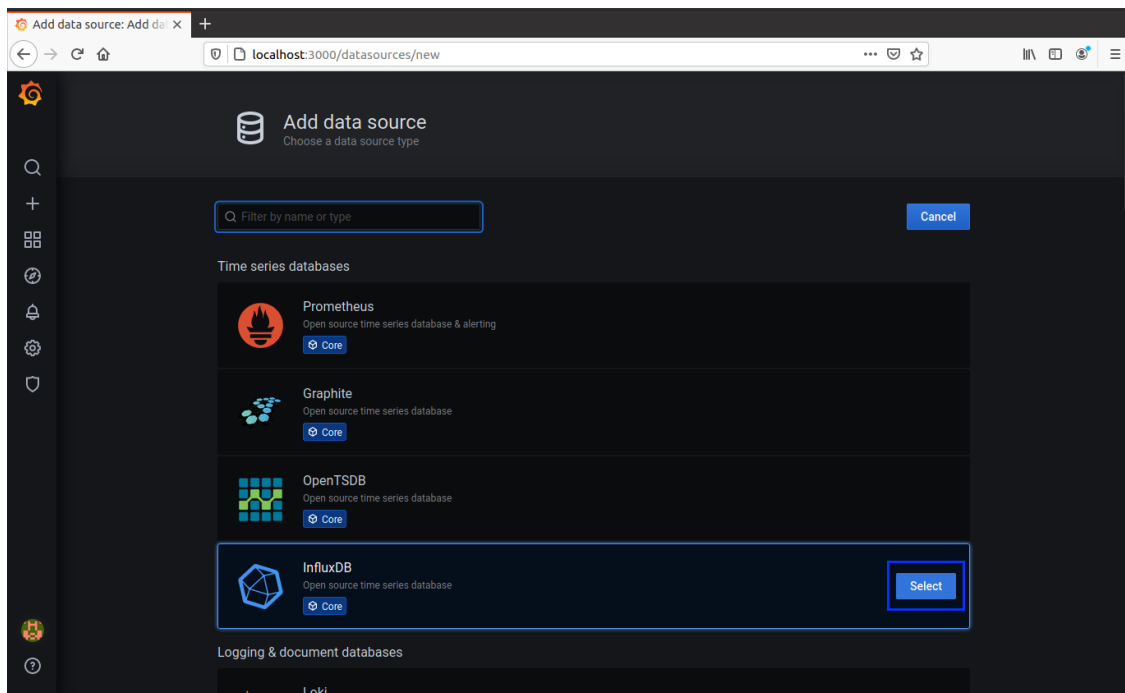


Fig. B.5 Add InfluxDB as data source in Grafana

Finally, we will have to set the URL that points to InfluxDB service, which is **http://localhost:8086** (see Fig. B. 6), and specify which is the database where the measurements will be stored, previously created in subsection B.1.1 as **onos** (see Fig. B. 7).

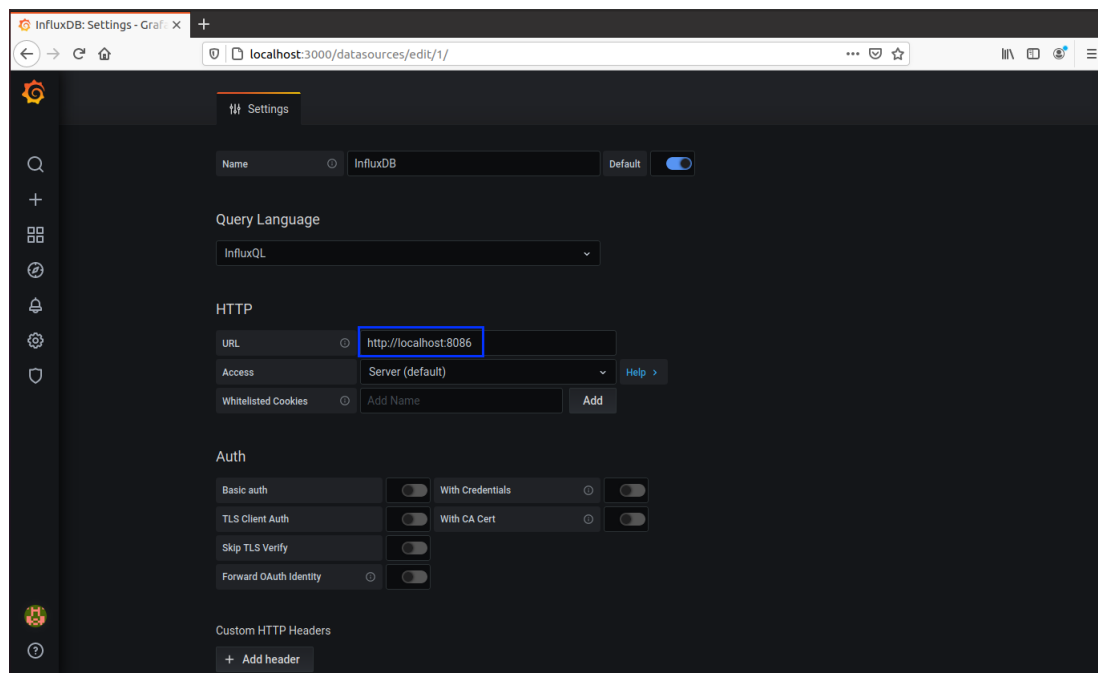


Fig. B.6 InfluxDB configuration in Grafana (I)

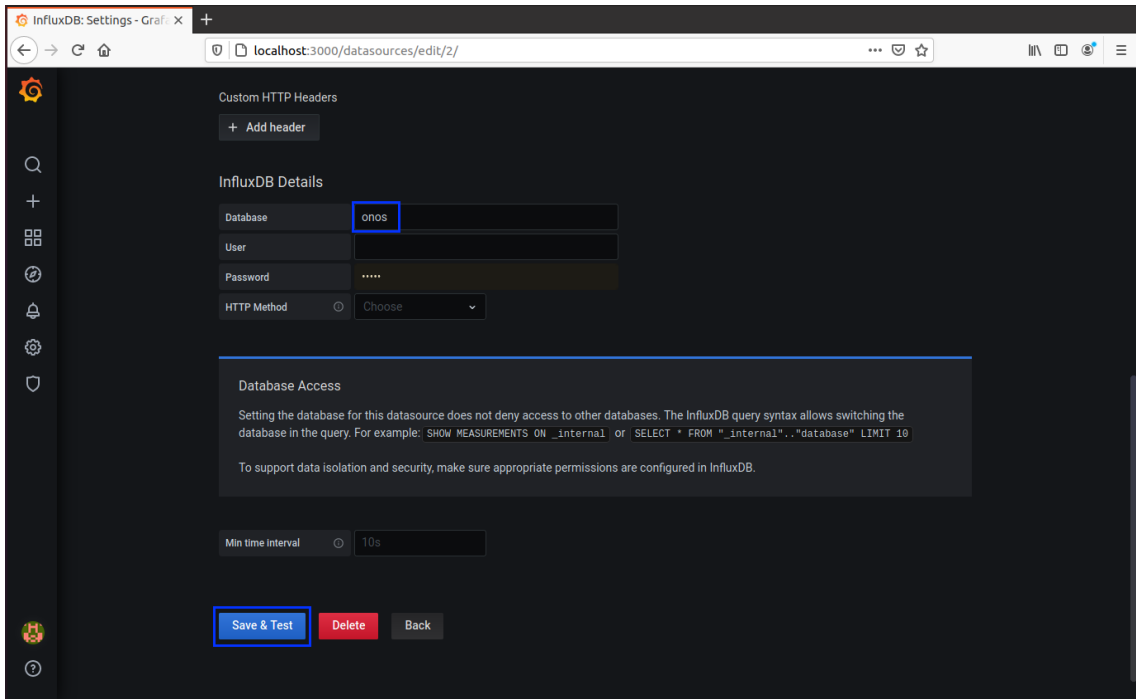


Fig. B.7 InfluxDB configuration in Grafana (II)

APPENDIX C. NETWORK TOPOLOGIES, MONITORING SCRIPTS, AND OTHER FILES OF INTEREST

Table. C.1 pom.xml file of our custom routing-app application

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
 ~ Copyright 2020 Open Networking Foundation
 ~
 ~ Licensed under the Apache License, Version 2.0 (the "License");
 ~ you may not use this file except in compliance with the License.
 ~ You may obtain a copy of the License at
 ~
 ~     http://www.apache.org/licenses/LICENSE-2.0
 ~
 ~ Unless required by applicable law or agreed to in writing, software
 ~ distributed under the License is distributed on an "AS IS" BASIS,
 ~ WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 ~ See the License for the specific language governing permissions and
 ~ limitations under the License.
-->
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.onosproject</groupId>
    <artifactId>onos-dependencies</artifactId>
    <version>2.4.0</version>
  </parent>

  <groupId>org.routing.app</groupId>
  <artifactId>routing-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>bundle</packaging>

  <description>ONOS OSGi bundle archetype</description>
  <url>http://onosproject.org</url>

  <properties>
    <onos.app.name>org.routing.app</onos.app.name>
    <onos.app.title>Custom Routing Application</onos.app.title>
    <onos.app.origin>Sergio Vera-UPC</onos.app.origin>
    <onos.app.category>default</onos.app.category>
    <onos.app.url>http://onosproject.org</onos.app.url>
    <onos.app.readme>ONOS OSGi bundle archetype.</onos.app.readme>
    <maven.test.skip>true</maven.test.skip>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.onosproject</groupId>
      <artifactId>onos-api</artifactId>
      <version>${onos.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.onosproject</groupId>
      <artifactId>onlab-osgi</artifactId>
      <version>${onos.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.onosproject</groupId>
      <artifactId>onlab-misc</artifactId>
      <version>${onos.version}</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
```

```

    <groupId>org.onosproject</groupId>
    <artifactId>onos-api</artifactId>
    <version>${onos.version}</version>
    <scope>test</scope>
    <classifier>tests</classifier>
  </dependency>
  <dependency>
    <groupId>org.onosproject</groupId>
    <artifactId>onos-cli</artifactId>
    <version>${onos.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.onosproject</groupId>
    <artifactId>onos-core-serializers</artifactId>
    <version>${onos.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.osgi</groupId>
    <artifactId>org.osgi.service.component</artifactId>
    <version>1.4.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.karaf.shell</groupId>
    <artifactId>org.apache.karaf.shell.console</artifactId>
    <version>4.2.9</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.onosproject</groupId>
    <artifactId>onos-app-iptology-api</artifactId>
    <version>1.6.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.12.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.jgrapht</groupId>
    <artifactId>jgrapht-core</artifactId>
    <version>1.5.1</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-math3</artifactId>
    <version>3.6.1</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>commons-net</groupId>
    <artifactId>commons-net</artifactId>
    <version>3.6</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.jheaps</groupId>
    <artifactId>jheaps</artifactId>
    <version>0.14</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20201115</version>
    <scope>provided</scope>
  </dependency>
</dependency>

```

```

    <groupId>com.googlecode.json-simple</groupId>
    <artifactId>json-simple</artifactId>
    <version>1.1.1</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.6</version>
    <scope>provided</scope>
  </dependency>
</dependencies>

<repositories>
  <repository>
    <id>maven-snapshots</id>
    <url>http://oss.sonatype.org/content/repositories/snapshots</url>
    <layout>default</layout>
    <releases>
      <enabled>false</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>

<build>
  <plugins>
    <plugin>
      <groupId>org.onosproject</groupId>
      <artifactId>onos-maven-plugin</artifactId>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <configuration>
        <skip>true</skip>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

C.1. Network topologies

The switches and hosts of each topology used for this project have been implemented by Mininet. If it is true that Mininet can be configured via CLI, it has a powerful API that can be employed to pre-configure the network. The command used to run the scripts in is ***sudo python file_name***, and each script has three big blocks.

The first of them, shown in table D. 6, is the method in charge of uploading the network configuration to ONOS, so then it can be downloaded and read by the application. To do so, it uses the API with the base URL -in our case- ***http://192.168.99.106:8181/onos/v1***, and having an endpoint ***network/configuration***. Taking a look at the swagger, which is already implemented in the onos project, the JSON format needed to be sent in the body when doing a POST request contains the following fields (see also Fig. D. 3):

Table. C.2 JSON structure for the network configuration

```

1.  {
2.    "devices": {
3.      "of:0000000000000004": {
4.        "basic": {
5.          "latitude": "42.38",
6.          "name": "S4",
7.          "longitude": "-5.82"
8.        },
9.        "classifiers": [
10.         {
11.           "ethernet-type": "LLDP",
12.           "target-queue": 0
13.         },
14.         {
15.           "ethernet-type": "BDDP",
16.           "target-queue": 0
17.         }
18.       ]
19.     },
20.     "ports": {},
21.     "apps": {
22.       "org.onosproject.provider.lldp": {
23.         "suppression": {
24.           "deviceTypes": [
25.             "ROADM",
26.             "OTN",
27.             "FIBER_SWITCH",
28.             "OPTICAL_AMPLIFIER",
29.             "OLS",
30.             "TERMINAL_DEVICE"
31.           ],
32.           "annotation": "{\\"no-lldp\\":null}"
33.         }
34.       }
35.     },
36.     "regions": {},
37.     "hosts": {
38.       "00:00:00:00:00:0A/None": {
39.         "basic": {
40.           "latitude": "41.63",
41.           "longitude": "-8"
42.         }
43.       }
44.     },
45.     "links": {
46.       "of:0000000000000001/5-of:0000000000000004/3": {
47.         "basic": {
48.           "bidirectional": true,
49.           "bandwidth": 1,
50.           "durable": true
51.         }
52.       }
53.     },
54.     "layouts": {}
55.   }
56. }

```

As can be seen, many parameters can be set when initializing ONOS, which opens a great window to exchange information between the program in charge of setting the topologies and the controller. In our case, these are the ones that have been specified:

- **Devices** → Set the latitude and longitude along with the name of the switch. The value of these parameters will be reflected in the GUI when running ONOS. The coordinates become really powerful when plotting real networks, as is the case of GÉANT.
- **Hosts** → As for devices, the hosts generated in Mininet will also be placed in the specified position.

- **Links** → Because the bandwidth of the links in ONOS cannot be set, the only way of telling it using Mininet is utilizing this POST service. The keys inside the links are the string representation in the only format supported by the controller. The method *addLink* (line 103 from table C.3) receives the *source*, the *source URI*, the *destination*, the *destination URI*, and the *bandwidth* (in Mbps). Based on these parameters, the information from the Mininet API, and knowing that the ID of the device always follows the same format, all the information for each link -in both directions- can be configured.

POST

/network/configuration

Uploads bulk network configuration

Parameters

Parameter	Value	Description	Parameter Type	Data Type
request	(required)	network configuration JSON rooted at the top node	body	undefined

Parameter content type: application/json

Response Messages

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

Try it out!

Fig. C.1 Request to upload the full network configuration

Table. C.3 Python method to upload the network configuration to ONOS

```

1.  #!/usr/bin/python
2.
3.  from mininet.topo import Topo
4.  from mininet.cli import CLI
5.  from mininet.link import TCLink
6.  from mininet.log import setLogLevel
7.  from mininet.net import Mininet
8.  from mininet.node import RemoteController, Host
9.  from mininet.util import quietRun
10. from os import listdir, environ
11. from requests import put
12. import re, socket, fcntl, array, struct, sys, requests, json, os
13.
14. URL = "http://192.168.99.106:8181/onos/v1/"
15. links = []
16. switches = []
17. hosts = []
18. name_switches = {}
19.
20. def netcfg(net):
21.     # Download the current network configuration
22.     r_get = requests.get(url=URL+'network/configuration', auth=('onos', 'rocks'))
23.
24.     if r_get.status_code == 200:
25.         r_json = r_get.json()
26.
27.         for h in net.hosts:
28.             for host in hosts:
29.                 if h.MAC() == host['uri']:
30.                     r_json['hosts'][h.MAC()+'/None'] = {
31.                         "basic": {

```



```

32.                                     "latitude":
33. host['latitude'],                                     "longitude":
34. host['longitude']                                     }
35.                                     }
36.         s_keys = r_json['devices'].keys()
37.         for switch in switches:
38.             for s_key in s_keys:
39.                 if switch['uri'] == s_key:
40.                     r_json['devices'][s_key] = {
41.                         "basic": {
42.                             "name":
43. name_switches[s_key],
44.                             "latitude":
45. switch['latitude'],
46.                             "longitude":
47. switch['longitude']
48.                         }
49.                     }
50.         for net_link in net.links:
51.             for link in links:
52.                 intf1 = str(net_link.intf1)
53.                 intf2 = str(net_link.intf2)
54.                 src = intf1.split('-')[0]
55.                 src_intf = intf1.split('-')[1]
56.                 dst = intf2.split('-')[0]
57.                 dst_intf = intf2.split('-')[1]
58.                 if src == link['src'] and dst == link['dst']:
59.                     # One way link
60.                     r_json['links'][link['src_uri']+'/' +src_intf.split('eth')[1]+'-'+link['dst_uri']+'/' +dst_intf.split('eth')[1]] = {
61.                         "basic": {
62.                             "bandwidth":
63. link['bandwidth'],
64.                             "durable": True,
65.                             "bidirectional": True
66.                         }
67.                     }
68.                     # Way back link
69.                     r_json['links'][link['dst_uri']+'/' +dst_intf.split('eth')[1]+'-'+link['src_uri']+'/' +src_intf.split('eth')[1]] = {
70.                         "basic": {
71.                             "bandwidth":
72. link['bandwidth'],
73.                             "durable": True,
74.                             "bidirectional": True
75.                         }
76.                     }
77.         # Upload the new network configuration
78.         r_post = requests.post(url=URL+'network/configuration', headers={'Content-type':
79. 'application/json', 'Accept': 'application/json'}, data=json.dumps(r_json), auth=('onos', 'rocks'))
80.         if r_post.status_code != 200:
81.             print r_post.content
82.         # Helper for hosts' location
83.         def addHost(uri, latitude, longitude):
84.             hosts.append({
85.                 "uri": uri,
86.                 "latitude": latitude,
87.                 "longitude": longitude
88.             })
89.
90.         # Helper for devices' location
91.         def addSwitch(uri, latitude, longitude, name):
92.             switches.append({
93.                 "uri": uri,
94.                 "latitude": latitude,
95.                 "longitude": longitude
96.             })
97.
98.         name_switches[uri] = name
99.
100.         # Helper for bandwidth management
101.         def addLink(src, src_uri, dst, dst_uri, bandwidth):
102.             links.append({
103.                 "src": src,

```

```

105.         "src_uri": src_uri,
106.         "dst": dst,
107.         "dst_uri": dst_uri,
108.         "bandwidth": bandwidth
109.     })
110.
111. def ping(net):
112.     i = 1
113.     while i < len(net.hosts):
114.         net.hosts[i-1].cmd('ping -c4 %s' %net.hosts[i].IP())
115.         i+=2
116.
117.     if len(net.hosts)%2 != 0:
118.         net.hosts[0].cmd('ping -c4 %s' %net.hosts[len(net.hosts)-1].IP())

```

The second main block, shown in table C.4, is the main method of the script, which creates the Mininet topology by passing the class containing the hosts and devices (*LongTopo*), the IP address and the port of the controller as well as some other additional parameters. Then, a ping is done involving all hosts in such a way that they are already discovered and can be accessed if needed, apart from showing the full topology in the GUI from the very first moment. This ping, sending just four packets, is done using the default forwarding application (fwd), since the routing-app application is not yet installed and there is no need to use a more complex one. Finally, the network configuration aforementioned is uploaded.

Table. C.4 Python main method for every topology script

```

1.  def main():
2.      net = Mininet(topo=LongTopo(), controller=None, autoSetMacs=True, link=TCLink)
3.      net.addController('co', controller=RemoteController, ip='192.168.99.106', port=6633)
4.      net.start()
5.
6.      print "*****"
7.      print "Carrying out a PING between hosts. This might take a while"
8.      print "*****"
9.
10.     # Make a ping between hosts to make them visible to the onos controller
11.     ping(net)
12.
13.     # Uninstall forwarding app (org.onosproject.fwd) after the ping
14.     r_delete = requests.delete(url=URL+'applications/org.onosproject.fwd/active', auth=('onos',
15. 'rocks'))
16.
17.     # Load the network configuration
18.     netcfg(net)
19.
20.     CLI(net)
21.     net.stop()
22.
23. if __name__ == '__main__':
24.     # Clean cache of previous topologies
25.     os.system('sudo mn -c')
26.     setLogLevel('info')
27.     main()

```

When it comes to configuring the network, the Mininet API previously mentioned has some methods that allow us to add hosts, switches, and links. As can be seen in tables C.3 and C.4, alongside the creation of all network components, the necessary information to upload the network configuration is prepared. The methods in charge of this are *addHost*, *addSwitch*, and *addLink*. Lastly, because links in Mininet are bidirectional, when uploading the configuration to

ONOS it will be necessary to tell the existence of two links (one in each direction), as shown below each *self.addLink* method.

C.1.1. Topology of sections 3.3. and 3.4

Table. C.5 Mininet topology for using Dijkstra as implemented in ONOS and using a custom routing based on link occupancies

```
#!/usr/bin/python

from mininet.topo import Topo
from mininet.cli import CLI
from mininet.link import TCLink
from mininet.log import setLogLevel
from mininet.net import Mininet
from mininet.node import RemoteController, Host
from mininet.util import quietRun
from os import listdir, environ
from requests import put
import re, socket, fcntl, array, struct, sys, requests, json, os

URL = "http://192.168.99.106:8181/onos/v1/"
links = []
switches = []
hosts = []
name_switches = {}

def netcfg(net):
    # Download the current network configuration
    r_get = requests.get(url=URL+'network/configuration', auth=('onos', 'rocks'))

    if r_get.status_code == 200:
        r_json = r_get.json()

        for h in net.hosts:
            for host in hosts:
                if h.MAC() == host['uri']:
                    r_json['hosts'][h.MAC()+'/None'] = {
                        "basic": {
                            "latitude": host['latitude'],
                            "longitude": host['longitude']
                        }
                    }

        s_keys = r_json['devices'].keys()

        for switch in switches:
            for s_key in s_keys:
                if switch['uri'] == s_key:
                    r_json['devices'][s_key] = {
                        "basic": {
                            "name": name_switches[s_key],
                            "latitude": switch['latitude'],
                            "longitude": switch['longitude']
                        }
                    }

        for net_link in net.links:
            for link in links:
                intf1 = str(net_link.intf1)
                intf2 = str(net_link.intf2)
                src = intf1.split('-')[0]
                src_intf = intf1.split('-')[1]
                dst = intf2.split('-')[0]
                dst_intf = intf2.split('-')[1]

                if src == link['src'] and dst == link['dst']:
```

```

# One way Link
r_json['links'][link['src_uri']+'/'+src_intf.split('eth')[1]+'-'+link['dst_uri']+'/'+dst_intf.split('eth')[1]] = {
    "basic": {
        "bandwidth": link['bandwidth'],
        "durable": True,
        "bidirectional": True
    }
}

# Way back Link
r_json['links'][link['dst_uri']+'/'+dst_intf.split('eth')[1]+'-'+link['src_uri']+'/'+src_intf.split('eth')[1]] = {
    "basic": {
        "bandwidth": link['bandwidth'],
        "durable": True,
        "bidirectional": True
    }
}

# Upload the new network configuration
r_post = requests.post(url=URL+'network/configuration', headers={'Content-type':
'application/json', 'Accept': 'application/json'}, data=json.dumps(r_json), auth=('onos', 'rocks'))

if r_post.status_code != 200:
    print(r_post.content)

# Helper for hosts' Location
def addHost(uri, latitude, longitude):
    hosts.append({
        "uri": uri,
        "latitude": latitude,
        "longitude": longitude
    })

# Helper for devices' Location
def addSwitch(uri, latitude, longitude, name):
    switches.append({
        "uri": uri,
        "latitude": latitude,
        "longitude": longitude
    })

    name_switches[uri] = name

# Helper for bandwidth management
def addLink(src, src_uri, dst, dst_uri, bandwidth):
    links.append({
        "src": src,
        "src_uri": src_uri,
        "dst": dst,
        "dst_uri": dst_uri,
        "bandwidth": bandwidth
    })

def ping(net):
    i = 1
    while i < len(net.hosts):
        net.hosts[i-1].cmd('ping -c4 %s' %net.hosts[i].IP())
        i+=2

    if len(net.hosts)%2 != 0:
        net.hosts[0].cmd('ping -c4 %s' %net.hosts[len(net.hosts)-1].IP())

class LongTopo(Topo):
    def __init__(self, *args, **kwargs):
        Topo.__init__(self, *args, **kwargs)

        # Add hosts and switches

```

```

h1 = self.addHost('h1')
addHost('00:00:00:00:00:01', '41.58', '-8')
h2 = self.addHost('h2')
addHost('00:00:00:00:00:02', '39.58', '-8')
h3 = self.addHost('h3')
addHost('00:00:00:00:00:03', '37.58', '-8')
h4 = self.addHost('h4')
addHost('00:00:00:00:00:04', '41.5', '1.7')
h5 = self.addHost('h5')
addHost('00:00:00:00:00:05', '39.58', '2.8')
h6 = self.addHost('h6')
addHost('00:00:00:00:00:06', '39.1', '1.3')
h7 = self.addHost('h7')
addHost('00:00:00:00:00:07', '39.1', '1.3')
h8 = self.addHost('h8')
addHost('00:00:00:00:00:08', '39.1', '1.3')
h9 = self.addHost('h9')
addHost('00:00:00:00:00:09', '39.1', '1.3')
h10 = self.addHost('h10')
addHost('00:00:00:00:00:0a', '39.1', '1.3')

s1 = self.addSwitch('s1', protocols='OpenFlow13')
addSwitch('of:0000000000000001', '39.58', '-5.82', 'S1')
s2 = self.addSwitch('s2', protocols='OpenFlow13')
addSwitch('of:0000000000000002', '39.93', '-3.44', 'S2')
s3 = self.addSwitch('s3', protocols='OpenFlow13')
addSwitch('of:0000000000000003', '39.23', '-3.44', 'S3')
s4 = self.addSwitch('s4', protocols='OpenFlow13')
addSwitch('of:0000000000000004', '42.38', '-5.82', 'S4')
s5 = self.addSwitch('s5', protocols='OpenFlow13')
addSwitch('of:0000000000000005', '42.38', '-3.44', 'S5')
s6 = self.addSwitch('s6', protocols='OpenFlow13')
addSwitch('of:0000000000000006', '42.38', '-1', 'S6')
s7 = self.addSwitch('s7', protocols='OpenFlow13')
addSwitch('of:0000000000000007', '39.58', '-1', 'S7')

# Add Links
self.addLink(h1, s1)
self.addLink(h2, s1)
self.addLink(h3, s1)
self.addLink(h4, s1)
self.addLink(h5, s1)
self.addLink(s1, s2, cls=TCLink, bw=100)
addLink('s1', 'of:0000000000000001', 's2', 'of:0000000000000002', 1)
addLink('s2', 'of:0000000000000002', 's1', 'of:0000000000000001', 1)
self.addLink(s1, s3, cls=TCLink, bw=100)
addLink('s1', 'of:0000000000000001', 's3', 'of:0000000000000003', 1)
addLink('s3', 'of:0000000000000003', 's1', 'of:0000000000000001', 1)
self.addLink(s2, s7, cls=TCLink, bw=100)
addLink('s2', 'of:0000000000000002', 's7', 'of:0000000000000007', 1)
addLink('s7', 'of:0000000000000007', 's2', 'of:0000000000000002', 1)
self.addLink(s3, s7, cls=TCLink, bw=100)
addLink('s3', 'of:0000000000000003', 's7', 'of:0000000000000007', 1)
addLink('s7', 'of:0000000000000007', 's3', 'of:0000000000000003', 1)
self.addLink(s1, s4, cls=TCLink, bw=100)
addLink('s1', 'of:0000000000000001', 's4', 'of:0000000000000004', 1)
addLink('s4', 'of:0000000000000004', 's1', 'of:0000000000000001', 1)
self.addLink(s4, s5, cls=TCLink, bw=100)
addLink('s4', 'of:0000000000000004', 's5', 'of:0000000000000005', 1)
addLink('s5', 'of:0000000000000005', 's4', 'of:0000000000000004', 1)
self.addLink(s5, s6, cls=TCLink, bw=100)
addLink('s5', 'of:0000000000000005', 's6', 'of:0000000000000006', 1)
addLink('s6', 'of:0000000000000006', 's5', 'of:0000000000000005', 1)
self.addLink(s6, s7, cls=TCLink, bw=100)
addLink('s6', 'of:0000000000000006', 's7', 'of:0000000000000007', 1)
addLink('s7', 'of:0000000000000007', 's6', 'of:0000000000000006', 1)
self.addLink(s7, h6)
self.addLink(s7, h7)
self.addLink(s7, h8)
self.addLink(s7, h9)
self.addLink(s7, h10)

```

```

topos = { 'mytopo': ( lambda: LongTopo() ) }

def main():
    net = Mininet(topo=LongTopo(), controller=None, autoSetMacs=True, link=TCLink)
    net.addController('co', controller=RemoteController, ip='192.168.99.106', port=6633)
    net.start()

    # Make a ping between hosts to make them visible to the onos controller
    ping(net)

    # Uninstall forwarding app (org.onosproject.fwd) after the ping
    r_delete = requests.delete(url=URL+'applications/org.onosproject.fwd/active', auth=('onos',
'rocks'))

    # Load the network configuration
    netcfg(net)

    CLI(net)
    net.stop()

if __name__ == '__main__':
    # Clean cache of previous topologies
    os.system('sudo mn -c')
    setLogLevel('info')
    main()

```

C.1.2. Reinforcement Learning topology

Table. C.6 Mininet topology for the RL scenario

```

#!/usr/bin/python

from mininet.topo import Topo
from mininet.cli import CLI
from mininet.link import TCLink
from mininet.log import setLogLevel
from mininet.net import Mininet
from mininet.node import RemoteController, Host
from mininet.util import quietRun
from os import listdir, environ
from requests import put
import re, socket, fcntl, array, struct, sys, requests, json, os

URL = "http://192.168.99.106:8181/onos/v1/"
links = []
switches = []
hosts = []
name_switches = {}

def netcfg(net):
    # Download the current network configuration
    r_get = requests.get(url=URL+'network/configuration', auth=('onos', 'rocks'))

    if r_get.status_code == 200:
        r_json = r_get.json()

        for h in net.hosts:
            for host in hosts:
                if h.MAC() == host['uri']:
                    r_json['hosts'][h.MAC()+'/None'] = {
                        "basic": {
                            "latitude": host['latitude'],
                            "longitude": host['longitude']
                        }
                    }

```

```

s_keys = r_json['devices'].keys()

for switch in switches:
    for s_key in s_keys:
        if switch['uri'] == s_key:
            r_json['devices'][s_key] = {
                "basic": {
                    "name": name_switches[s_key],
                    "latitude": switch['latitude'],
                    "longitude": switch['longitude']
                }
            }

for net_link in net.links:
    for link in links:
        intf1 = str(net_link.intf1)
        intf2 = str(net_link.intf2)
        src = intf1.split('-')[0]
        src_intf = intf1.split('-')[1]
        dst = intf2.split('-')[0]
        dst_intf = intf2.split('-')[1]

        if src == link['src'] and dst == link['dst']:
            # One way Link

r_json['links'][link['src_uri']+'/' +src_intf.split('eth')[1]+'-'+link['dst_uri']+'/' +dst_intf.split('eth')[1]] = {
    "basic": {
        "bandwidth": link['bandwidth'],
        "durable": True,
        "bidirectional": True
    }
}
# Way back Link

r_json['links'][link['dst_uri']+'/' +dst_intf.split('eth')[1]+'-'+link['src_uri']+'/' +src_intf.split('eth')[1]] = {
    "basic": {
        "bandwidth": link['bandwidth'],
        "durable": True,
        "bidirectional": True
    }
}

# Upload the new network configuration
r_post = requests.post(url=URL+'network/configuration', headers={'Content-type':
'application/json', 'Accept': 'application/json'}, data=json.dumps(r_json), auth=('onos', 'rocks'))

if r_post.status_code != 200:
    print(r_post.content)

# Helper for hosts' Location
def addHost(uri, latitude, longitude):
    hosts.append({
        "uri": uri,
        "latitude": latitude,
        "longitude": longitude
    })

# Helper for devices' Location
def addSwitch(uri, latitude, longitude, name):
    switches.append({
        "uri": uri,
        "latitude": latitude,
        "longitude": longitude
    })

    name_switches[uri] = name

# Helper for bandwidth management
def addLink(src, src_uri, dst, dst_uri, bandwidth):

```

```

links.append({
    "src": src,
    "src_uri": src_uri,
    "dst": dst,
    "dst_uri": dst_uri,
    "bandwidth": bandwidth
})

def ping(net):
    i = 1
    while i < len(net.hosts):
        net.hosts[i-1].cmd('ping -c4 %s' %net.hosts[i].IP())
        i+=2

    if len(net.hosts)%2 != 0:
        net.hosts[0].cmd('ping -c4 %s' %net.hosts[len(net.hosts)-1].IP())

class LongTopo(Topo):

    def __init__(self, *args, **kwargs):
        Topo.__init__(self, *args, **kwargs)

        # Add hosts and switches
        h1 = self.addHost('h1')
        addHost('00:00:00:00:00:01', '41.58', '-8')
        h2 = self.addHost('h2')
        addHost('00:00:00:00:00:02', '39.58', '-8')
        h3 = self.addHost('h3')
        addHost('00:00:00:00:00:03', '37.58', '-8')
        h4 = self.addHost('h4')
        addHost('00:00:00:00:00:04', '41.5', '1.7')
        h5 = self.addHost('h5')
        addHost('00:00:00:00:00:05', '39.58', '2.8')
        h6 = self.addHost('h6')
        addHost('00:00:00:00:00:06', '39.1', '1.3')
        h7 = self.addHost('h7')
        addHost('00:00:00:00:00:07', '39.1', '1.3')
        h8 = self.addHost('h8')
        addHost('00:00:00:00:00:08', '39.1', '1.3')
        h9 = self.addHost('h9')
        addHost('00:00:00:00:00:09', '39.1', '1.3')
        h10 = self.addHost('h10')
        addHost('00:00:00:00:00:0a', '39.1', '1.3')

        s1 = self.addSwitch('s1', protocols='OpenFlow13')
        addSwitch('of:0000000000000001', 'S1')
        s2 = self.addSwitch('s2', protocols='OpenFlow13')
        addSwitch('of:0000000000000002', 'S2')
        s3 = self.addSwitch('s3', protocols='OpenFlow13')
        addSwitch('of:0000000000000003', 'S3')
        s4 = self.addSwitch('s4', protocols='OpenFlow13')
        addSwitch('of:0000000000000004', 'S4')
        s5 = self.addSwitch('s5', protocols='OpenFlow13')
        addSwitch('of:0000000000000005', 'S5')
        s6 = self.addSwitch('s6', protocols='OpenFlow13')
        addSwitch('of:0000000000000006', 'S6')
        s7 = self.addSwitch('s7', protocols='OpenFlow13')
        addSwitch('of:0000000000000007', 'S7')
        s8 = self.addSwitch('s8', protocols='OpenFlow13')
        addSwitch('of:0000000000000008', 'S8')

        # Add Links
        self.addLink(h1, s1)
        self.addLink(h2, s1)
        self.addLink(h3, s1)
        self.addLink(h4, s1)
        self.addLink(h5, s1)
        self.addLink(s1, s2, cls=TCLink, bw=2000)
        addLink('s1', 'of:0000000000000001', 's2', 'of:0000000000000002', 1)
        addLink('s2', 'of:0000000000000002', 's1', 'of:0000000000000001', 1)
        self.addLink(s1, s3, cls=TCLink, bw=2000)
        addLink('s1', 'of:0000000000000001', 's3', 'of:0000000000000003', 1)

```



```

        addLink('s3', 'of:0000000000000003', 's1', 'of:0000000000000001', 1)
        self.addLink(s1, s4, cls=TCLink, bw=2000)
        addLink('s1', 'of:0000000000000001', 's4', 'of:0000000000000004', 1)
        addLink('s4', 'of:0000000000000004', 's1', 'of:0000000000000001', 1)
        self.addLink(s2, s5, cls=TCLink, bw=1000)
        addLink('s2', 'of:0000000000000002', 's5', 'of:0000000000000005', 1)
        addLink('s5', 'of:0000000000000005', 's2', 'of:0000000000000002', 1)
        self.addLink(s2, s7, cls=TCLink, bw=1000)
        addLink('s2', 'of:0000000000000002', 's7', 'of:0000000000000007', 1)
        addLink('s7', 'of:0000000000000007', 's2', 'of:0000000000000002', 1)
        self.addLink(s3, s6, cls=TCLink, bw=2000)
        addLink('s3', 'of:0000000000000003', 's6', 'of:0000000000000006', 1)
        addLink('s6', 'of:0000000000000006', 's3', 'of:0000000000000003', 1)
        self.addLink(s4, s5, cls=TCLink, bw=1000)
        addLink('s4', 'of:0000000000000004', 's5', 'of:0000000000000005', 1)
        addLink('s5', 'of:0000000000000005', 's4', 'of:0000000000000004', 1)
        self.addLink(s4, s7, cls=TCLink, bw=1000)
        addLink('s4', 'of:0000000000000004', 's7', 'of:0000000000000007', 1)
        addLink('s7', 'of:0000000000000007', 's4', 'of:0000000000000004', 1)
        self.addLink(s8, h6)
        self.addLink(s8, h7)
        self.addLink(s8, h8)
        self.addLink(s8, h9)
        self.addLink(s8, h10)

topos = { 'mytopo': ( lambda: LongTopo() ) }

def main():
    net = Mininet(topo=LongTopo(), controller=None, autoSetMacs=True, link=TCLink)
    net.addController('co', controller=RemoteController, ip='192.168.99.106', port=6633)
    net.start()

    # Make a ping between hosts to make them visible to the onos controller
    ping(net)

    # Uninstall forwarding app (org.onosproject.fwd) after the ping
    r_delete = requests.delete(url=URL+'applications/org.onosproject.fwd/active', auth=('onos',
'rocks'))

    # Load the network configuration
    netcfg(net)

    CLI(net)
    net.stop()

if __name__ == '__main__':
    # Clean cache of previous topologies
    os.system('sudo mn -c')
    setLogLevel('info')
    main()

```

C.2. Grafana

With regards to the visualization of the data, the script in charge of processing it is shown in table C.7. Once the database is created, the measurements along with their values will be added automatically by the collector.py script. The three that have been created to monitor the network are **flow_active_entries**, **link_occupations**, and **port_load**. As can be seen in the figures C.2, C.3, and C.4, every measurement in InfluxDB can have different *tags* (static values) and *fields* (dynamic values), of which will be adapted according to what needs to be represented. Together with every entry in the database, a timestamp is associated at which the measure was taken, which will allow us to present the data in Grafana according to time.

```
> SELECT * FROM flow_active_entries
name: flow_active_entries
time                device percentage      value
----                -
1623254127530157119 S1      14.285714285714285 3
1623254127530157119 S2      14.285714285714285 3
1623254127530157119 S3      14.285714285714285 3
1623254127530157119 S4      14.285714285714285 3
1623254127530157119 S5      14.285714285714285 3
1623254127530157119 S6      14.285714285714285 3
1623254127530157119 S7      14.285714285714285 3
```

Fig. C.2 Fields from flow_active_entries measurement

```
> SELECT * FROM link_occupations
name: link_occupations
time                link                percentage
----                -
1623254127802768609 of:0000000000000001/3-of:0000000000000002/3 0
1623254127802768609 of:0000000000000001/4-of:0000000000000003/3 0
1623254127802768609 of:0000000000000001/5-of:0000000000000004/3 0
1623254127802768609 of:0000000000000002/3-of:0000000000000001/3 0
1623254127802768609 of:0000000000000002/4-of:0000000000000007/3 0
1623254127802768609 of:0000000000000003/3-of:0000000000000001/4 0
1623254127802768609 of:0000000000000003/4-of:0000000000000007/4 0
1623254127802768609 of:0000000000000004/3-of:0000000000000001/5 0
1623254127802768609 of:0000000000000004/4-of:0000000000000005/3 0
1623254127802768609 of:0000000000000005/3-of:0000000000000004/4 0
1623254127802768609 of:0000000000000005/4-of:0000000000000006/3 0
1623254127802768609 of:0000000000000006/3-of:0000000000000005/4 0
1623254127802768609 of:0000000000000006/4-of:0000000000000007/5 0
1623254127802768609 of:0000000000000007/3-of:0000000000000002/4 0
1623254127802768609 of:0000000000000007/4-of:0000000000000003/4 0
1623254127802768609 of:0000000000000007/5-of:0000000000000006/4 0
```

Fig. C.3 Fields from link_occupations measurement

```
> SELECT * FROM port_load
name: port_load
time          device          port rate
----          -
1623254127802768609 of:000000000000000001 3 0
1623254127802768609 of:000000000000000001 4 0
1623254127802768609 of:000000000000000001 5 0
1623254127802768609 of:000000000000000002 3 0
1623254127802768609 of:000000000000000002 4 0
1623254127802768609 of:000000000000000003 3 0
1623254127802768609 of:000000000000000003 4 0
1623254127802768609 of:000000000000000004 3 0
1623254127802768609 of:000000000000000004 4 0
1623254127802768609 of:000000000000000005 3 0
1623254127802768609 of:000000000000000005 4 0
1623254127802768609 of:000000000000000006 3 0
1623254127802768609 of:000000000000000006 4 0
1623254127802768609 of:000000000000000007 3 0
1623254127802768609 of:000000000000000007 4 0
1623254127802768609 of:000000000000000007 5 0
```

Fig. C.4 Fields from `port_load` measurement

Initially, the script drops all points from every measurement from the onos database, so any pre-existing data is removed every time the script is initialized to capture data, and also downloads the current network configuration (see Fig C.5). Then, owing to that every **5 seconds** the points from each measurement are updated, two scheduled threads are created having as a target the methods *load_statistics* and *flow_active_entries*, respectively.

On the one hand, the first method uses the statistics available for every port (see Fig C.6) to calculate the rate detected as well as the link occupation. The same as in the main Java class of the application, a weighted simple moving average of **size 4** is used to smooth possible outliers when calculating the data rates. On the other hand, the second method uses the statistics available for all flow active entries (see Fig C.7) in each device to show the percentage distribution of flows. This will allow us to monitor the load, in terms of the number of installed flow rules, of each device in the network.

GET /network/configuration Gets entire network configuration base

Response Messages

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

[Try it out!](#) [Hide Response](#)

Curl

```
curl -X GET --header 'Accept: application/json' 'http://192.168.99.106:8181/onos/v1/network/configuration'
```

Request URL

```
http://192.168.99.106:8181/onos/v1/network/configuration
```

Response Body

```
{
  "devices": {
    "of:000000000000000004": {
      "basic": {
        "latitude": "42.38",
        "name": "S4",
        "longitude": "-5.82"
      },
      "classifiers": [
        {
          "ethernet-type": "LLDP",
          "target-queue": 0
        },
        {
          "ethernet-type": "BDDP",
          "target-queue": 0
        }
      ]
    },
    "of:000000000000000005": {
```

Response Code

```
200
```

Response Headers

```
{
  "content-length": "4313",
  "content-type": "application/json",
  "server": "Jetty(9.4.22.v20191022)"
}
```

Fig. C.5 Request to get the entire network configuration

GET

/statistics/ports/{deviceId}/{port}

Gets port statistics of a specified device and port

Parameters

Parameter	Value	Description	Parameter Type	Data Type
deviceId	of:0000000000000001	device ID	path	string
port	5	port	path	string

Response Messages

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

Try it out!

Hide Response

Curl

```
curl -X GET --header 'Accept: application/json' 'http://192.168.99.106:8181/onos/v1/statistics/ports/of%3A0000000000000001/5'
```

Request URL

```
http://192.168.99.106:8181/onos/v1/statistics/ports/of%3A0000000000000001/5
```

Response Body

```
{
  "statistics": [
    {
      "device": "of:0000000000000001",
      "ports": [
        {
          "port": 5,
          "packetsReceived": 386,
          "packetsSent": 341,
          "bytesReceived": 20998,
          "bytesSent": 19042,
          "packetsRxDropped": 0,
          "packetsTxDropped": 0,
          "packetsRxErrors": 0,
          "packetsTxErrors": 0,
          "durationSec": 32
        }
      ]
    }
  ]
}
```

Response Code

```
200
```

Fig. C.6 Request to get the statistics of a specified device and port

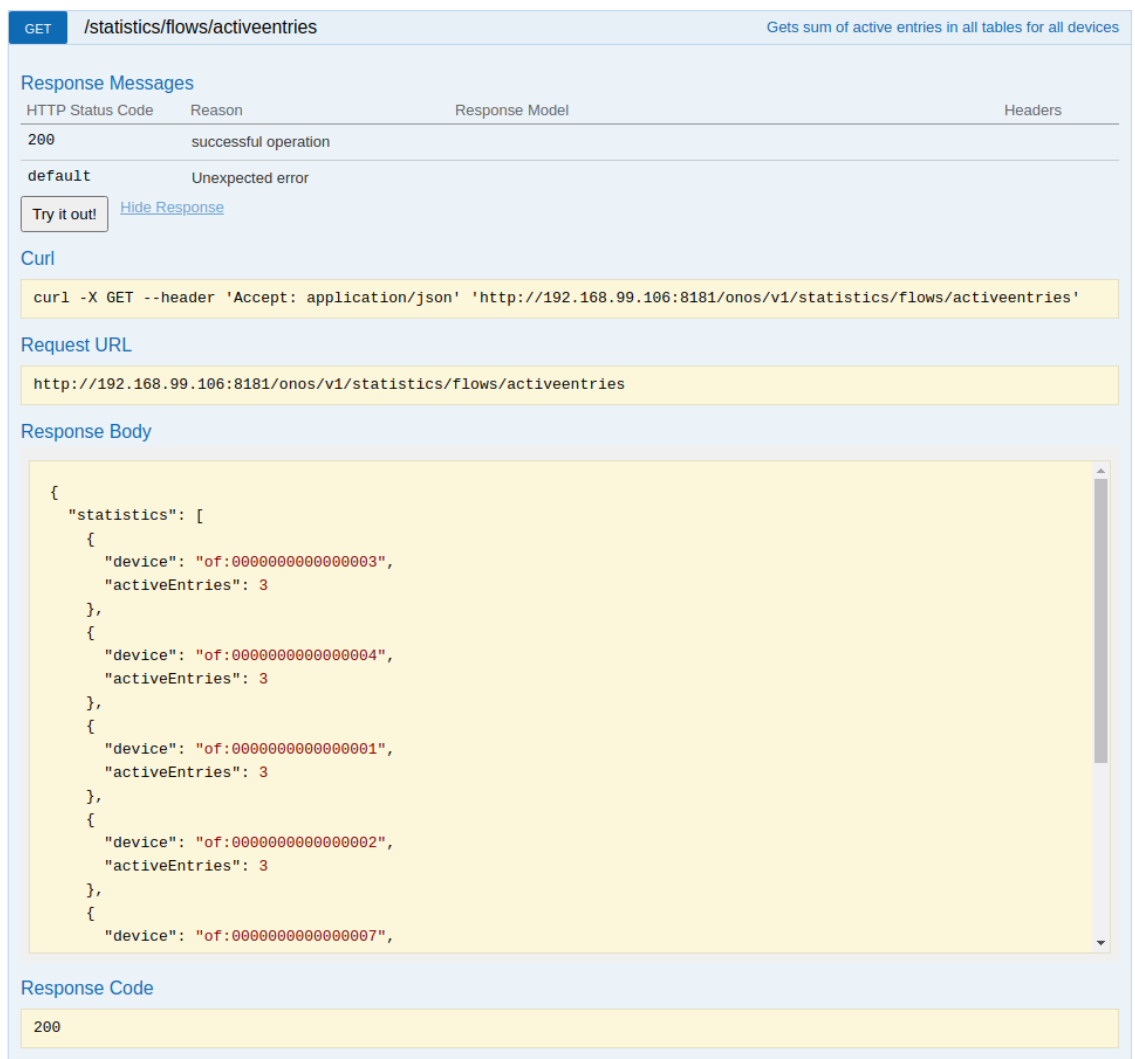


Fig. C.7 Request to get the sum of active flow entries in a device

Table. C.7 Python script to monitor the network in Grafana (collector.py)

```
#!/usr/bin/python

from __future__ import division
from influxdb import InfluxDBClient
from mininet.net import Mininet
from threading import Thread
import json, requests, time

client = InfluxDBClient(host='localhost', port=8086)
URL = "http://192.168.99.106:8181/onos/v1/"
POLL_INTERVAL = 5
WINDOW_SIZE = 4
loads = []
active_entries = []
network_configuration = []
devices = {}
links = {}
previous_port_statistics = {}
current_port_statistics = {}
weightedAverageConnectPointRate = {}
```

```

def configuration():
    client.switch_database('onos')
    client.query('DROP SERIES FROM /*/*')

    r_get = requests.get(url=URL+'network/configuration', auth=('onos', 'rocks'))

    if r_get.status_code == 200:
        network_configuration = r_get.json()

        for device in network_configuration['devices'].keys():
            devices[device] = network_configuration['devices'][device]['basic']['name']
        for link in network_configuration['links'].keys():
            links[link] = network_configuration['links'][link]['basic']['bandwidth']

def flow_active_entries():
    while True:
        start = time.time()
        r_get = requests.get(url=URL+'statistics/flows/activeentries', auth=('onos', 'rocks'))

        if r_get.status_code == 200:
            r_json = r_get.json()
            active_entries = r_json['statistics']

            json_body = []
            total_entries = sum(map(lambda x: int(x['activeEntries']), active_entries))
            for active_entry in active_entries:
                json_body.append({
                    "measurement": "flow_active_entries",
                    "tags": {
                        "device": devices[active_entry['device']]
                    },
                    "fields": {
                        "value": active_entry['activeEntries'],
                        "percentage":
(active_entry['activeEntries']/total_entries)*100
                    }
                })
            res = client.write_points(json_body)
            time.sleep(POLL_INTERVAL)

def load_statistics():
    while True:
        start = time.time()
        json_body = []
        for link in links.keys():
            r_get_dst =
requests.get(url=URL+'statistics/ports/'+link.split("-")[1].split("/")[0]+'/' + link.split("-")[1].split("/")
[1], auth=('onos', 'rocks'))

            if r_get_dst.status_code == 200:
                r_json_dst = r_get_dst.json()
                previous_port_statistics[link.split("-")[1]] =
current_port_statistics[link.split("-")[1]]
                current_port_statistics[link.split("-")[1]] =
r_json_dst['statistics'][0]['ports'][0]
                dst_rate = calculateAverageWeightedRate(link.split("-")[1],
float((current_port_statistics[link.split("-")[1]]['bytesReceived']-previous_port_statistics[link.split("-"
)[1]]['bytesReceived'])/(POLL_INTERVAL)))
                dst_occupancy = float(((dst_rate*8)/(1000000))/(links[link]))
                json_body.append({
                    "measurement": "port_load",
                    "tags": {
                        "device": link.split("-")[0].split("/")[0],
                        "port": link.split("-")[0].split("/")[1]
                    },
                    "fields": {
                        "rate": (dst_rate*8)/(1000000)
                    }
                })
            json_body.append({
                "measurement": "link_occupations",

```

```

        "tags": {
            "link": link
        },
        "fields": {
            "percentage": dst_occupancy*100
        }
    })

    res = client.write_points(json_body)
    time.sleep(POLL_INTERVAL)

def calculateAverageWeightedRate(connectPoint, rate):
    cpRates = weightedAverageConnectPointRate[connectPoint]
    sum = 0
    i = 0

    cpRates.append(rate)
    if len(cpRates) > WINDOW_SIZE:
        cpRates.pop(0)
    weightedAverageConnectPointRate[connectPoint] = cpRates

    for rate in cpRates:
        i += 1
        sum = sum+(rate*i)

    return sum/((i*(i+1))/2)

def initialise_port_statistics():
    for link in links.keys():
        r_get_src =
requests.get(url=URL+'statistics/ports/'+link.split("-")[0].split("/")[0]+'/' +link.split("-")[0].split("/")
[1], auth=('onos', 'rocks'))
        r_get_dst =
requests.get(url=URL+'statistics/ports/'+link.split("-")[1].split("/")[0]+'/' +link.split("-")[1].split("/")
[1], auth=('onos', 'rocks'))

        if r_get_src.status_code == 200 and r_get_dst.status_code == 200:
            r_json_src = r_get_src.json()
            r_json_dst = r_get_dst.json()
            previous_port_statistics[link.split("-")[0]] =
r_json_src['statistics'][0]['ports'][0]
            previous_port_statistics[link.split("-")[1]] =
r_json_dst['statistics'][0]['ports'][0]
            current_port_statistics[link.split("-")[0]] =
r_json_src['statistics'][0]['ports'][0]
            current_port_statistics[link.split("-")[1]] =
r_json_dst['statistics'][0]['ports'][0]
            weightedAverageConnectPointRate[link.split("-")[0]] = []
            weightedAverageConnectPointRate[link.split("-")[1]] = []

if __name__ == '__main__':
    configuration()
    initialise_port_statistics()
    thread1 = Thread(target=load_statistics)
    thread1.start()
    thread2 = Thread(target=flow_active_entries)
    thread2.start()

```


C.2.1. Results of section 3.3.2

Next are shown the occupancy of the links for the switch S1:

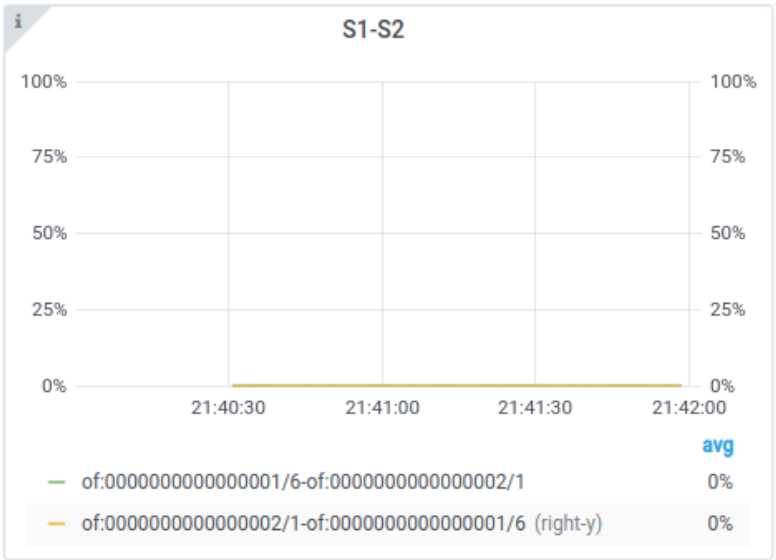


Fig. C.8 Occupancy of the link S1-S2 depicted in Grafana using Dijkstra as implemented in ONOS

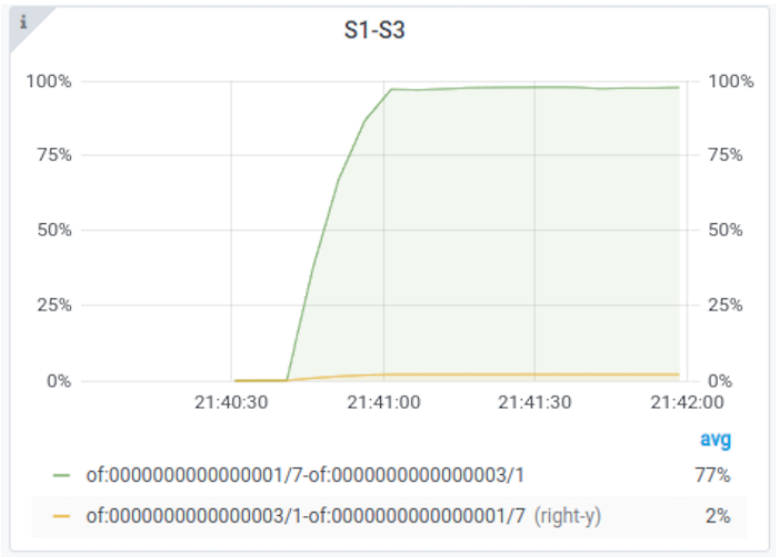


Fig. C.9 Occupancy of the link S1-S3 depicted in Grafana using Dijkstra as implemented in ONOS

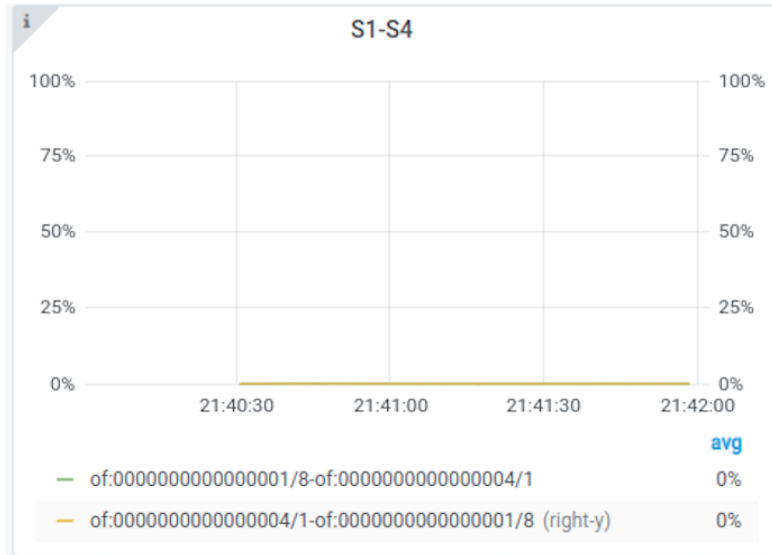


Fig. C.10 Occupancy of the link S1-S4 depicted in Grafana using Dijkstra as implemented in ONOS

C.2.2. Results of section 3.4.2

Next are shown the occupancy of the links for the switch S1:

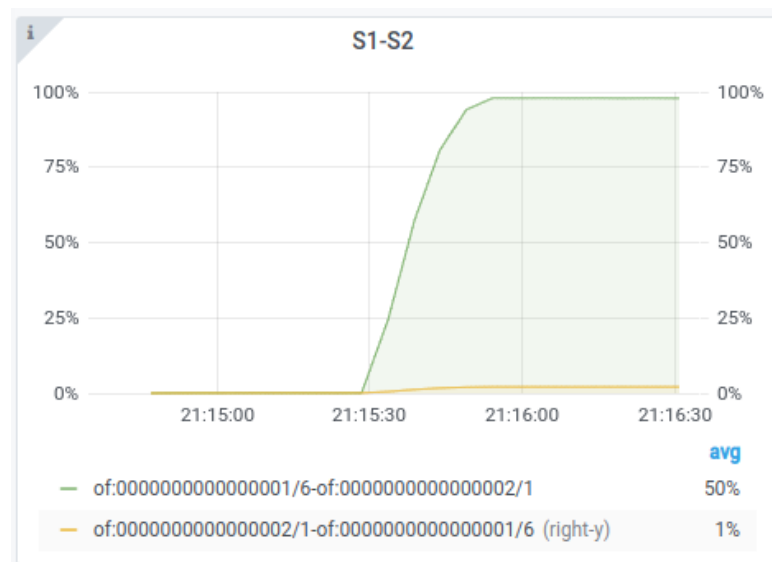


Fig. C.11 Occupancy of the link S1-S2 depicted in Grafana using a custom routing based on link occupancies

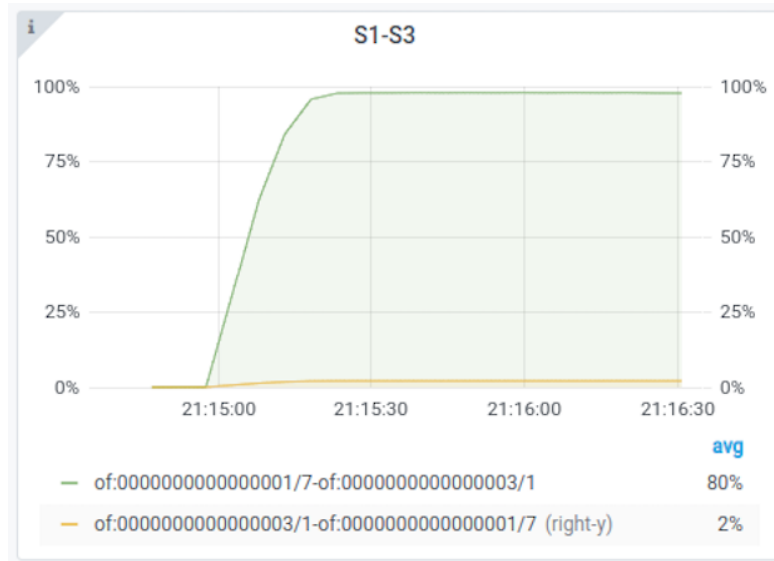


Fig. C.12 Occupancy of the link S1-S3 depicted in Grafana using a custom routing based on link occupancies

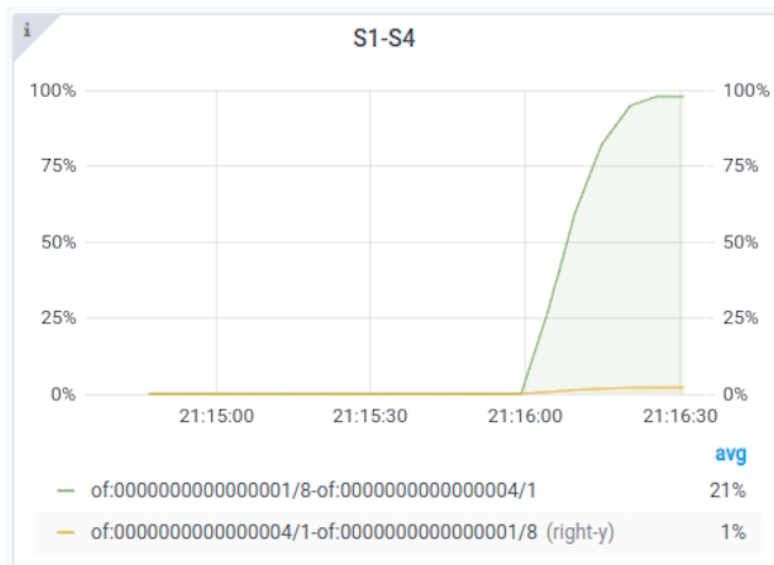


Fig. C.13 Occupancy of the link S1-S4 depicted in Grafana using a custom routing based on link occupancies

C.3. MGEN

In this last point, it is explained in detail the script used to generate the results obtained from the log files generated with every flow. These log files contain the detailed information of the flows received. The following excerpt shows, namely, the timestamp at which the packet was received, the sequence number of the packet, the source and destination IP addresses, the timestamp at which the packet was sent, and the size. Apart from the receiving event (RECV), others that are important to mention are the start (START) and stop (STOP), which

denotes the initialisation and stopping of processing transmission and reception events.

```
15:44:50.454321 RECV proto>UDP flow>1 seq>1679 src>::ffff:10.0.0.1/5001
dst>10.0.0.7/5001 sent>15:44:50.454160 size>1000
gps>INVALID,999.000000,999.000000,4294966297
```

With all of this information, a custom tool executed via a python script has been created (see Table C.9), containing most of the parameters -but not limited to- that can be extracted from this log file, which are the **throughput**, the **latency** -maximum, minimum, and average-, the **loss rate**, and the **jitter**. This tool generates a JSON file, which has the following form:

Table. C.8 Excerpt of the JSON file containing the results of MGEN

```
1.  {
2.    "h1h6": {
3.      "throughput": "0.9032Mbps",
4.      "latency": {
5.        "min": "0.014ms",
6.        "max": "466.149ms",
7.        "avg": "17.683ms"
8.      },
9.      "loss rate": "0.07%",
10.     "jitter": "0.07486ms"
11.   },
12.   "h2h7": {
13.     "throughput": "0.9032Mbps",
14.     "latency": {
15.       "min": "0.012ms",
16.       "max": "159.154ms",
17.       "avg": "2.436ms"
18.     },
19.     "loss rate": "0.04%",
20.     "jitter": "0.04712ms"
21.   },
22.   "h3h8": {
23.     "throughput": "0.9032Mbps",
24.     "latency": {
25.       "min": "0.016ms",
26.       "max": "459.851ms",
27.       "avg": "18.483ms"
28.     },
29.     "loss rate": "0.08%",
30.     "jitter": "0.02296ms"
31.   }
32. }
33. }
```

The formulae used to compute each parameter are:

$$\text{Throughput [Mbps]} = \frac{\sum_{i=0}^{N=\text{packets}} \text{bytes}_{\text{packet}_i}}{\text{transmission time}} \quad (\text{D. 1})$$

$$\text{Latency}_i [\text{ms}] = \text{received time}_i - \text{sent time}_i \quad (\text{D. 2})$$

$$\text{Loss rate [\%]} = \left(\frac{\text{packets sent} - \text{packets received}}{\text{packets sent}} \right) \cdot 100 \quad (\text{D. 3})$$

$$\text{Jitter [ms]} = \text{Jitter} + \frac{|\text{latency}_{i-1} - \text{latency}_i| - \text{Jitter}}{16} \quad (\text{D. 4})$$

Table. C.9 Python script to generate the results from MGEN log files (parser.py)

```
#!/usr/bin/env python
# coding: utf-8

from __future__ import division
from collections import OrderedDict
import re, sys, os, csv, datetime, json, shutil
import numpy as np

log_paths = []

for root, dirs, files in os.walk("/home/sdn/onos/utis/mininet/topologies/mgen/log/"+sys.argv[1]):
    for filename in sorted(files):
        log_paths.append(root+'/'+filename)

results = OrderedDict()
for path in log_paths:
    content = []
    with open(path, "r") as file:
        for line in file:
            if (line.split(" ")[1] != "START" and line.split(" ")[1] != "LISTEN"):
                content.append(line)
    content.pop(len(content)-1)

    '''
    Calculate the throughput in Mbps
    '''
    throughput = []
    bytes = 0
    start_time = datetime.datetime.strptime(content[0].split(" ")[0], '%H:%M:%S.%f')
    end_time = datetime.datetime.strptime(content[len(content)-1].split(" ")[0], '%H:%M:%S.%f')

    for line in content:
        bytes += int(line.split("size>")[1].split(" ")[0])

        throughput.append(round((bytes*8)/(round((end_time-start_time).total_seconds()*1000000), 5))

    '''
    Calculate the latency in milliseconds (min, max, & avg)
    As of now, only is taken into account the one way latency (from sender to receiver) since UDP is used
    '''
    values = []
    latency = OrderedDict()

    for line in content:
        received_time = datetime.datetime.strptime(line.split(" ")[0], '%H:%M:%S.%f')
        sent_time = datetime.datetime.strptime(line.split("sent>")[1].split(" ")[0], '%H:%M:%S.%f')
        values.append((received_time-sent_time).total_seconds()*1000)

    latency['min'] = round(np.min(values, axis=0), 3)
    latency['max'] = round(np.max(values, axis=0), 3)
    latency['avg'] = round(np.average(values, axis=0), 3)

    '''
    Calculate the loss rate in percentage
    '''
    packets_sent = content[len(content)-1].split("seq>")[1].split(" ")[0]
    packets_received = len(content)
    loss_rate = round(((int(packets_sent)-packets_received)/int(packets_sent))*100, 2)

    '''
    Calculate the jitter in miliseconds based on RFC 1889 (specifically sections 6.3.1 and A.8)
    As RTP timestamp, it is taken the sent time
    '''
    total_jitter = 0
    segment_jitter = 0
    aux = 0

    for index, line in enumerate(content):
```

```

aux+=1
received_time_j = datetime.datetime.strptime(line.split(" ")[0], '%H:%M:%S.%f')
sent_time_j = datetime.datetime.strptime(line.split("sent>")[1].split(" ")[0], '%H:%M:%S.%f')
difference_j = (received_time_j-sent_time_j).total_seconds()*1000
difference_i = 0
if index-1 == -1:
    received_time_i = 0
    sent_time_i = 0
else:
    received_time_i = datetime.datetime.strptime(content[index-1].split(" ")[0], '%H:%M:%S.%f')
    sent_time_i = datetime.datetime.strptime(content[index-1].split("sent>")[1].split(" ")[0],
'%H:%M:%S.%f')
    difference_i = (received_time_i-sent_time_i).total_seconds()*1000
difference = difference_j-difference_i
total_jitter += (abs(difference)-total_jitter)/16
segment_jitter += (abs(difference)-segment_jitter)/16

if aux == 1000:
    aux = 0
    segment_jitter = 0

'''
Create the directory if it does not exist and save all the information into a json file
'''
results_content = OrderedDict()

results_content["throughput"] = str(throughput[0])+ 'Mbps'
results_content["latency"] = OrderedDict([('min', str(latency['min'])+'ms'),
                                           ('max', str(latency['max'])+'ms'),
                                           ('avg', str(latency['avg'])+'ms')])
results_content["loss rate"] = str(loss_rate)+'%'
results_content["jitter"] = str(round(total_jitter, 5))+ 'ms'

    results[path.split("/")[-1].split(".")[0]] = results_content

if os.path.isdir("/home/sdn/onos/utils/mininet/topologies/mgen/results/"+sys.argv[1]):
    shutil.rmtree("/home/sdn/onos/utils/mininet/topologies/mgen/results/"+sys.argv[1])
os.makedirs("/home/sdn/onos/utils/mininet/topologies/mgen/results/"+sys.argv[1])

with open('/home/sdn/onos/utils/mininet/topologies/mgen/results/'+sys.argv[1]+'results.json', 'w') as f:
    json.dump(results, f, indent=4)

```

C.3.1. How to generate MGEN flows

When it comes to the generation of MGEN flows, both files (ending with the .mgn file extension) needed to generate the flow are shown, being in this case, as an example, a host with IP address 10.0.0.6 transmitting 125 packets/second of size 1000 bytes (1 Mbps) for 120 seconds:

- **sender.mgn:**
0.0 ON 1 UDP SRC 5001 DST 10.0.0.6/5001 PERIODIC [125 1000]
120.0 OFF 1
- **receiver.mgn:**
0.0 LISTEN UDP 5000-5001

With regard to Mininet, the following commands show how to generate these flows inside Mininet. The parameters in brackets are the ones to modify according to the network environment, including the name in Mininet of the receiver host, the name of the file that will contain the log, and the name of the file to store the information, respectively (the same but the other way round for the sender case):

- **receiver** → [receiver_host] mgen input [receiver.mgn] output [mgenlog.txt]
- **sender** → [sender_host] mgen input [sender.mgn]

Table. C.10 Python file to generate the JSON files for the comparison

```

import argparse, json, copy
from collections import OrderedDict
from tqdm import tqdm
import numpy as np
from Helper import Helper
from Graph import Graph

def generateFlowsAndSortThemChronologically(number, beginningNumber, maximumNumber):
    flows = []
    startTimes = []
    endTimes = []
    count = 0

    flowCount = MAX_STEPS_TRAINING
    transmissionTime = flowCount * 40 * 0.043

    possibleValues = []
    for i in range(0, flowCount * 40 + 1):
        possibleValues.append(i)

    print("number: ", number)
    for count in tqdm(range(1, flowCount + 1), ascii=True, unit="flow"):
        flow = OrderedDict()

        flow["source"] = SOURCE
        flow["destination"] = DESTINATION

        start = int(np.random.choice(possibleValues))
        end = int(np.random.choice(possibleValues))

        if count == 1:
            flow["start"] = 0
            start = 0
        else:
            while start in startTimes or start in endTimes:
                start = np.random.randint(1, flowCount * 40)

            flow["start"] = start

        # Ensuring that the total transmission time (end - start) is not greater than a 5% of
        # the total simulation time,
        # We provide a more realistic and dynamic simulation. We just want to avoid the
        # situation in which tens of flows
        # Are accumulated at the beginning
        endVisitedValues = copy.deepcopy(possibleValues)
        while end <= start or end - start > transmissionTime:
            end = int(np.random.choice(endVisitedValues))
            endVisitedValues.remove(end)

        if len(endVisitedValues) == 0:
            print("inside if")
            start = int(np.random.choice(possibleValues))
            while start in startTimes or start in endTimes:

```

```

        start = np.random.randint(1, flowCount * 40)
        endVisitedValues = copy.deepcopy(possibleValues)

        flow["start"] = start
        startTimes.append(start)
        possibleValues.remove(start)

        flow["end"] = end
        endTimes.append(end)
        possibleValues.remove(end)

        rate = 0
        multiple = False

        while not multiple:
            rate = np.random.randint(80, 251)

            if rate % 8 == 0:
                multiple = True

        flow["rate"] = rate

        flows.append(flow)

        count += 1

        if len(flows) == flowCount:
            break

    sortedFlowsByStartTime = sorted(flows, key=lambda flow: flow["start"])
    sortedFlowsByEndTime = sorted(flows, key=lambda flow: flow["end"])

    sortedFlows = []
    i, j = 0, 0

    # The hash is obtained with the following keys: source, destination, and time (either
    # start or end, depending on the case)
    while i < len(sortedFlowsByStartTime) and j < len(sortedFlowsByEndTime):
        if sortedFlowsByStartTime[i]["start"] < sortedFlowsByEndTime[j]["end"]:
            sortedFlows.append(OrderedDict())
            sortedFlows[len(sortedFlows) - 1]["hash"] =
hash((sortedFlowsByStartTime[i]["source"],
sortedFlowsByStartTime[i]["destination"],
sortedFlowsByStartTime[i]["start"],
sortedFlowsByStartTime[i]["end"]]))
            sortedFlows[len(sortedFlows) - 1]["time"] = sortedFlowsByStartTime[i]["start"]
            sortedFlows[len(sortedFlows) - 1]["ingress"] = sortedFlowsByStartTime[i]
            i += 1
        else:
            sortedFlows.append(OrderedDict())
            sortedFlows[len(sortedFlows) - 1]["hash"] =
hash((sortedFlowsByEndTime[j]["source"],
sortedFlowsByEndTime[j]["destination"],
sortedFlowsByEndTime[j]["start"],
sortedFlowsByEndTime[j]["end"]]))
            sortedFlows[len(sortedFlows) - 1]["time"] = sortedFlowsByEndTime[j]["end"]
            sortedFlows[len(sortedFlows) - 1]["egress"] = sortedFlowsByEndTime[j]
            j += 1

```



```

if i == len(sortedFlowsByStartTime):
    while j < len(sortedFlowsByEndTime):
        sortedFlows.append(OrderedDict())
        sortedFlows[len(sortedFlows) - 1]["hash"] =
hash((sortedFlowsByEndTime[j]["source"],

sortedFlowsByEndTime[j]["destination"],

sortedFlowsByEndTime[j]["start"],

sortedFlowsByEndTime[j]["end"])))
        sortedFlows[len(sortedFlows) - 1]["time"] = sortedFlowsByEndTime[j]["end"]
        sortedFlows[len(sortedFlows) - 1]["egress"] = sortedFlowsByEndTime[j]
        j += 1
    elif j == len(sortedFlowsByEndTime):
        while i < len(sortedFlowsByStartTime):
            sortedFlows.append(OrderedDict())
            sortedFlows[len(sortedFlows) - 1]["hash"] =
hash((sortedFlowsByStartTime[i]["source"],

sortedFlowsByStartTime[i]["destination"],

sortedFlowsByStartTime[i]["start"],

sortedFlowsByStartTime[i]["end"])))
            sortedFlows[len(sortedFlows) - 1]["time"] = sortedFlowsByStartTime[i]["start"]
            sortedFlows[len(sortedFlows) - 1]["ingress"] = sortedFlowsByStartTime[i]
            i += 1

hashes = []

for flow in sortedFlows:
    if "ingress" in flow:
        hashes.append(flow["hash"])

if len(hashes) > len(set(hashes)):
    exit()

maximumRate = 0
flowsStacked = OrderedDict()
for flow in sortedFlows:
    if "ingress" in flow:
        maximumRate += flow["ingress"]["rate"]

        if flow["ingress"]["rate"] not in flowsStacked:
            flowsStacked[flow["ingress"]["rate"]] = 0
        else:
            flowsStacked[flow["ingress"]["rate"]] += 1
    elif "egress" in flow:
        maximumRate -= flow["egress"]["rate"]

    if maximumRate > 5900:
        print("exceeds")
        generateFlowsAndSortThemChronologically(number, beginningNumber, maximumNumber)

print("outside exceeds")
fileNumber = str(number + beginningNumber + 1)

with open("generated-flows/flows_comparison " + fileNumber + ".json", "w") as file:
    file.write(json.dumps(sortedFlows))

graph = Graph()

```

```

graph.plotFlowsInformation(sortedFlows, "flows-generator", fileNumber)

number += 1

if number >= maximumNumber:
    exit()
else:
    generateFlowsAndSortThemChronologically(number, beginningNumber, maximumNumber)

if __name__ == "__main__":
    with open("DQN_config.json", "r") as DQN_config:
        DQN_config = json.load(DQN_config)

    # Constant variables read from the Config.json file
    SOURCE = DQN_config['SOURCE']
    DESTINATION = DQN_config['DESTINATION']
    ACTION_COUNT = DQN_config['ACTION_COUNT']
    ACTIONS = [[0, 1, 4, 7], [0, 2, 5, 7], [0, 3, 6, 7], [0, 1, 6, 7], [0, 3, 4, 7]]
    ADAM_LEARNING_RATE = DQN_config['ADAM_LEARNING_RATE']
    BATCH_SIZE = DQN_config['BATCH_SIZE']
    EPSILON_DECAY = DQN_config['EPSILON_DECAY']
    GAMMA = DQN_config['GAMMA']
    MAX_STEPS_TRAINING = DQN_config['MAX_STEPS_TRAINING']
    MAX_STEPS_COMPARISON = DQN_config['MAX_STEPS_COMPARISON']
    MAX_EPISODES_TRAINING = DQN_config['MAX_EPISODES_TRAINING']
    MAX_EPISODES_COMPARISON = DQN_config['MAX_EPISODES_COMPARISON']
    MIN_EPSILON = DQN_config['MIN_EPSILON']
    REPLAY_MEMORY_SIZE = DQN_config['REPLAY_MEMORY_SIZE']
    REPLAY_START_LEARNING_SIZE = DQN_config['REPLAY_START_LEARNING_SIZE']

    parser = argparse.ArgumentParser()
    parser.add_argument("--number", help="Number of files to generate: [integer]")
    parser.add_argument("--beginning", help="Beginning number for the files to generate:
[integer]")
    args = vars(parser.parse_args())

    Helper.createGeneratedFlowsDirectoryDirectory()

    generateFlowsAndSortThemChronologically(0, int(args["beginning"]), int(args["number"]))

```

APPENDIX D. REINFORCEMENT LEARNING TRAINING WITH DIFFERENT HYPERPARAMETERS

In this appendix we will show the average reward obtained when training with different hyperparameters to the ones shown in section 4.3, just to show the complexity of fine-tuning the RL agent. In order not to be naming every time the values used, we will just mention the ones that are different in comparison to the ones selected for the training.

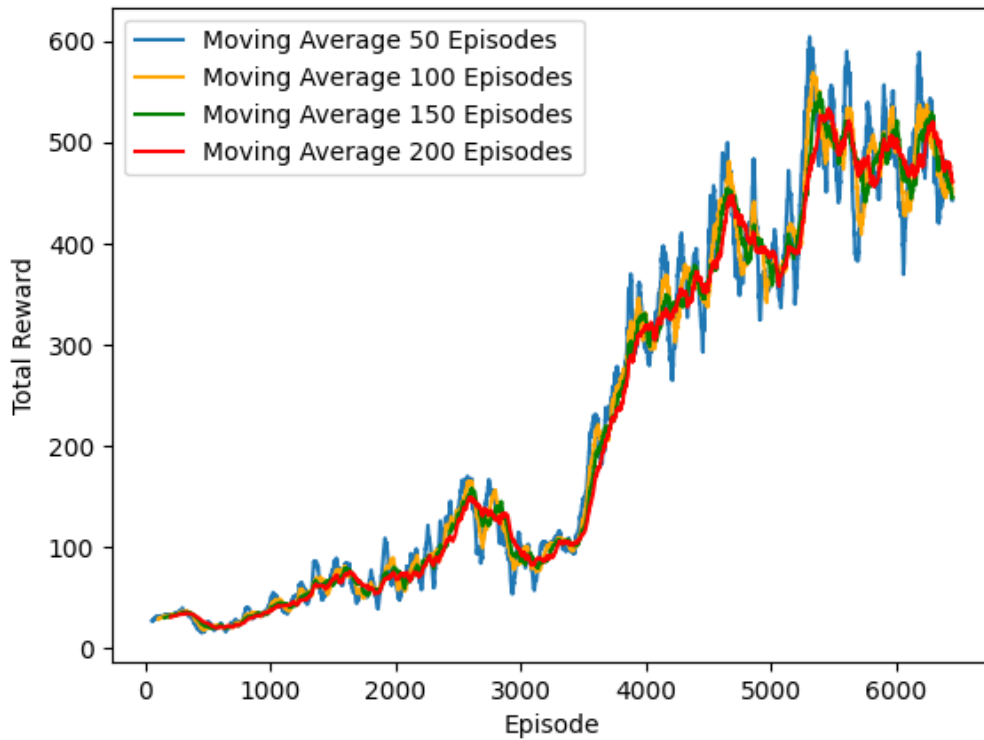


Fig. D.1 Total episode reward with batch size of 64 and replay memory size of 5,000

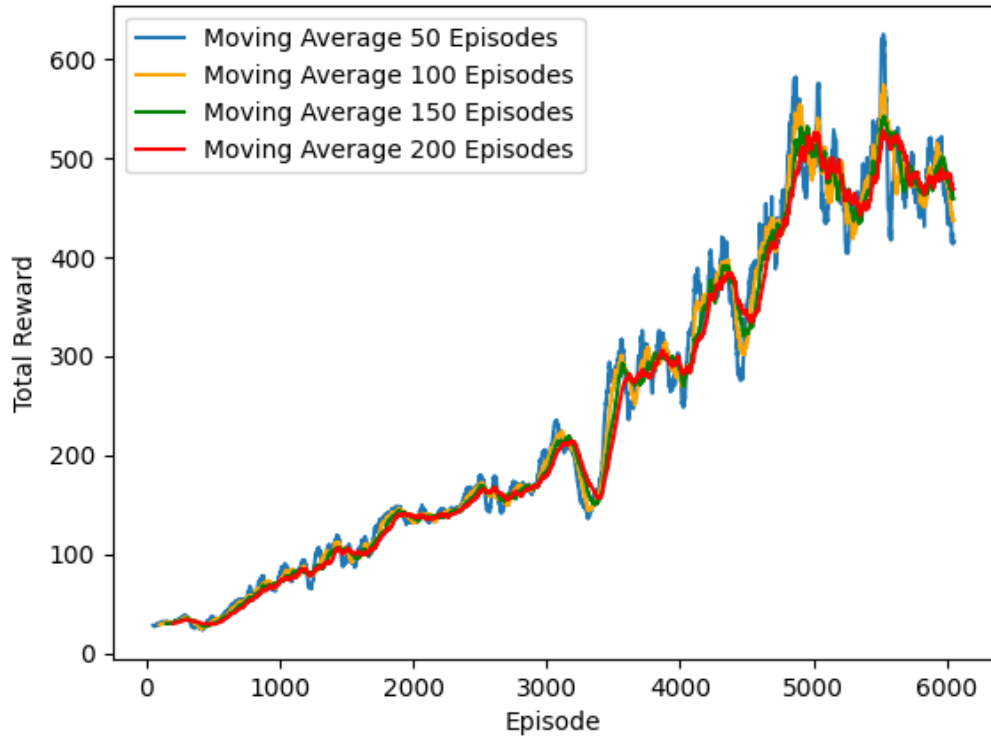


Fig. D.2 Total episode reward with replay memory size of 5,000

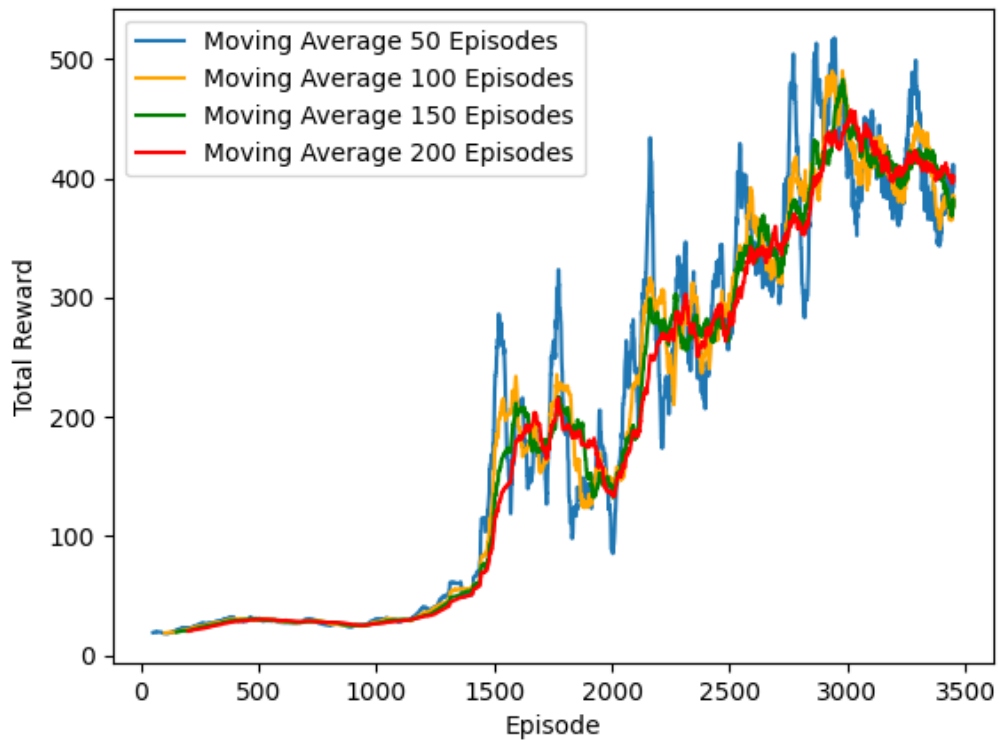


Fig. D.3 Total episode reward with batch size of 256, replay memory size of 5,000, and epsilon decay of 0.9975

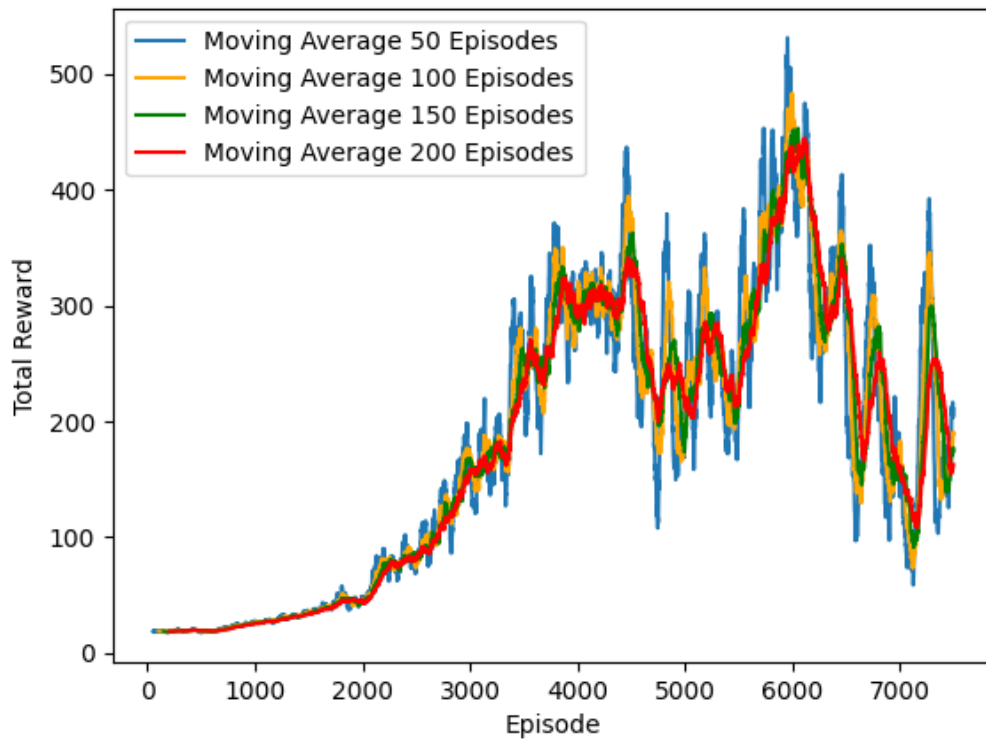


Fig. D.4 Total episode reward with batch size of 64 and replay memory size of 1,000,000

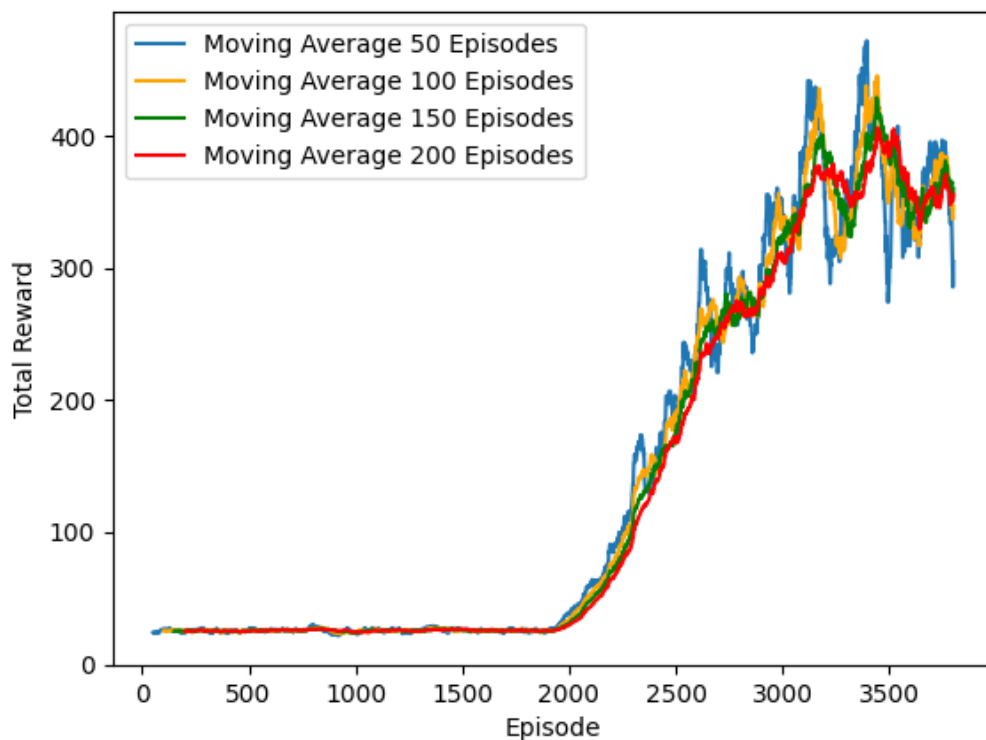


Fig. D.5 Total episode reward with batch size of 64, epsilon decay of 0.995, replay memory size of 1,000,000, and replay start learning size of 50,000

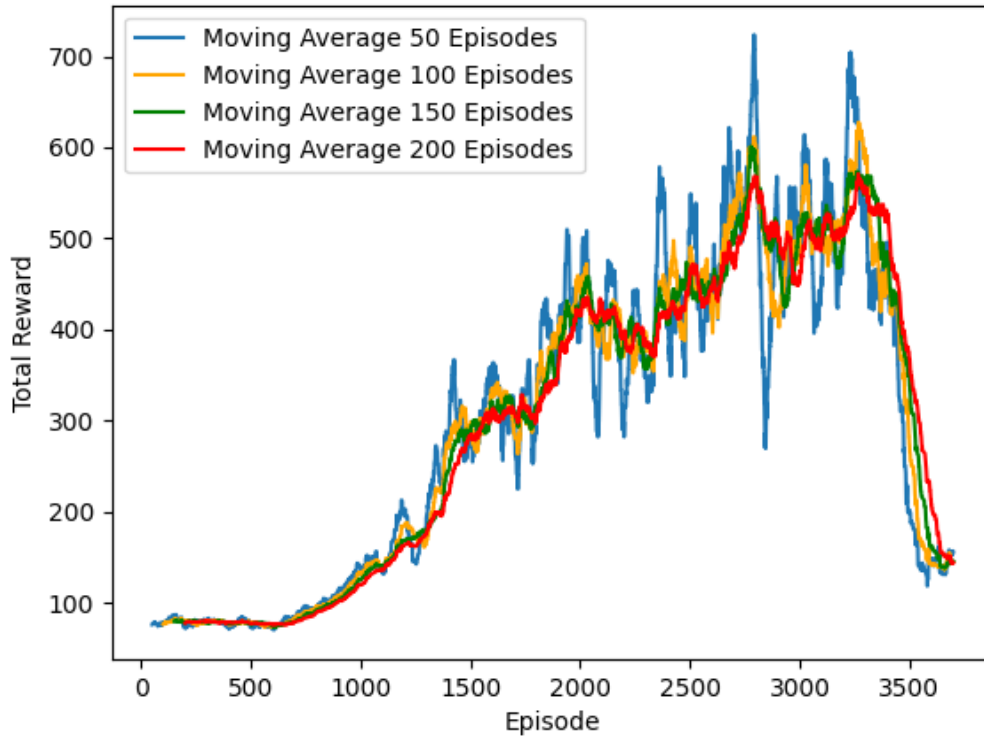


Fig. D.6 Total episode reward with batch size of 64, epsilon decay of 0.9975, replay memory size of 1,000,000, and replay start learning size of 50,000

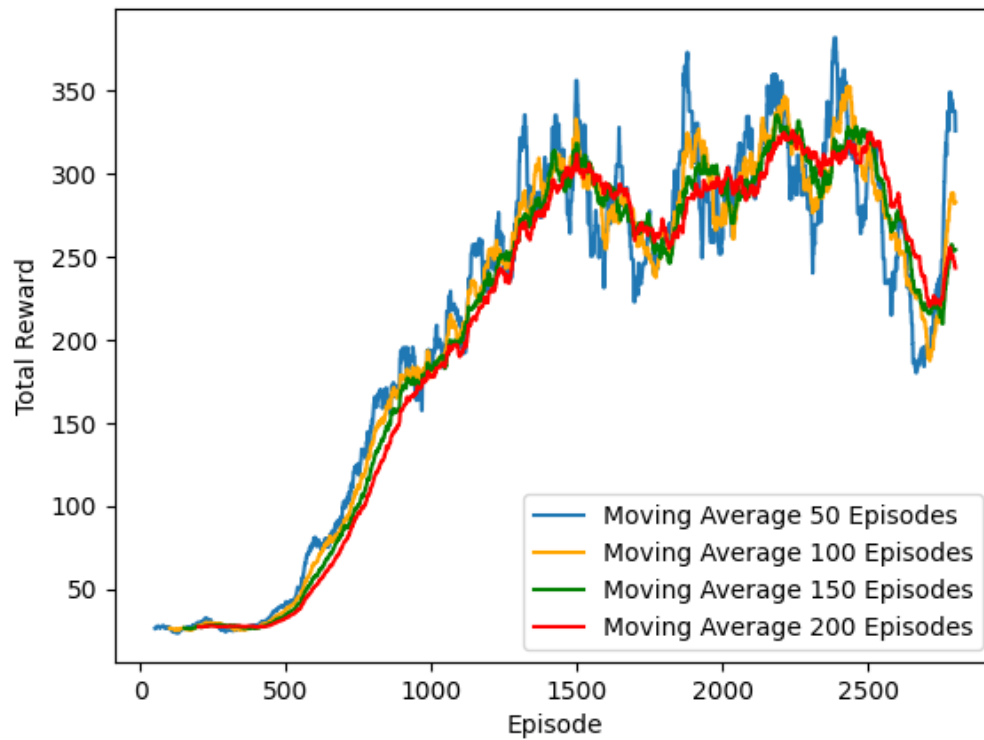


Fig. D.7 Total episode reward with learning rate of 0.0001, batch size of 32, epsilon decay of 0.995, and replay memory size of 1,000,000

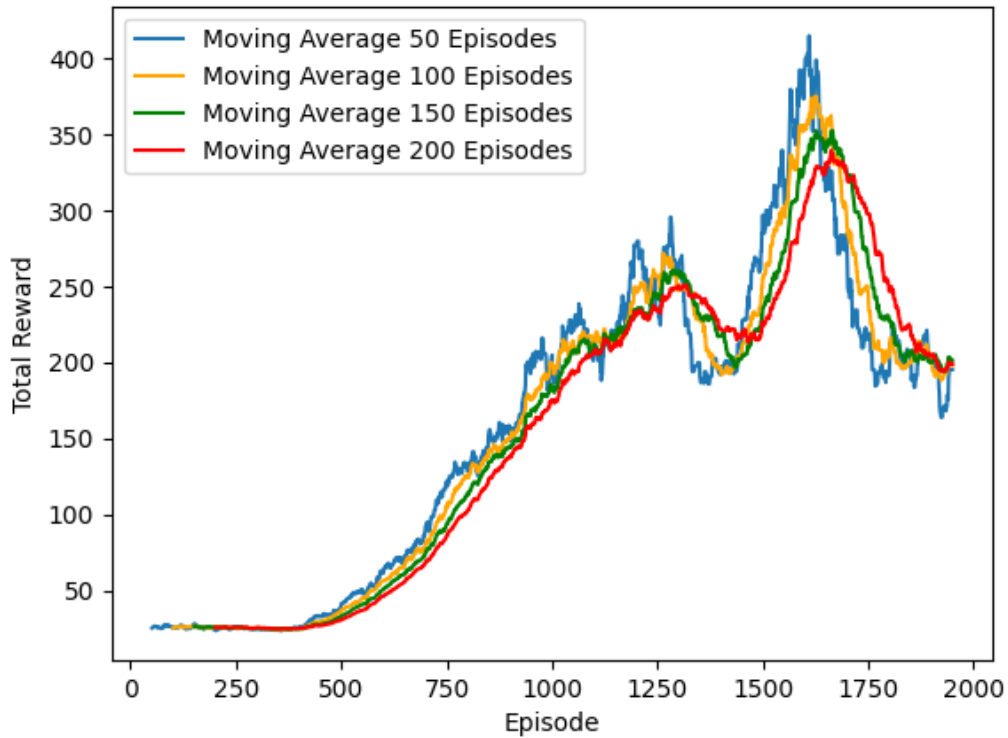


Fig. D.8 Total episode reward with learning rate of 0.0001, batch size of 32, gamma of 0.99, epsilon decay of 0.995, and replay memory size of 200,000

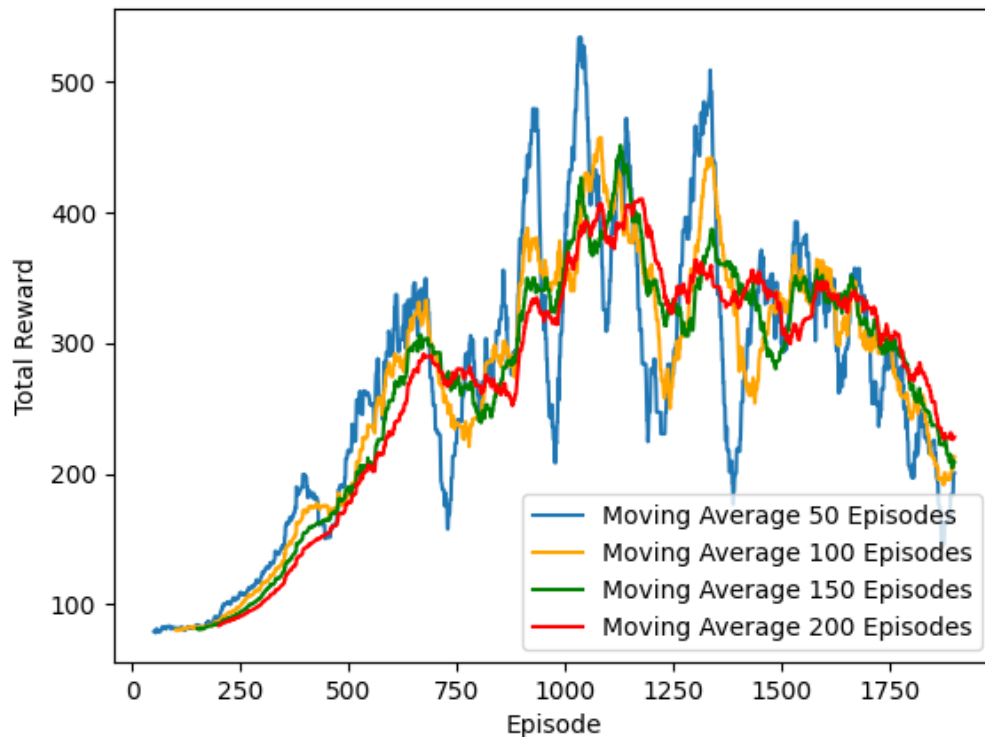


Fig. D.9 Total episode reward with batch size of 64, epsilon decay of 0.995, replay start learning size of 50,000, and replay memory size of 1,000,000