Strategies for Improving Resilience in Distributed Communication Systems

By

Alejandro Llorens-Carrodeguas

Ph.D. Advisor

Cristina Cervelló-Pastor

Department of Network Engineering
Universitat Politècnica de Catalunya

Thesis submitted to Universitat Politècnica de Catalunya in accordance with the requirements of the degree of DOCTOR OF PHILOSOPHY IN NETWORK ENGINEERING.

Barcelona, November 23, 2022

Abstract

With the development of industry and society, new verticals have emerged such as Industry 4.0, cooperative sensing, augmented reality, and massive Internet of things (IoT). These verticals require network robustness and availability to maintain high levels of quality of service (QoS) and quality of experience (QoE), just as 5G and beyond networks must guarantee to the end-users. In this regard, several technologies have been identified by network operators and academic institutions as crucial pillars in deploying these networks. More specifically, due to their advantages, software-defined networking (SDN), network function virtualization (NFV), and multi-access edge computing (MEC) are likely to make the deployment of 5G and beyond networks easier for operators.

SDN enables network programmability using either centralized or distributed controllers, which has caused a paradigm shift in communication networks since its introduction. Similarly, the use of NFV has grown because it provides new design techniques and orchestrates and manages network services by decoupling the network function's logic from proprietary hardware and running them as software applications. Lastly, MEC brings computing, storage, and network resources to the network edge near the end-users, which leverages the benefits of a reduced data path between the client and the service provider.

Nevertheless, network resilience and robustness are not guaranteed using these technologies. The literature addressing resilient and fault-tolerant strategies is focused on proposing unique mechanisms to be applied in each of the above paradigms. Furthermore, the proposed approaches are mainly related to communication mechanisms used in distributed domains to improve network scalability and resilience. Thus, they lack other perspectives and factors that can also impact the robustness and fault tolerance of the network, such as the virtual network function (VNF) allocation in resource-constrained environments.

This thesis therefore focuses on designing and implementing novel strategies to improve network resilience and fault tolerance in SDN/NFV environments through communication and event-allocation approaches. In particular, the proposed communication mechanism, different from existing related works, guarantees network information exchange among control elements of both SDN and NFV systems. This proposal enables auto-discovery and failure awareness, thus ensuring the necessary dynamism in the next-generation networks. Additionally, the control elements can improve their decision-making process by considering information external to the region controlled by them. Then other controllers' control and management policies, such as load-balancing methods, can be improved by considering this newly available information.

This thesis also introduces the VNF management and allocation process as factors that can impact network resilience when analyzing the VNF's characteristics and its deployments in resource-constrained environments. For the former case, we analyze the consequences of applying load-balancing and auto-scaling mechanisms over a cluster of transparent VNFs (i.e., those VNFs deployed in a bump-in-the-wire (BITW) manner). Due to the particularities of this kind

of virtual function, network loops may appear when applying the above mechanisms without considering their characteristics. Therefore, this study employs an SDN-based solution to address the problems of managing a cluster of these VNFs. The latter case explores the deployment of VNFs in resource-constrained environments like single-board computers (SBCs), which have gained attention as computing infrastructure close to the end-users. The specifications of this scenario (i.e., limited computational resources and battery-powered) entail challenges since existing platforms to orchestrate and manage VNFs do not consider energy levels during their placement decisions and, consequently, are not optimized for energy-constrained environments. Thus, an energy-aware scheduler is presented to deploy and manage VNFs in an SBC cluster. The proposed scheduler reduces the battery consumption of the cluster's nodes, thus improving the system's resilience.

To enhance the proposed scheduler's scope, an intelligent global controller is described to deploy events in a multi-cluster edge system. This approach integrates the implemented communication mechanism to gather the node's information to select the best participants when deploying virtual functions. This solution incorporates a machine-learning (ML) method to guarantee cost-effective resource utilization and increase the system's lifetime.

In general, the approaches proposed in this thesis have been evaluated in real testbeds using leading technologies. The obtained results have proved the applicability of the proposed communication mechanism to exchange network information among SDN/NFV environments' control elements. Additionally, the event-allocation strategy proposals have achieved outstanding performance, reducing battery consumption in energy-constrained systems while maintaining events' requests with high levels of acceptance ratio.

Keywords: DDS, DRL, energy-efficient scheduler, MEC, NFV, resilience, SBC, SDN, transparent VNF

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Dr. Cristina Cervelló Pastor, who gave me the opportunity to undertake this incredible adventure in the BAMPLA research group. I will forever be thankful for the confidence, guidance, knowledge, and support you have provided me throughout these years. I have no doubts that I could not have had a better person as my advisor.

I would like to extend my gratitude to Dr. Juan Felipe Botero and Dr. Akshay Jain for the time and efforts dedicated to assessing this study. Their reviews and comments were really appreciated and incredibly helpful in improving the quality of this thesis.

I am also very grateful to Dr. Dimitrios P. Pezaros for hosting me as a visiting researcher at the Network Systems Research Laboratory (NETLAB), University of Glasgow. Your support and research vision were extremely valuable. Special thanks to the fellow researchers I met during my time there. They have been a source of friendship and collaboration.

Special thanks to my officemates at the UPC through these years, with whom I have shared great moments. Thanks also to the members of the Network Engineering Department at UPC, especially to Prof. Sebastià Sallent Ribes for his keenness, wisdom, and willingness to help others facing challenges.

My most profound gratitude to my family for their endless love, support, and encouragement over the years. To my parents, grandparents, uncles, aunts, and the rest of my relatives, this is also your thesis because all of you have somehow contributed to the person I am today. I will be eternally grateful.

My deepest appreciation to my family-in-law also for their support, cheering, and reliability since the first moment I met them. A special mention to my beloved twins, who have always been there for me. From now on, you will not be the only doctors in the family.

Last, and definitely not least, I would like to thank my partner in life, Irita. There are no words to describe what your love, friendship, and faith in me represent in my life. I could not have imagined undertaking this journey without you.

"To my family, the beat of my heart"

TABLE OF CONTENTS

List of Tables

List of Figures

List of Abbreviations

AC Area Controller

AF Application Function

AMF Access and Mobility Management Function

AUSF Authentication Server Function

BITW Bump-in-the-wire

CAPEX Capital Expenditure

CUPS Control and User Plane Separation

DDS Data Distribution Service

DN Data Network

DOD Depth of Discharge

DRL Deep Reinforcement Learning

E2E End-to-End

EC Edge Computing

FC Fog Computing

GC Global Controller

IoT Internet of Things

IP Internet Protocol

LAN Local Access Network

MAC Media Access Control

MANO Management and Orchestration

MEC Multi-Access Edge Computing

ML Machine Learning

NAT Network Address Translation

NFV Network Function Virtualization

NFVI NFV Infrastructure

NFVO NFV Orchestrator

NS Network Service

NSSF Network Slice Selection Function

OF OpenFlow

OFS OpenFlow Switch

OMG Object Management Group

ONAP Open Network Automation Platform

OPEX Operational Expenditure

OSM Open Source MANO

PCF Policy Control Function

QoE Quality of Experience

QoS Quality of Service

RAN Radio Access Network

RL Reinforcement Learning

RMSE Root Mean Square Error

SBC Single-Board Computer

SDN Software-Defined Networking

SMF Session Management Function

SOC State of Charge

TCP Transmission Control Protocol

UDM Unified Data Management

UE User Equipment

UPF User Plane Function

VDU Virtual Deployment Unit

VIM Virtual Infrastructure Management

VLAN Virtual Local Access Network

VNF Virtual Network Function

VNFM Virtual Network Function Manager

WAN Wide Access Network

CHAPTER 1

INTRODUCTION

The fifth-generation (5G) ecosystems and beyond are envisioned to be revolutionary due to their unprecedented capabilities in terms of latency, reliability, data delivery rates, and the number of connected devices [1]. These characteristics will support a new era of services and applications and introduce opportunities in society's different sectors such as education, transport, healthcare, and industry. When analyzing expectations of 5G, three main features have been recognized: enhanced mobile broadband (eMBB), ultra-reliable low-latency communications (URLLC), and massive machine-type communication (mMTC) [2]. Several services and use cases have been categorized along with these features, such as augmented and virtual reality, autonomous vehicles, smart factories, and agriculture applications. They all have various requirements that 5G and beyond networks must support cost-effectively.

To achieve their promise, innovative and wide-reaching changes are needed to operate and manage the network and to provide services. In this regard, advanced technologies such as software-defined networking (SDN), network function virtualization (NFV), and multi-access edge computing (MEC) have been identified as fundamental pillars to capitalize the 5G and beyond networks' potential [3]. By leveraging these technologies, flexible, highly programmable, and autonomously manageable infrastructures can be built to satisfy emerging and future service demands (e.g., reliability, scalability, and energy consumption).

To guarantee these services' requirements, crucial network concepts such as resilience and fault tolerance must be considered in the enabling technologies (i.e., SDN and NFV) of the 5G and beyond networks. Resilience represents the network's ability to maintain an acceptable quality of service (QoS) when operational challenges appear [4]. Fault tolerance is the capacity to provide continued operation in the presence of faults that is often considered sufficient with redundant or distributed elements that ensure network survivability [5].

When analyzing both concepts from the SDN perspective, we notice they are closely related to two main controllers' research areas: communication mechanisms and control and management policies. The former is intrinsic to distributed SDN architectures where several controllers are deployed either flatly or hierarchically to increase the network scalability by avoiding centralized controllers' single point of failure. Therefore, a communication channel is necessary to exchange network details among the federated controllers. By replicating the controller's data, network robustness is improved because already available information can be used to diminish the network response when a controller fails. However, restringing communication only among SDN controllers represents a limited solution, especially when an integration is expected between SDN and NFV in 5G and beyond networks. Thus, proper communication mechanisms are required to enable federated information among the control elements of both enabling technologies.

The SDN control and management methods also play a key role in achieving resilience and fault tolerance. Due to its high programmability and flow monitoring features [5], SDN offers the required means to develop mechanisms capable of positively impacting the network capabilities mentioned above. One of the most-used approaches is load balancing, through which the SDN controllers decide the most suitable destination for a given traffic flow. In this manner, the associated problems of network link congestion, link disruption, and overloaded service hosts are avoided, which means better network health. However, since a joint performance among SDN, NFV, and MEC is expected in the next-generation networks, applying load-balancing techniques in virtual network functions (VNFs), deployed at the edge, poses several challenges due to sensitive traffic variations and limited resources of the MEC nodes [6]. For instance, a VNF with limited allocated resources degrades the QoS perceived by users [7]. In contrast, over-provisioned resources to VNFs increase the service-associated costs. Thereby, dynamic placement readjustment of VNFs along with the scaling of their assigned resources are required to optimize the use of MEC resources while providing high QoS values. Nevertheless, scaling out actions of determined VNFs may cause occasional network loops and subsequent problems (e.g., irregular connections and system failures) when applying load-balancing methods without considering the VNF's characteristics. An example of this scenario is the virtualization of traditional network functions like traffic accelerators, which can be deployed in a bump-in-the-wire (BITW) manner to avoid altering communication endpoints. Thus, revised load-balancing approaches [7–10] lack consideration of the VNF's behavior when performing its decisions, which can lead to undesired network performance that could affect its resilience.

As mentioned, the NFV and MEC convergence is crucial to accommodate the latency requirements of the next-generation networks since the processing for the requested services can be placed near the end-users [11]. Additionally, the used edge nodes can interoperate in commodity clusters to enable failure recovery and accumulate processing capacity, thus satisfying requirements such as availability and resilience. A device required by edge nodes is the single-board computer (SBC), which has become a mainstream option for Internet of things

(IoT) environments [12], although its utilization has been extended to other sectors. In the last several years, the hardware capabilities of these nodes have been significantly improved, with a 1.5 GHz quad-core of CPU in the last generation versus a 700 MHz single core of CPU in the first launched model [13, 14]. Such improvement has magnified their utilization as standalone devices or in a cluster. The variety of use-case scenarios (e.g., low-latency cyber-physical systems, resource-constrained computing, and next-generation data centers [15]) is another example of the popularity of SBCs. One of the essential characteristics of an SBC is its small size, which enables a higher density of devices and lowers cost when covering huge areas [13], specifically, on-boarding autonomous vehicles to provide services during natural disasters. However, the constraints in computational resources must always be considered when deploying services on these nodes.

Currently, an increasingly common strategy for efficiently managing deployed clusters is coupling the network paradigms mentioned above with containerization technologies [16]. This kind of deployment allows lightweight virtualization by enclosing only required code dependencies for the service execution, which introduces versatility in the used nodes while accelerating and simplifying the service instantiation [17]. Nevertheless, deploying and managing constituent VNFs of services in a commodity cluster can be a laborious and error-prone task. Therefore, a platform capable of orchestrating and controlling the lifecycle of containers and applications is necessary. In this regard, Kubernetes [18] has become the most prominent framework to perform these functions. However, in this platform and others, existing schedulers were initially designed for data center environments, thus basing their allocation processes on key performance resources such as CPU and memory. Therefore, they do not integrate energy measurements of the participant nodes in the placement decision, which represents an important metric when deploying services in battery-powered SBCs. This scenario can result in under-utilizing the available energy resources or attempting deployments destined to fail due to insufficient resources. Both situations negatively impact the system's resilience, demonstrating the importance of the VNF allocation process when considering energy- and resource-constrained environments.

For the above reasons, the focus of this thesis is to propose and implement mechanisms to increase network resilience and fault tolerance by considering scenarios neglected in the revised literature. The proposed approaches are evaluated in real testbeds using the identified enabling technologies of 5G and beyond networks.

1.1 Research Problem and Objectives

Considering the detailed context of the preceding section, this research focuses on the design, implementation, and evaluation of novel strategies related to network resilience, fault tolerance, and VNF allocation in SDN/NFV environments. For such purposes, this investigation is guided and motivated by solving the following research questions:

- Is it possible to achieve network resilience and fault tolerance in SDN/NFV environments through communication and event allocation mechanisms?
- How can a method be developed to enable communication among control elements in SDN/NFV systems while allowing faster response and failure awareness?
- How can dynamic energy-efficient VNF allocation be provided in energy-constrained devices while guaranteeing a cost-effective resource utilization and increasing the system's lifetime?

A set of primary and secondary objectives is defined to address the identified problems. The principal goals of this thesis are summarized below:

1. To design, implement, and evaluate a mechanism capable of distributing control elements in SDN/NFV environments while guaranteeing network resilience, exchange of network information, and device failure awareness
2. To design, develop, and assess scheduling strategies to allocate service requests, formed by several VNFs, in commodity clusters of energy-constrained devices while increasing the lifetime of the system and therefore its resilience and fault-tolerance capabilities

Additionally, a set of secondary objectives is specified to provide support in accomplishing the main research goals:

1. Perform a thorough and systematic state-of-the-art review on topics closely related to SDN, NFV, and other enabling technologies
2. Identify existing solutions linked to network resilience and VNF allocation mechanisms to determine their scope, contributions, and drawbacks
3. Develop a modular communication mechanism for logically distributed SDN controllers in a flat or hierarchical architecture while guaranteeing fault tolerance for 5G and beyond networks' control plane
4. Implement control and management policies, such as load balancing and auto-scaling, in an SDN-based solution to improve resilience and fault tolerance at the service and infrastructure levels
5. Develop an allocation mechanism to deploy constituent VNFs of service requests in a cluster of energy-constrained devices by considering battery levels and computational resource utilization
6. Extend the scope of the proposed allocation strategy by integrating the implemented communication mechanism and machine-learning (ML) techniques to deploy VNFs in multi-cluster scenarios

7. Assess the effectiveness and performance of the proposed solutions by conducting evaluations in real testbeds

1.2 Resources

To conduct this research, the Department of Network Engineering (ENTEL) provided the required resources and support as well as outstanding guidance and expertise. Additionally, access to training courses, participation in international conferences, stays abroad, and other relevant scientific opportunities were granted to increase the reach and scope of this study while obtaining adequate feedback.

From the materials and tools applied in this study, Java and Python were the programming languages used to implement the proposed algorithms and testbeds where they were evaluated. R programming language was utilized to graph and process the results. The RTI Connext DDS [19] was employed to build the communication mechanism used by SDN controllers and other entities of the proposed solutions.

1.3 Contributions

The contributions of this thesis, in agreement with the proposed research problems and objectives and in consideration of the articles published in journals and international conferences (see Appendix A), can be summarized as follow:

1. To contribute a new mechanism for the eastbound/westbound interface in the SDN architecture, a **data distribution service (DDS) approach to logically distributed SDN controllers** (i.e., in a flat or hierarchical manner) is proposed. This mechanism can guarantee a robust control plane for 5G and beyond networks since the control elements can exchange information, allowing faster responses to network and controller failures. Additionally, the proposed solution enables the auto-discovery of the network elements. Furthermore, an authentication procedure is included in its design to prevent unauthorized publishers from introducing incorrect messages.

2. An **SDN-based solution to manage a cluster of transparent VNFs** (i.e., those deployed in a BITW manner) is introduced. The proposal allows dynamic load-balancing and auto-scaling decisions while ensuring bidirectional flow affinity without packet modification. To the best of our knowledge, we are the first to address the associated problems of applying these policies to transparent VNFs.

3. A novel **scheduling process to allocate service requests in an energy-constrained SBC cluster** is proposed. This scheduler includes a regression model to establish the relationship between battery consumption and used resources in the cluster. This model

enables intelligent decisions by the scheduler since the requested events can be assigned based on the expected battery levels and used resources, thus improving the cluster's resilience and optimizing the management of the available resources. In addition, the proposed allocation mechanism makes better use of the cluster's available resources by considering the computing and controller nodes in the placement process.

4. An **intelligent controller as a global allocation mechanism** is presented to deploy VNFs across multiple domains of SBC clusters. The proposal introduces a deep reinforcement learning (DRL) algorithm that considers resource utilization, energy consumption, and service requirements to select the most suitable nodes. Furthermore, the intelligent controller uses **DDS to communicate with the controller node of each cluster** to exchange nodes' status, service requests, and allocation actions.

5. The methods proposed throughout this thesis, specifically the DDS mechanism, SDN-based solution for transparent VNF cluster, energy-friendly scheduler, and intelligent controller, have been evaluated in **real testbeds that use leading technologies**. Additionally, the source code of **the implemented solutions** has been **published in a GitHub repository**, which can be accessed by request.

1.4 Thesis Outline

The outline of this thesis is depicted in Fig. 1.1, where each chapter is represented in terms of its contributions.

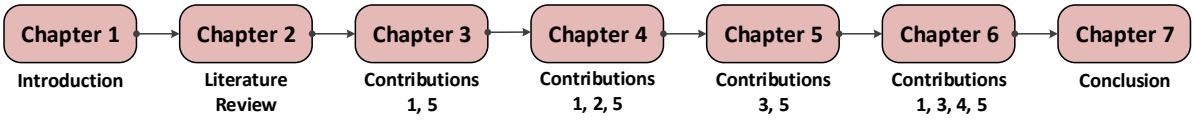


FIGURE 1.1. Thesis outline.

Chapter 2 provides an overview of the leading network technologies used throughout this thesis. In particular, it discusses the theoretical background of SDN and NFV concepts needed to accomplish the research objectives formulated in Chapter 1. Chapter 2 also presents the literature review by focusing on relevant research related to network resilience and fault tolerance in SDN/NFV environments. This analysis provides an updated perspective of state-of-the-art works by identifying the research challenges stated in this thesis.

Chapter 3 describes a fault-tolerant architecture of SDN controllers that enables robustness in 5G and beyond networks. The primary feature in the proposed deployment is a new communication mechanism based on DDS to federate the control elements. A detailed description of these elements and how they interact with different controller blocks is given. Furthermore, a national

testbed is developed for evaluation purposes, and several experiments are conducted to analyze different network performance metrics.

Chapter 4 proposes an SDN-based solution to improve the availability and resilience of a deployed network service. Different modules comprise the solution and provide load-balancing and auto-scaling capabilities to manage a service's constituent VNFs. These features, along with the utilization of the DDS application presented in Chapter 3, enable resilience at the service and infrastructure levels. The proposed solution is validated by examining its performance in various situations.

Chapter 5 addresses the network resilience from the NFV and MEC perspectives. Particularly, a VNF allocation mechanism is presented to improve the resource utilization of energy-constrained edge nodes. In this way, their resilience is increased by integrating the participant devices' energy measurements into the placement decisions. These data are later used to train a regression model that establishes a relationship between the resources used and energy consumption. By predicting the energy consumption through this model, network-wide placement decisions are enabled, thus lengthening the total lifetime of the network and optimizing the use of available resources. The proposed allocation process is evaluated in a testbed formed by a cluster of SBCs devices.

Chapter 6 extends the results obtained in Chapter 5 by considering multiple domains of commodity clusters of SBCs. To this aim, a global allocation strategy is proposed to select the most suitable nodes among the participant clusters to deploy a service request. The introduced mechanism is based on DRL and considers resource utilization (e.g., CPU, memory, and energy levels) and service requirements to select the best actions. This chapter also extends the utilization of the DDS application presented in Chapter 3 since it is used to communicate the global entity with the clusters' controller nodes. Therefore, nodes' status, service requests, and placement decisions are notified to the corresponding element.

Finally, Chapter 7 summarizes the preliminary conclusions derived from this thesis and suggests key directions for future work.

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

To contribute to the fault tolerance of 5G and beyond networks and the VNFs allocation in these systems, we performed an initial analysis of related concepts and known enablers, as well as state-of-art literature. The following sections present a comprehensive study of SDN and its current implementations. Additionally, this chapter discusses the latest findings and limitations regarding the controller communication strategies and deployments since these aspects are related to the resilience of this system. Regarding NFV deployments, we review different strategies to improve their resilience by properly scheduling and processing the constituent VNFs of requested services.

2.1 Software-Defined Networking

SDN has been recognized as a crucial pillar in the development of 5G and beyond networks to fulfill their network requirements [3, 20, 21]. The SDN architecture is formed by three layers: application, control, and data [22]. More specifically, the control plane manages all the devices (e.g., routers and switches) in the data plane in a unified manner through the controller device. Meanwhile, the network applications are implemented and executed in the application layer and are used by the controller. The control and user plane separation (CUPS) is one of SDN's main features. It guarantees that the resources of each plane can be scaled independently. Additionally, the CUPS allows the deployment of user plane functions close to users, therefore reducing network response times and bandwidth consumption. In addition, the separation of control and user planes enables the programmability of network policies, thus ensuring versatility in network configurations [22]. Moreover, SDN offers new methods to flexibly instantiate network functions and services while reducing expenses and boosting performance. For instance, it provides the

ability to adapt network resources and topology to changes in the configuration and placement of network functions and services [23, 24].

Depending on the particular use case, the SDN concept can be applied in various solutions where the controller is a crucial element in the control operations. In this regard, the architectural deployment of the controllers is important to guarantee that the overall network's performance is adequate. Additionally, the resilience and fault tolerance of the control plane depend on how the controllers are deployed. However, not only is the architectural deployment of the controllers significant, but mechanisms and policies to efficiently manage their assigned nodes are also required.

2.1.1 Controller Deployment

From an architectural point of view, SDN controllers can be classified into two categories: centralized and distributed. A unique controller forms the centralized architecture for the sake of simplicity. Centralized controllers (see Fig. 2.1), such as NOX-MT [25], Maestro [26], Beacon [27], Ryu [28], and Floodlight [29], have been designed to achieve a determined throughput for purposes of enterprises and data centers. However, this design represents a network bottleneck and has scalability limitations when the network traffic increases.

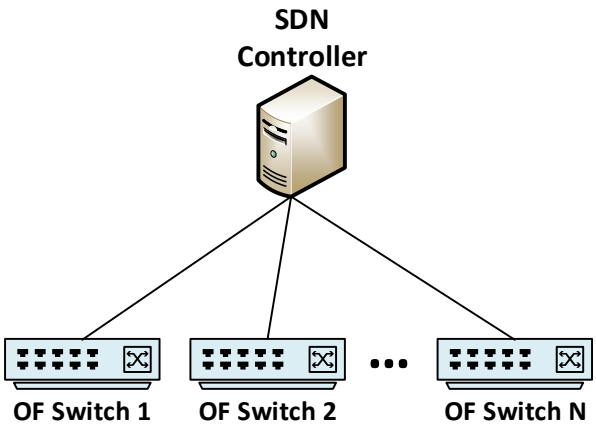


FIGURE 2.1. Centralized controller deployment.

In contrast, Fig. 2.2 depicts a distributed architecture that improves network scalability, flexibility, and reliability. For instance, this deployment avoids having a single point of failure since other distributed controllers can assume the functions and devices of a failed controller. This type of design is used in a wide range of examples such as Onix [30], HyperFlow [31], and ONOS [32]. The deployment used in the previous cases is more resilient to different kinds of disruptions. In addition, the distributed deployment of controllers can respond and adapt to new requirements and conditions due to the automation that offers the programmability of SDN. It should be noted that despite distributed controllers improving the network's resilience and

scalability, this kind of architecture requires reliable and robust communication mechanisms to maintain data consistency (orange line). Moreover, this sort of architecture imposes a consistent network-wide view on the controllers and generates significant control traffic, which can affect the latency and throughput.

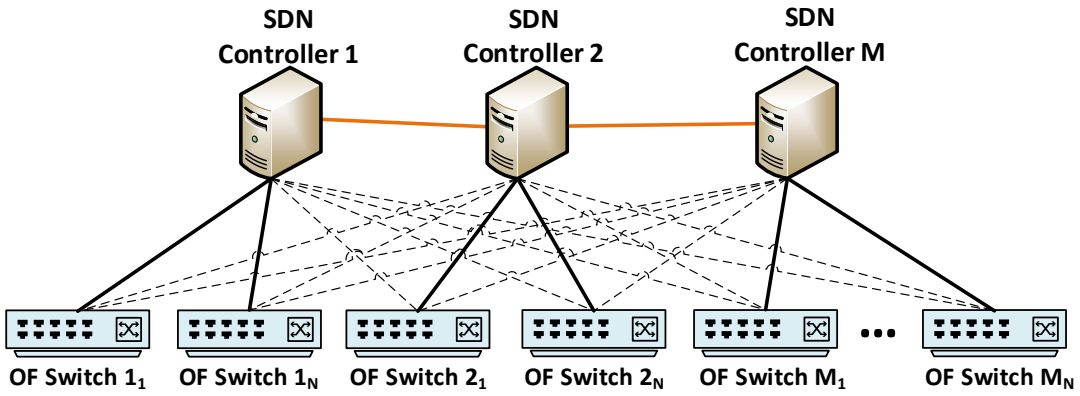


FIGURE 2.2. Distributed controller deployment.

To overcome these limitations, the authors of [33–35] implement a hierarchical architecture, as shown in Fig. 2.3, which represents a combination of the previous deployments. It guarantees scalability since specific functions can be implemented in the controllers to specify their role in the network. Thus, only traffic between different SDN domains is forwarded to the controller in the upper level. Each controller manages its domain in a hierarchical architecture and distributes specific data to other controllers. Similar to distributed deployment, communication among controllers requires reliable mechanisms to guarantee proper performance in the hierarchy.

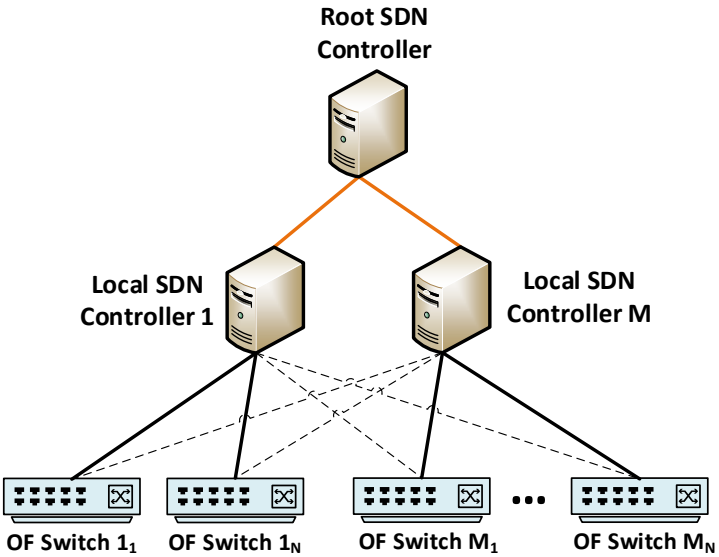


FIGURE 2.3. Hierarchical controller deployment.

2.1.2 Controller Communication Mechanisms

Distributed systems are characterized by their fault-tolerant capability. Therefore, distributed and hierarchical controllers improve the control plane resilience and scalability while reducing the problems caused by the network partition. The controllers must be well synchronized to execute their functions, thus allowing a correct performance in such deployments. In this regard, the eastbound/westbound interfaces in SDN controllers are responsible for their communication. As seen in the literature review, several works have implemented these interfaces to exchange network information among distributed controllers.

Koponen et al. [30] have proposed a distributed system formed by a commodity cluster of several physical servers that run multiple Onix instances and use Apache ZooKeeper [36] to store and synchronize the network information base data. This information represents a graph of all entities in the network topology. Consequently, this proposal provides scalability and resilience by replicating and distributing data among multiple running instances.

In [31], the authors present HyperFlow as a distributed event-based control plane for OpenFlow (OF). Its solution allows the deployment of any number of network controllers while maintaining a logically centralized network control. This proposal relies on the publish/subscribe paradigm to communicate controllers through the distributed file system WheelFS [37], which offers flexible wide-area storage for distributed applications. Therefore, the controllers can build a global network graph while managing the network devices.

Fu et al. [33] have presented a two-level architecture of SDN controllers. Physical switches and routes are managed by a set of controllers at the bottom level of the architecture. Simultaneously, its upper layer comprises domain controllers that control the area controllers as devices and synchronize the global network view through a distributed database.

Additionally, the authors of [34, 35] have implemented an extensible distributed SDN control plane (DISCO) on top of Floodlight controllers to establish communication among them. Each controller manages its network domain and synchronizes with other elements to provide end-to-end network services. DISCO performs its functions through two key elements: messenger and agents. The former discovers neighboring controllers while maintaining a distributed publish/subscribe communication channel. The latter uses this channel to exchange network information among participants.

Benamrane et al. [38] have proposed a communication interface for distributed control (CIDC) plane that allows the synchronized exchange of notifications and services among several distributed controllers. Their proposal is composed of four modules: consumer, producer, data updater, and data collector. Each controller in the study performs as a consumer for external events and a producer for local ones. The authors' solution improves the results in terms of latency, overhead, and system consumption with regard to the OpenDaylight distribution working on a commodity cluster.

In [39], Lin et al. describe west-east bridge (WE-Bridge) as a mechanism to enable peering

and cooperation between different SDN domains. Its contribution improves interdomain routing by announcing fine-granularity information of domain views. The authors test its design through an international SDN testbed using CERNET, CSTNET, Internet2, and SURFnet networks.

In addition to the previous contributions, there are several native solutions in existing controller distributions to communicate distributed control planes. In this regard, ONOS [32] distribution performs similar to Onix. More specifically, this SDN controller runs on multiple servers, each of which is the master controller for several switches. In this sense, additional instances can be added to the ONOS cluster when the number of nodes increases to distribute the control plane workload. Toward this aim, this approach's network view data model is implemented through the Titan [40] graph database and Cassandra [41] key-value store for the distribution and persistence of network information. Another SDN controller with a native mechanism to distribute control planes is OpenDaylight [42]. In particular, its architecture integrates the OpenDaylight SDN interface application (ODL-SNDi App) [43, 44] to federate controllers. The most important element in this application is the SDNi wrapper, which is responsible for sharing and collecting data among synchronized instances. However, it represents the main limitation of this approach since the wrapper is based on the border gateway protocol (BGP). Thus, the communication is not in real time, which can affect the recovery time when a failure occurs.

Most of the works above use the publish/subscribe paradigm to guarantee the communication channel among SDN controllers, which depicts its widespread use to distribute control planes. To the best of our knowledge, there is no standard for the eastbound/westbound interface in the SDN architecture. In this sense, and in contrast to the previous solutions, this research explores a new mechanism to exchange network information among controllers based on DDS.

2.1.2.1 Data Distribution Service: Main Concepts

DDS is a middleware protocol for datacentric connectivity from the Object Management Group (OMG). This standard addresses the publish/subscribe communication for real-time and embedded systems [45].

It provides a global data store in which publishers and subscribers write and read data, respectively. Additionally, this mechanism offers a fast and predictable distribution of time-critical data over a variety of transport networks. Moreover, DDS provides a flexible data distribution infrastructure by integrating several types of data sources. Thus, it can deliver a large amount of data with microsecond performance and granular QoS control.

This middleware is mainly formed by the following entities: domain, domain participants, topics, publishers, data writers, subscribers, and data readers. The relationship among these elements is depicted in Fig. 2.4.

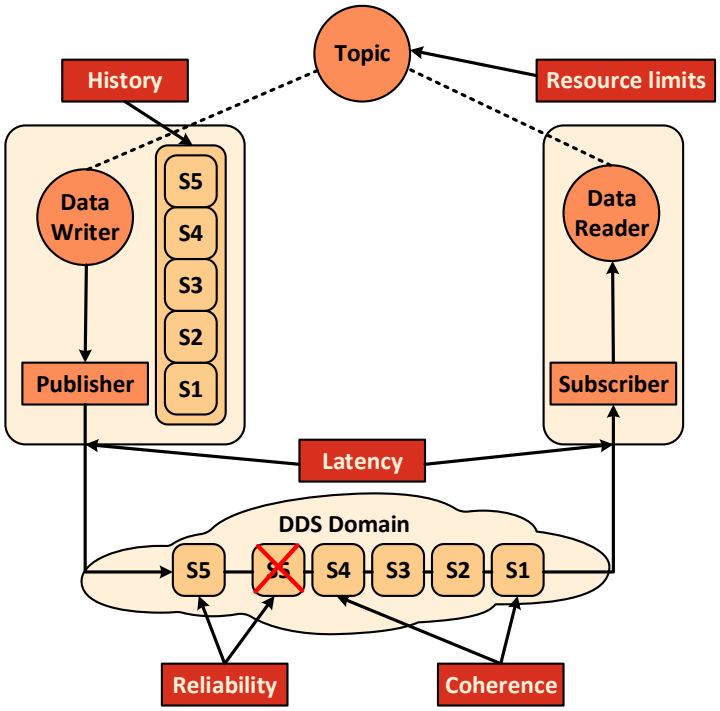


FIGURE 2.4. Overview of the DDS's entities [46].

The DDS domain represents the global data store containing the information provided by the applications registered to that domain. These applications use a DDS topic to exchange information. It describes the type and structure of the data while allowing strongly categorized data dissemination. The data writers and data readers can subscribe and publish specific topics, respectively. In this regard, several data writers and data readers can be managed by a publisher and a subscriber, respectively. These two entities discover each other automatically and match when having compatible topics and QoS parameters.

The topics are exchanged among peers within the DDS domain according to an established contract during the discovery phase, which is performed dynamically. Thus, the participant entities do not require any configuration of the endpoints in advance since they are automatically discovered by the DDS. In this stage, each domain participant maintains a local database with the active data writers and data readers in the domain.

2.1.2.2 Data Distribution Service in SDN Controllers

The DDS standard offers several advantages to users such as easy integration, efficient performance, scalability, advanced security, enabled QoS, and automatic discovery. These characteristics have been exploited in some SDN-based solutions.

In [46], Bertaux et al. propose a communication system that combines DDS and SDN to efficiently support dynamic real-time application distribution on non-dedicated network infras-

tructures. This proposal is based on the advantages of DDS and SDN. The former defines a set of mechanisms that implement QoS-aware control and access to host resources. The latter manages the resources of exposed interfaces according to application requirements.

The authors of [47] describe an IoT network architecture that integrates SDN and DDS. This architecture comprises a powerful and straightforward abstract layer that is independent of the specific networking protocols and technologies. This layer plays a mediator role between the IoT system and the network. It uses the DDS data reader listener to receive PACKET_IN events captured by the controller. Similarly, the abstraction layer encapsulates new packets in DDS topics and sends them to the packet forwarder service by using the DDS data writer. The utilization of DDS in the IoT architecture offers the required scalability, reliability, flexibility, security, and real-time data delivery.

Hakiri et al., in [48], have proposed DDSFlex as an extensible southbound protocol. Its solution exchanges abstract policies between the network controller and several SDN switches. This new protocol supports a proactive model through overlay tunnels to slice the network. Therefore, DDSFlex combines fine-grained flows in virtual devices and coarse-grained flows in the physical underlay network.

Similarly, the authors of [49] have introduced proactive broker-less subscriber interest-defined overlay networking (POSEIDON) as a broker-less, extensible architecture for proactive overlay SDN deployments. POSEIDON represents a publish/subscribe middleware that allows the configuration and management of the data plane by the controller. Additionally, it enables the status description of the network elements.

The contributions of these papers are focused on vertical communication. However, this investigation extends the use of the DDS to exchange network information among SDN controllers, giving it a different use (i.e., horizontal communication) in comparison with existing works. In this way, it ensures a robust control plane for 5G and beyond networks since the control elements share data, allowing a faster response to network and controller failures.

2.1.3 Controller Control and Management Policies

The decoupled implementation of the network control logic in SDN controllers enables high levels of programmability in network applications and fine-grained flow monitoring [5]. These characteristics can support the development of mechanisms that provide resilience at the different levels of the SDN architecture (i.e., application, control, and data plane). Thus, the network's resilience and fault tolerance capabilities can be improved by distributing control elements and applying proper mechanisms.

In this regard, resilience areas such as performability, disruption, and traffic tolerance are assured with multi-path and load-balancing solutions [5]. Most load-balancing approaches in SDN controllers focus on designing new methods to decide the most suitable destination for a given traffic flow. In this manner, the authors of [50, 51] have proposed SDN load-balancing

algorithms to dynamically distribute the request to servers. These proposals select the most appropriate server based on real-time metrics such as CPU and memory utilization. Liu et al. [50] have selected the least loaded server to handle traffic requests, whereas the authors of [51] used a dynamic weighted random selection (DWRS) algorithm, which assigns higher weights to underutilized services. Thus, the least loaded server has a higher probability of being chosen. Moreover, Chen et al. [52] have proposed an SDN-based load-balancing solution to distribute traffic among a server cluster. Concretely, the clients' requests are redirected to a virtual IP address before reaching the selected server. Similar to traditional load balancers, this solution uses a virtual IP and packet modification to route the traffic to the servers.

In [53], the authors introduce two OpenFlow-based mechanisms to balance multi-path TCP (MPTCP) traffic to a pool of hosts. Both solutions are at the expense of packet modification, even though they guarantee the same server is selected for subflows attached to the same MPTCP session. Likewise, Ma et al. [8] relied on OpenFlow switch (OFS) to dynamically reroute traffic inside the network function chaining. The chaining management utilizes a unique pair of virtual local access network (VLAN) tags for each chain. In addition, the proposed solution captures incoming packets and adds the VLAN tags through the open-source PF_RING network socket [54]. Ma et al.'s approach allows the controller to update the number of VLAN tags used for traffic balancing at any time.

The authors of [55] have promoted a load-balancing method to distribute traffic flows among switches by considering the equipment characteristics. In the solution, an SDN controller directs a data plane scalability mechanism to monitor traffic and used resources on the forwarding devices. This procedure adjusts the number of active switches to meet network requirements. Additionally, the load balancer installs the required flow rules on target devices and migrates flow rules to improve resource usage.

Abdelltif et al. [56] have developed a load-balancing service for SDN controllers to optimize resource utilization while reducing the user response time. The proposed mechanism is formed by three application modules (i.e., service classification, dynamic load balancing, and monitoring) running on top of an SDN controller. The dynamic load balancer distributes the incoming traffic to the servers by considering the service type. The authors prove the feasibility of the proposal through experimental findings.

Other authors have integrated ML techniques in their SDN-based solutions to perform optimal decisions during network management. These approaches address routing optimization, QoS/QoE prediction, and resource management to improve the system's resilience and fault tolerance.

The authors of [57] have proposed NeuRoute as a dynamic routing framework for SDN controllers. The solution predicts the traffic matrix in real time using neural networks to learn the traffic characteristics. Thus, the controllers can generate forwarding rules to optimize the network throughput. Additionally, the authors of [58] have designed a long short-term memory

(LSTM) framework to predict the traffic matrix in large-scale networks. The simulation results depict a fast convergence of their proposal with accurate performance. These approaches allow the controllers to take action in advance and proactively configure routing policies to avoid network congestion.

Similarly, Sendra et al. [59] have explored routing optimization techniques by proposing an intelligent protocol, which uses the reinforcement learning (RL) method to choose the best data transmission paths according to the best criteria and network status. The authors of [60] have taken this a step forward, applying DRL to optimize routing by using an agent that adapts automatically to the current traffic conditions. Thus, its solution can minimize network delay through tailored configurations.

Lin et al. [61] have introduced an RL-based QoS-aware adaptive routing (QAR) technique. This approach enables time-efficient adaptive packet forwarding in a multi-layer hierarchical SDN system. The routing path with the maximum QoS-aware reward function is selected based on the traffic conditions and users' applications. The obtained results outperformed conventional learning approaches in terms of QoS provisioning with quick convergence.

Finally, this research aims to extend the controller control and management policies by allocating virtualized functions and applying intelligent decisions in resource-constrained (e.g., CPU, memory, battery levels) scenarios. In this regard, utilizing ML algorithms can be crucial to obtaining the best results. It should be noted that most of the approaches mentioned above have been tested in simulation environments. However, the current research implements real testbeds.

2.2 Network Function Virtualization

In line with SDN, NFV plays a crucial role in the deployment of 5G and beyond networks [20, 21]. This technology decouples the network function's logic from proprietary hardware and runs them as software applications on general-purpose hardware [3, 62]. NFV generates new ways to design, orchestrate, deploy, and manage network service (NS), improving business agility while reducing capital expenditure (CAPEX) and operational expenditure (OPEX). Furthermore, the agility and flexibility of this paradigm to manage network resources and services support carriers in deploying various verticals with different requirements while reducing costs.

The standardization work of the NFV development has been led by the European Telecommunications Standards Institute (ETSI). The standard documents propose an architectural framework [63] composed of three main blocks: NFV infrastructure (NFVI), VNFs, and NFV management and orchestration (MANO) [64]. The NFVI provides abstraction between the hardware and virtual resources through a virtualization layer. The VNFs represent the deployed network functions using the virtual resources of the NFVI. The NFV MANO provides a standard architecture used as a reference by vendors and open-source MANO projects for the monitoring and provisioning of VNFs [65]. This architecture is formed by the virtual infrastructure management

(VIM) (e.g., OpenStack [66] and Amazon Web Service [67]), the virtual network function manager (VNFM), and the NFV orchestrator (NFVO). The last two elements (i.e., VNFM and NFVO) are under the umbrella of the group of open-source MANO, which includes several projects such as open network automation platform (ONAP) [68], open source MANO (OSM) [69], Open Baton [70], and Cloudify [71]. ONAP and OSM are the most prominent of these projects in both academic and industry sectors since big operators such as AT&T and Telefonica support their development [72]. Overall, the above-mentioned projects are intended to create and coordinate the required resources of the network services and manage the lifecycle of its constituent VNFs. Figure 2.5 depicts the constituent elements of the NFV architecture and how the VNFs and the NFVI are related to the different blocks of the MANO.

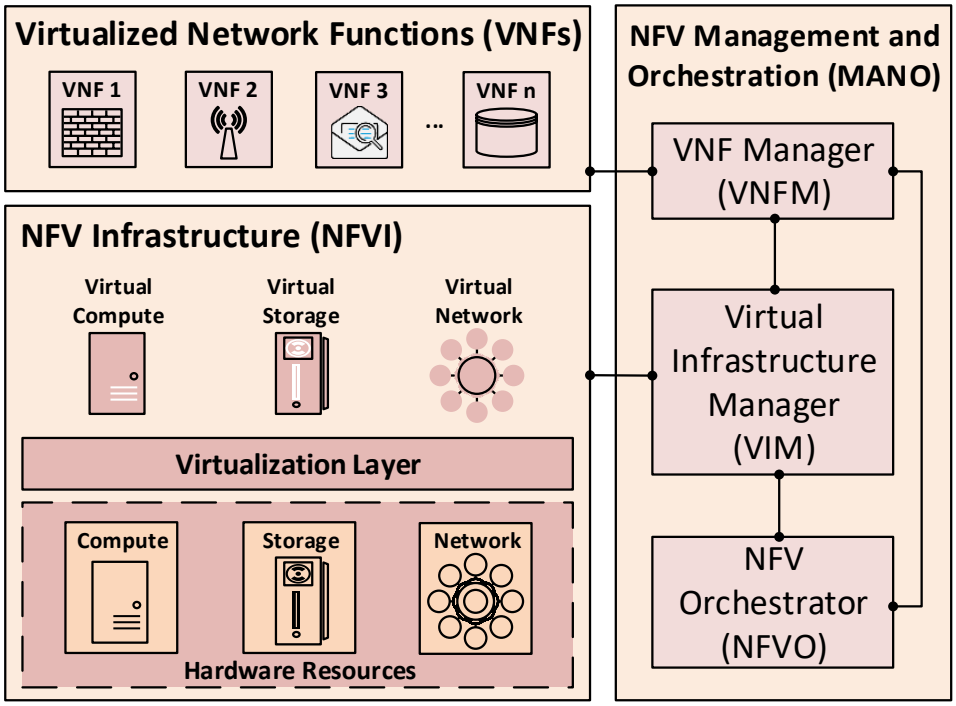


FIGURE 2.5. Overview of the NFV framework.

From the point of view of this current study, one of the most crucial characteristics of NFV is its versatility to be combined with other paradigms such as SDN and edge computing (EC). Its alliance with SDN allows the virtualization of SDN controllers and forwarding and routing devices. Meanwhile, the convergence between NFV and EC ensures service feasibility and performance in 5G and beyond networks since this combination enables dynamic deployment of network services, scaling actions of VNFs, and cost reductions.

2.2.1 Load Balancing and Auto-Scaling of VNFs

Implementing network services at the edge [73] shortens its deployment while reducing bandwidth congestion, packet loss, and latency. Despite this potential solution, the deployment of VNFs at the network edge poses several challenges since they are more sensitive to traffic variations and are deployed on resource-constrained devices [6]. When the allocated resources are insufficient to meet the requested demands, the users perceive QoS degradation [7]. In contrast, service-associated costs increase if the assigned resources are over-provisioned.

Thus, the dynamic placement readjustment of VNFs along with scaling their assigned resources (e.g., number of instances or capacity) based on their workloads or user traffic demands is imperative. Nevertheless, scaling actions over virtual functions such as traffic accelerators and firewalls may cause occasional loops in the network and subsequent problems (e.g., slow and irregular connections and system failures). These functions are usually deployed in a BITW manner to avoid altering the communication endpoints. In this manner, load-balancing and auto-scaling mechanisms that promote network resilience are required. This section reviews the literature related to load-balancing and auto-scaling solutions in virtual environments.

The authors of [9] have provided insights into the network service descriptors (NSDs) by addressing fields related to scaling actions. Hinojosa et al. have highlighted the importance of a proper NSD configuration to ensure the appropriate scaling operations of VNFs. Additionally, they offer an overview of the different available scaling procedures in NFV and how to trigger them in an automated manner.

In [74], the researchers present an auto-scaling approach for the 5G data plane based on throughput utilization. This approach makes scale-out decisions based on the utilization threshold, while scale-in events are run periodically to remove empty data plane functions. Incoming user requests are assigned to the data plane with the highest throughput. Alawe et al. [75] have presented an algorithm based on control theory for the horizontal scaling and load balancing of 5G access and mobility management function (AMF). Furthermore, scaling actions are triggered according to the traffic load.

The authors of [76] write about a threshold-based scaling mechanism that considers workload variations of the evolved packet core (EPC) to perform scaling actions. Three modules comprise this proposal: data collection, scaling decision, and scaling execution. To evaluate their horizontal scaling procedure, the authors deploy a set of clusters for different EPC elements. Each cluster contains a load balancer that uses a round-robin algorithm to distribute the traffic among the cluster's instances.

Dutta et al. [7] have offered a solution for the dynamic and automatic scaling of VNFs. Their main objective was to ensure efficient resource utilization while improving the QoE of the offered services. In their solution, the incoming traffic is distributed to the pool members through a load-balancing mechanism based on a round-robin configuration. However, this approach implies packet modification since the load balancer works as a front-end service that reroutes incoming

requests.

Similarly, Ma et al. [8] have proposed a generic solution for the horizontal scaling of VNFs. Their solution relies on two load balancers in a master-slave configuration to distribute the traffic in the uplink and downlink directions. Moreover, they also address the flow affinity problem when balancing traffic among VNFs. They use connection-aware traffic load balancers based on a hashing function to maintain affinity between connections and network functions to solve the problem. The authors of [77] have demonstrated the importance of jointly considering load-balancing and auto-scaling strategies for VNF deployments. They have highlighted the necessity of designing these policies to improve QoS and enable more efficient use of resources.

Lange et al. [10] have developed an architecture for NFV that exploits the benefits of network softwarization and ML-based approaches to orchestrate and manage the lifecycle of VNFs. The proposed solution can perform scaling actions in advance by monitoring the deployed VNF's state and predicting its resources. At the same time, load-balancing actions are leveraged to OpenStack load-balancing features. However, the results obtained do not address the auto-scaling of VNFs and load-balancing performance. Thus, further experiments are necessary to evaluate their proposal's overall feasibility.

Despite the proposed solutions, none of those works addresses the issue of load balancing and auto-scaling transparent VNF clusters (i.e., those clusters whose VNFs are deployed in a BITW manner). Of these works, the most similar to the proposal presented in Chapter 4 is [8] because it presents a solution to address both issues by taking into account bidirectional flow affinity. The current research defines bidirectional flow affinity as the property of processing packets associated with a specific flow by the same network function in both directions of their data path (i.e., uplink and downlink). Utilizing flow affinity implies that packets belonging to the same flow will undergo similar treatments along their paths (e.g., processed by the same set of network functions) [78, 79]. Nevertheless, the solution of Ma et al. implies packet modification, which is not convenient for scenarios with BITW deployments as this type of deployment imposes additional challenges in terms of scalability.

2.2.2 Allocation Mechanisms of VNFs

The VNF allocation process represents one of the significant challenges of NFV since it implies the placement of requested functions within physical resources. Therefore, it has received much attention in the academic and industry sectors. The evolution of virtualization technologies introduces new scenarios where the virtual functions can be deployed not only in the cloud but also at the network edge. In this regard, network paradigms like EC and fog computing (FC), along with containerization technologies, enable the deployment of modern services such as the IoT, cooperative sensing, augmented reality, and Industry 4.0 near the end-users.

These services have stringent requirements regarding latency, availability, resilience, and scalability [80]. By deploying these services at the edge, their latency requirements can be

accommodated. Additionally, several nodes can interoperate to formulate clusters that enable failure recovery and accumulative processing capacity, thus satisfying availability, resilience, and scalability requirements. Through the virtualization of running network functions and deployed services, the cluster of nodes becomes more versatile, and service deployment is simplified and accelerated. Furthermore, future migration to newer generations of hardware becomes significantly less time-consuming [17].

In the last several years, SBCs have emerged as edge nodes for FC and EC either as standalone devices or in clusters. However, the application development for such devices should always take into account the constraints in computational resources, which are not on par with more expensive conventional CPUs and full-scale computers. Furthermore, the allocation mechanisms for such devices must consider energy levels (e.g., state of charge) since most of them are used in battery-dependent use cases. The following sections present several works related to these topics, i.e., SBC, battery state, and VNF allocation.

2.2.2.1 Single-Board Computers

In the last years, the applicability of SBCs has increased and now covers a wide range of use cases. In [15], Johnston et al. perform an in-depth analysis of the state-of-the-art use cases for these devices. The authors explain the main characteristics of SBCs and detail the different existent models. Additionally, they identify the broad domains where the SBCs might be deployed. Johnston et al. emphasize the importance of these devices not only as edge and fog nodes because of their power requirements and size but also as computational game changers that can shorten the data-generating parts of the networks by bringing computation closer to them.

To improve the resilience and performance of SBCs, the authors of [81–83] have used SBC clusters. These papers reveal two ways to deploy a cluster: by coupling physical elements together or by using the platform-as-a-service (PaaS) concept to create and manage the cluster. Bashford et al. [81] have presented a new method to create physical clusters of SBCs, called the Pi Stack. This method minimizes the amount of cabling required to create a cluster by reusing some elements of its physical construction as a communication channel for both power and management. Using the proposed technique, the authors compare three different SBC clusters composed of nodes from several vendors. Their results reinforce these devices' importance as infrastructure for future IoT deployments.

In contrast, Sagkriotis et al. [82] have explored the feasibility of a virtualized SBC cluster to host scalable containerized applications. By using Kubernetes [18], they achieve resilience of the deployed energy-monitoring application. They also demonstrate that an SBC cluster can host fog-oriented services. Likewise, Pahl et al. [83] have built an SBC cluster by considering the PaaS paradigm. However, they deployed their own dedicated tool to manage and configure the cluster instead of using a widely available platform like Kubernetes. Nevertheless, the authors do not compare it with other management platforms. Thus, a further evaluation must be done to

demonstrate the feasibility and benefits of their approach.

Taking the PaaS concept and the above papers as a reference, we develop an SBC cluster in Chapter 5 to deploy services and tasks while adhering to their requirements. We extend the Kubernetes scheduler by including energy measurements and estimations to deploy several events, thus improving the cluster's resilience by extending its lifetime while adapting Kubernetes to energy-constrained devices.

2.2.2.2 State of Charge Estimation

Knowing the remaining battery capacity in battery-powered devices helps avoid service disruption due to battery depletion. To this end, several works have aimed to estimate the state of charge (SOC) of the battery by using different methods.

In [84], Hu et al. present the extended Kalman filter (EKF) technique as an SOC estimation algorithm. The authors evaluate the proposed estimator using two types of lithium-ion (Li-ion) batteries under different loading profiles and temperatures. The optimal model parameters used in the EKF are obtained from generic functions for battery modeling that combines several degrees of polynomials. The findings reveal that the resulting model depends on the training datasets. Thus, updating model parameters due to changes in these datasets may be a challenge in real-time scenarios.

The researchers in [85] have developed an enhanced coulomb counting method for estimating the SOC and the state of health (SOH) of Li-ion batteries. They consider the correction of the operating efficiency and the impact on the SOH to improve the estimation accuracy. Due to their simple calculation and low hardware requirements, the authors' proposal can be easily implemented in all portable devices such as SBCs. Similarly, Sagkriotis et al. [82] have also developed an application based on the coulomb counting method to monitor and evidence the energy profile of nodes that comprise an SBC cluster. Their outcome indicates the low-energy consumption characteristic of these devices while executing virtualized services. In [86], Pop et al. propose a new SOC mechanism by combining direct measurement of the electro-motive force (EMF) and coulomb counting. They demonstrate the effectiveness of their approach by improving the SOC and accuracy of the remaining runtime.

In contrast to the previous work, the authors of [87] have suggested two methods for the actual bias modeling of batteries: a polynomial and Gaussian process regression model that uses a typical battery circuit model to examine the bias modeling and the SOC estimation. The results of their model depict a significant improvement in comparison with the baseline models (i.e., first- and second-order resistance-capacitance [RC] models). Furthermore, their approach is able to maintain similar computational efficiency.

The studies above have proven the accuracy of SOC estimation algorithms regarding battery capacity predictions. However, they consider battery operation parameters such as current, voltage, and temperature to make their estimations. The main focus of this thesis is to establish a

relationship between previous SOC measurements and utilized resources (e.g., CPU) to train models that predicts energy consumption (see Chapters 5 and 6). By doing so, we further contribute to automating the allocation of the VNF in such environments. Network-wide placement decisions are enabled through a model capable of predicting energy consumption. Such decisions can improve the network's total lifetime and optimize the management of the available resources.

2.2.2.3 Energy-Efficient Allocation Mechanisms

Although workload scheduling and VNF allocation are well-studied fields in cloud and fog computing (see a comprehensive study of the VNF resource allocation problem in [88]), there is a lack of practical implementation and prototypes in the reviewed literature. This thesis addresses this gap by testing the suggested solutions against a testbed comprised of representative SBC devices and batteries.

The authors of [89] have proposed a sleep schedule method based on compressive sensing to monitor environment data and device operating status for wireless sensor networks (WSNs). The approach arranges the working state of the nodes at different times to be in semi-sleep or sleep modes and selects the semi-sleep nodes at random in each cycle. Then, according to the gathered data from the active nodes, the whole network's data can be reconstructed. Similarly, Lim and Bleakley in [90] have presented an adaptive scheduling algorithm for WSNs, which determines the sampling schedule based on user accuracy goals, network connectivity, and preliminary collected data. Results show an improvement over baseline algorithms in terms of network lifetime. However, none of the previous papers has conducted any study on battery consumption, which is a crucial parameter for WSN environments where most nodes are battery-powered.

In [91, 92], the researchers have proposed energy-efficient scheduling algorithms to place tasks and reduce energy consumption in a cloud-computing environment. Both works assign tasks to the appropriate virtual machines, and hosts are selected based on characteristics and current capacity. However, both studies use a simulation environment (i.e., the Cloudsim Toolkit [93]). The authors evaluate their proposals rather than using a private cloud like OpenStack to demonstrate the effectiveness of their solutions in real scenarios.

Gazori et al. [94] have developed a DRL approach to address the task scheduling problem in fog-based IoT applications. As the primary function, its scheduler decides whether to process the task in a fog node or send it to the cloud data center. The authors include an energy consumption model in the scheduler's proposal, thus guaranteeing the selection of the most appropriate virtual machine in terms of power consumption. Likewise, the researchers in [95] have adopted a Q-learning algorithm to schedule tasks in an energy-efficient way. Their approach aims to minimize the task response time and maximize the CPU utilization of a node. The authors reduce the energy consumption of the whole cloud system by improving its resource utilization.

In [96], Varasteh et al. have offered a framework to solve the problem of the power-aware and delay-constrained joint VNFs' placement and routing (PD-VPR). In the first phase of the

proposed solution, a centrality-based ranking method maps the VNFs to physical nodes, and the delay budget between consecutive VNFs is split in a second stage. Then the shortest path through the selected nodes is found through the Lagrange relaxation-based aggregated cost (LARAC) algorithm [97].

The authors of [98] have addressed energy-efficient VNF placement and chaining over NFV-enabled infrastructures by proposing an algorithm to minimize the power consumption of servers and networking devices involved in accommodating VNF chains. To overcome the NP-hard nature of this problem, Soualah et al. have formulated a decision tree model, which has been solved through a proposed energy-efficient algorithm based on Monte Carlo tree search strategy [99]. The algorithm creates the decision tree continuously until finding an optimized solution. The researchers have evaluated the performance of its proposal in a private SDN/NFV infrastructure through extensive simulations. The results show that the proposed algorithm reduces resource usage and power consumption.

Jayanetti et al. [100] have presented an RL model for energy and time-optimized scheduling of tasks in edge/cloud environments. Its design integrates energy and deadline in the reward model to train the agent. Thus, it can establish a trade-off between conflicting objectives such as energy optimization and time minimization in workload executions. The authors also introduce a hybrid DRL model comprising multiple-actor networks and one critic network. The researchers used the Cloudsim simulation toolkit for evaluation purposes to test their approach. The evaluated dataset represents a synthetic workflow structure created through the Pegasus workflow framework [101].

The authors of [102] address the VNF readjustment and consolidation problem by modeling it as an integer linear programming (ILP) issue that considers a trade-off between the minimization of latency, hardware utilization, service-level objective (SLO) violation cost, and VNF readjustment cost. To boost the feasibility of the model in large-scale networks, Wahab et al. propose an optimized k-medoids clustering approach that proactively divides the substrate network into several disjointed clusters. Then each cluster aims to optimize some parameters such as CPU, energy, and delay. Simulation results show that the proposed solution reduces the readjustment time while decreasing resource utilization compared with baseline solutions (e.g., K-means).

In [103, 104], Eramo et al. have proffered a heuristic method based on the Viterbi algorithm [105] to determine the migration policy with low computational complexity in response to variations in the service requests. The former is focused on cold migrations since the considered virtual machines are redundant and suspended before performing a migration action. The latter goes ahead and considers VNF placement, service routing, and VNF migration when changing the service request, thus performing live migrations.

Pei et al. [106] have focused on the VNF placement problem in SDN/NFV environments, which they model as binary integer programming (BIP). They propose a double deep Q-network-based VNF placement algorithm (DDQN-VNPA) to solve it by considering the VNF placement cost, the instances running cost, and service rejection penalties. Their approach determines the optimal

solution from a prohibitively large solution space in the first stage. Then the VNFs are placed or released according to a threshold-based policy. The researchers evaluate its proposal with trace-driven simulations on real network topology. Despite achieving good results in comparison with baseline algorithms in terms of service rejection ratio, end-to-end delay, and VNF running time, Pei et al. do not analyze the energy consumption since this parameter was encapsulated in the instance running cost.

The authors of [107] introduce the accessible scope concept as a constraint to narrow the searching space by dividing into small groups those servers that can be used to allocate a specific request. Thus, the solution space is reduced, and the time efficiency of the VNF allocation is improved while minimizing the server energy consumption. After considering the accessible scope, Qi et al. apply the multi-stage graph algorithm [108] and greedy algorithm [109] to verify the influence of the accessible scope on the acceptance ratio, energy consumption, and bandwidth usage. The obtained results reveal how the proposed approach significantly reduces the runtime in large-scale network scenarios concerning those in which the accessible scope is not used. However, the results do not analyze the energy consumption metric despite the fact that minimizing this value is one of the primary objectives.

In [110], Mu et al. describe a method that considers energy consumption and performance interference when allocating VNFs. The authors formulate the problem as a bin-packing one that is NP-complete. Thus, they implement two solutions according to the homogeneity of the servers. In the case that all the servers are of the same type, the researchers propose a first-fit heuristic (FFH) algorithm to solve the problem with a lower bound. For a more general case, Mu et al. introduce the deep deterministic automatic placement (DDAP), which is based on DRL. The results show that DDAP achieves lower energy consumption and running time cost with respect to state-of-the-art methods such as FFH and ant colony system (ACS).

Santos et al., in [111], have formulated a system model for dynamic service function chaining (SFC) placement regarding computing resources, availability levels, and energy consumption. To solve the SFC problem, the authors propose two policy-based RL algorithms: proximal policy optimization (PPO) and advantage actor-critic (A2C). These algorithms aim to minimize energy consumption while considering availability levels. The performed simulations validate that the proposed algorithms can be used for SFC deployments while guaranteeing better results than a greedy approach in terms of energy consumption and acceptance rate. Pursuing similar objectives, the authors of [112] utilize an RL method to implement a VNF placement policy to address the VNF forward graph embedding (VNF-FGE) problem. Solozabal et al. seek to optimize the VNF-FGE by applying neural combinatorial optimization (NCO) [113] with defined constraints such as latency, bandwidth, resource utilization, and power consumption. For validating purposes, the researchers compare the feasibility of the proposed approach by comparing its solutions with the Gecode solver [114] and FFH algorithm. Similar to previous works, they lack an analysis related to energy consumption in the results.

The authors of [115] have presented an energy-aware game-theory-based solution for VNF allocation within NFV environments. They consider each VNF as a player of the problem which competes for the nodes' capacity pool, aiming to minimize individual cost functions. In this game, the network nodes dynamically adjust their processing capacity by considering an adaptive rate strategy, which seeks to minimize energy consumption and processing delay. As a result of the nodes' strategy, the resource-sharing cost of the VNFs assumes a polynomial form in the workload, thus admitting a unique Nash equilibrium. The authors have evaluated its proposal by comparing the energy consumption and processing delay between those obtained from a game played and non-energy-aware nodes and VNFs. The results show a negligible effect on the processing delay while saving significant power consumption.

In [116], Tajiki et al. have described a resource allocation architecture that enables energy-aware SFC for SDN-based networks by considering delay, link utilization, and server usage constraints. Then the authors formulate several problems related to the VNF placement, allocation of VNFs to flows, and flow routing as ILP optimization problems. To solve the problems in acceptable timescales, the researchers implement a set of heuristics (i.e., nearest service function first, offline and online resource allocation) to find near-optimal solutions. The proposed heuristics have been evaluated in simulation environments by comparing them with the mathematical models. The results show an optimality gap of 14% between the heuristic algorithms and the optimal solution when analyzing energy consumption.

Khemili et al., in [117], have proposed a placement algorithm that aims to maximize the exploitation of MEC's resources in a balanced manner. To this end, they apply formal concept analysis (FCA) [118] in the first stage to place VNFs by considering their sequencing order in the SFC. In the second phase, the researchers use Fuzzy-FCA to consolidate VNF groups into the least number of virtual machines. To evaluate the proposed algorithm, this study compares its performance with the MultiSwarm algorithm [119] in terms of packaging efficiency, server and network energy consumption, latency, and resource cost. The implementation and evaluation of the algorithms use the Edge-CloudSim tool, and the results show that the proposed mechanisms minimize the number of active virtual machines used in the VNF allocation, thus reducing energy consumption with regard to the baseline strategy.

Thanh et al. [120] address the resource and energy utilization modeling for IoT applications in edge/cloud environments. More specifically, they have deployed a small use case in a testbed to investigate the energy and resource usage of SFC embedding strategies. Then the authors propose a resource and energy-aware SFC strategy to cope with dynamic load and traffic situations due to several SFC requests. Its proposal has been evaluated in a flow-level event-based Java simulator (i.e., BK-EdgeCloud) by considering energy consumption, acceptance ratio, and utilization. Obtained results outperform existing approaches when analyzing the above metrics, overcoming relevant challenges from real deployments while satisfying IoT applications' demands.

The authors of [121] have focused on the problem of assigning and scheduling flows to VNFs

in an energy-efficient manner. By considering that the VNFs are already deployed in the physical machines and traffic flows with deadlines, Assi et al. formulate the mathematical problem as a flows-to-VNF assignment and scheduling problem. Then they propose an energy-aware VNF scheduling using genetic algorithms to avoid solving an ILP. This heuristic divides the original problem into two sub-problems: mapping flows to VNFs and scheduling flows on VNFs. The researchers evaluate its proposal with respect to two other solutions considering different criteria: minimize the makespan and number of servers used. The comparison reflects that the proposed algorithm combines the advantages of baseline approaches by reducing energy consumption.

2.3 Open Issues

Although extensive research can be found regarding topics related to resilience, fault tolerance, and VNF allocation in SDN/NFV environments, most of these studies still include several limitations that restrict their applicability in 5G and beyond network use cases. Some of these limitations and open issues are summarized as follows:

- Little or no research work implements their approaches in real-world testbeds. They are evaluated in simulation environments that require high values of computational resources, thus limiting their applicability in real use cases.

- There is a lack of solutions based on the DDS concept approaching the eastbound/westbound interface in SDN architecture. Existing works use this technology as a southbound interface to communicate with network devices (e.g., routers and switches).

- Most existing studies addressing the resilience and fault tolerance capabilities of the network using SDN controllers are limited to applying multi-path and load-balancing solutions. However, these devices can also guarantee these characteristics by allocating virtualized functions and applying routing decisions to deploy service requests over multiple domains.

- Despite the wide range of studies addressing the load balancing and auto-scaling of VNFs as methods to improve network resilience, none of them considers the own characteristics of the instances compounding the cluster over which they apply their approaches. For example, VNFs such as firewalls and traffic accelerators are usually deployed in a BITW manner. This type of VNFs represents a transparent cluster when they are scaled up. Thus, network loops and associated problems may appear without load-balancing and auto-scaling mechanisms that consider this characteristic.

- Most state-of-the-art research neglects the evaluation of energy-efficient scheduling algorithms in resource-constrained nodes such as SBCs.

- Despite the plethora of works addressing energy-efficient scheduling algorithms, most use classic energy models in which maximum and idle values are considered, with the ultimate goal of minimizing the total consumption by reducing resource utilization. Additionally, unrealistic assumptions are supposed since energy values are arbitrarily selected. Consequently, most solutions lack the flexibility and applicability to be adapted to envisioned use cases in 5G and beyond networks.

To fill the identified gaps in the literature, this thesis proposes several strategies involving a DDS mechanism and scheduling algorithms to improve the network resilience when using SDN and NFV environments. The former enables the exchange of network information between SDN controllers and other entities using the proposed solution (see Chapters 3, 4, and 6). The proposed solutions of VNF allocation assign events to resource-constrained nodes while guaranteeing low energy consumption and increasing resilience (see Chapters 5 and 6). The set of nodes utilized to evaluate the proposed approaches are off-the-shelf SBCs, typically employed in use cases such as unmanned aerial vehicles (UAVs) and WSN. The implemented schedulers consider the controller node in the allocation process, therefore increasing the number of scheduled events and the acceptance ratio of the systems used. In accordance, our solutions better use the clusters' available resources since they consider the computing and controller nodes.

CHAPTER 3

HIERARCHICAL SDN ARCHITECTURE: DESIGN AND IMPLEMENTATION

This chapter is based on:

- **A. Llorens-Carrodeguas**, C. Cervelló-Pastor, I. Leyva-Pupo, J. M. Lopez-Soler, J. Navarro-Ortiz, and J. A. Exposito-Arenas, "An Architecture for the 5G Control Plane based on SDN and Data Distribution Service," In *2018 Fifth International Conference on Software Defined Systems (SDS)*, Barcelona, Spain, 2018.
- **A. Llorens-Carrodeguas**, C. Cervelló-Pastor, and I. Leyva-Pupo, "Software Defined Networks and Data Distribution Service as Key Features for the 5G Control Plane," In *Distributed Computing and Artificial Intelligence, Special Sessions, 15th International Conference, DCAI 2018*, Advances in Intelligent Systems and Computing, vol 801. Springer, Cham, 2018.
- **A. Llorens-Carrodeguas**, C. Cervelló-Pastor, I. Leyva-Pupo, J. M. López-Soler, and J. Navarro-Ortiz, "The Role of Data Distribution Service in Failure-Aware SDN Controllers," In *XXXIII Simposium Nacional de la Unión Científica Internacional de Radio (URSI)*, Granada, Spain, 2018.
- **A. Llorens-Carrodeguas**, C. Cervelló-Pastor, and I. Leyva-Pupo, "A Data Distribution Service in a Hierarchical SDN Architecture: Implementation and Evaluation," In *28th International Conference on Computer Communication and Networks (ICCCN)*, Valencia, Spain, 2019.

This chapter presents a fault-tolerant SDN architecture for the control plane of 5G and beyond networks by introducing a new mechanism to federate the controllers in the proposed system. Section 3.1 introduces the SDN hierarchy in the control plane of the 5G network after considering it as a reference for the research development. In Section 3.2, we explain the constituent modules of the hierarchical SDN controllers as well as the role of DDS in the proposal. Regarding the implementation details of the DDS application, Section 3.3 describes the functions of the constituent modules of this application and how they interact with other controllers' applications. Additionally, a detailed description of the built testbed and the conducted experiments is presented in Section 3.4. Finally, Section 3.5 evaluates the behavior of the DDS application in the SDN controllers. More specifically, we analyze the obtained results in three different scenarios regarding communication latency and overhead. Moreover, we compare the CPU consumption of the controllers while acting as publisher or subscriber devices.

3.1 Hierarchical SDN Architecture in 5G and Beyond Networks

5G and beyond networks can adopt the proposed architecture since our system considers the CUPS concept. The separation of these planes ensures that their resources are scaled independently. It also allows the placement of the user plane functions closer to users, thereby reducing network response times and bandwidth consumption. To better illustrate how our proposal works in these networks, we have taken the first 5G system architecture standard defined by the 3GPP in its specification TS 23.501 [122].

The 5G 3GPP architecture is based on the concepts of control and user planes split, service-based architecture, and network slicing. This architecture is mainly compounded by the following network functions: authentication server function (AUSF), access and mobility management function (AMF), network slice selection function (NSSF), policy control function (PCF), session management function (SMF), unified data management (UDM), user plane function (UPF), application function (AF), radio access network (RAN), data network (DN), and user equipment (UE) [122].

The proposed system considers major requirements such as scalability, reliability, and fault tolerance. In this regard, we introduce an SDN controller hierarchy to satisfy these demands by bridging between the control and user planes' elements (i.e., the SMF and the UPF). The hierarchical architecture is composed of two types of SDN controllers: the area controller (AC) in the bottom level and the global controller (GC) in the top level. In this way, we ensure scalability because specific functions can be defined according to the role of each kind of controller. Communication among these entities is achieved through specific applications running on them.

Figure 3.1 illustrates the 5G control plane architecture along with the proposed hierarchical SDN system. We introduce the *agent* concept in this plane by considering the EC paradigm that seeks to place processing and storage resources closer to end-users. Therefore, these elements

(i.e., the agents) will be near the users to communicate with them and collect network information such as connectivity, mobility, and communication patterns, which enables anomaly detection and behavior and QoE prediction by the network operators. The agents only collect information from users different from the 5G network data analytic function (NWDAF) [123], which collects data from the 5G core network functions.

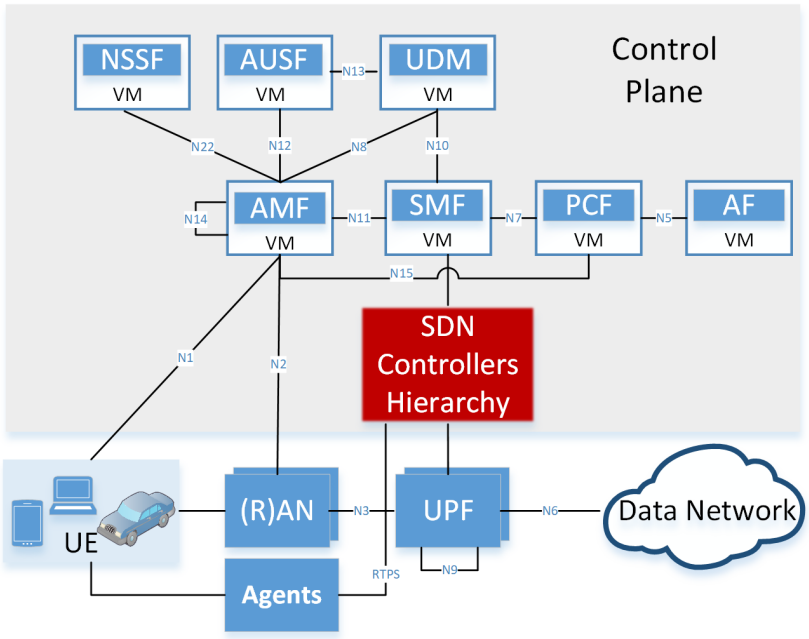


FIGURE 3.1. Proposed hierarchical SDN architecture in the 5G control plane.

3.2 Design of the SDN Controller Hierarchy

The upper layer of the hierarchical system is formed by a group of federated GCs. These controllers are responsible for managing and controlling the ACs in the bottom layer of the hierarchy. From a functional point of view, the GCs perform load balancing and keep a global network view. Meanwhile, the ACs are in charge of the UPFs' control and flow management. Figure 3.2 depicts a deeper insight into the SDN controller hierarchy, showing the relationship among the constituent modules of both types of controllers. It must be noted that these blocks have different functions regarding the kind of controller.

Both types of controllers use DDS to exchange network information. The GCs communicate to maintain a consistent network state and establish interdomain flow routes. Similarly, the ACs update their GCs using DDS when a topology modification occurs or any assigned node changes its state. Likewise, the GCs inform their ACs when the global topology changes and could affect the communication among nodes of different ACs. In addition, the agents also use DDS to communicate with GCs and ACs. In this manner, the controllers are notified of a service

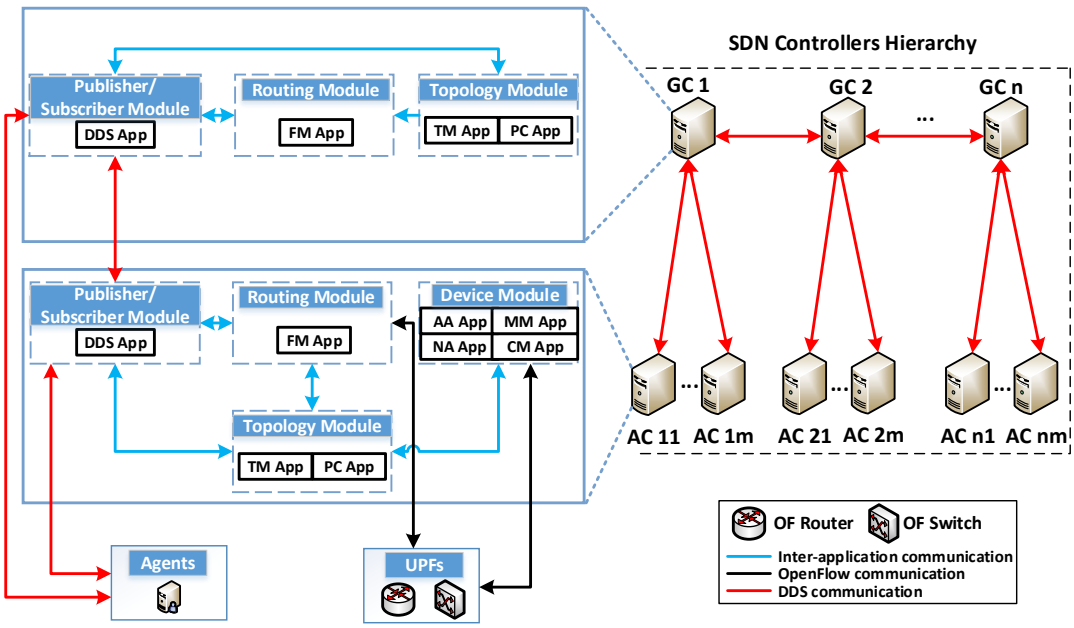


FIGURE 3.2. Hierarchical architecture of federated SDN controllers.

request and its execution status. Furthermore, they can receive collected network information and take action in advance by considering the gathered data.

Moreover, the use of DDS allows a better performance during recovery stages since GCs exchange their network information. Thus, other GCs can assume their functions when a problem arises due to a controller's failure.

3.2.1 Global Controller Modules

This kind of controller has been envisioned with a modular design where each module performs specific functions (i.e., publishing/subscribing, routing, and topology management). The logic of these functions runs on top of the SDN controllers, as shown in Fig. 3.2.

The *publisher/subscriber module* enables the exchange of network information between GCs and ACs. This block also receives messages from the agents related to network events such as link failures, broken controllers, and service requests. The DDS application (DDS App) is responsible for the logic of this module and is based on an OMG's standard middleware for distributed real-time applications.

Inter-area forwarding rules are applied by the *routing module* to enable the communication between user plane devices belonging to different ACs. The flow management application (FM App) is responsible for the module's logic. Additionally, it frequently communicates with other applications (e.g., DDS App) to perform its functions.

The management and storage of the network information related to ACs and GCs is in control of the *topology module*. It comprises two main applications: the topology management application

(TM App) and the path calculation application (PC App). The former manages a database with the network information (i.e., nodes and links). The latter executes a constrained-based shortest-path algorithm (e.g., the Dijkstra algorithm) to calculate new routes based on the physical topology and its utilization information.

3.2.2 Area Controller Modules

ACs have a similar design as the one described for the GCs. This type of controller adds an extra module to perform the functions related to user plane devices' control and management. Therefore, the ACs are composed of four modules.

The *publisher/subscriber module* executes functions identical to its homologous block in the GCs except for not exchanging information with other ACs. Similarly, the *routing module* through the FM App applies intra-area forwarding rules by considering the network information. Likewise, the TM App and PC App also run on the *topology module*. More specifically, this module manages, through these applications, the network information associated with physical devices (e.g., routers and switches) connected to the AC and calculates new routes when there is a change in the topology.

The *device module* manages information regarding connected end-user devices through several applications that run on them: the mobility management application (MM App), the connectivity management application (CM App), the authorization and authentication application (AA App), and the network access application (NA App). The MM App is in charge of the handover procedure for wireless users. In addition, the CM App and AA App are responsible for authenticating new users and managing service requests. Finally, the NA App attends to the setup procedure between the user's device and the network by establishing initialization parameters such as QoS and bandwidth. These applications do not intend to replace existing control plane entities in the 5G service-based architecture. In fact, they can perform along with some of the 5G entities, such as the case of AMF.

It should be noted that the above-mentioned modules related to GC and AC can exist according or not to the selected controller distribution during the implementation phase. This research does not involve the implementation of all the mentioned modules. It only addresses the one corresponding to the exchange of network information and federating SDN controllers (see Section 3.3) since this represents a primary element to improve the fault tolerance in the proposed architecture.

3.3 Implementation of the DDS Application in SDN Controllers

The main functions of the DDS App are the exchange of network information, the detection of new and failed controllers, and the synchronization among them. This application is composed of three blocks: publisher, subscriber, and synchronization. Its modularity implementation offers the

ability to simultaneously send and receive network information. The following sections explain the performance of each element of the DDS App.

3.3.1 Publisher Block

This component is charged with sending network information to other controllers. However, several steps are necessary before performing this simple function. In this regard, a proper configuration of some QoS parameters (e.g., the persistence service, publisher's queue, and reliability) is required to avoid outperforming data consumers in the case of slower subscribers. After completing the initial configuration, the publisher block in each controller verifies that it has an active connection with other controllers before sending any data.

Once the controllers are synchronized, the publisher sends data while developing new information. Toward this aim, it gathers the network information from the controller's database (e.g., the MD-SAL data store in OpenDaylight controllers) and sends it through a topic named "Topology". This topic represents a data stream composed of eight fields: Identifier, NodeId, TerminationPointId, LinkId, SourceNode, SourceNodeTp, DestinationNode, and DestinationNodeTp. Additionally, each field constitutes a string of data. Through the identifier field, the publisher notifies the kind of information (i.e., node, flow, and link) it will send.

Finally, this element sends the topic to each registered controller. During this process, the synchronization block is responsible for managing the connection (see Section 3.3.3). Figure 3.3 depicts the algorithm of this component to clarify its performance in the controller.

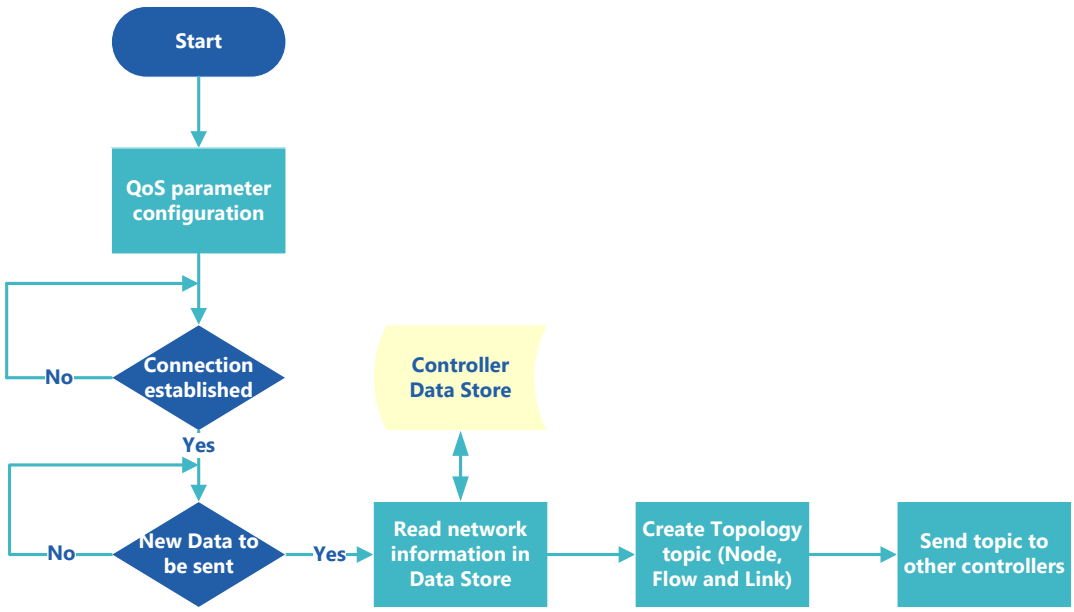


FIGURE 3.3. Publisher block flowchart.

3.3.2 Subscriber Block

In brief, this element receives the network information sent by other controllers. Similar to the previous block, the configuration of several QoS parameters is necessary for a well-synchronized connection with the publisher controllers. However, their designs are different because the subscriber implementation dynamically uses listeners to perform its functions.

In this sense, we have configured several states to recognize when new data can be read. With this configuration, a listener is invoked when a defined status changes. In this manner, we avoid polling the publisher, thus saving time and resources.

After triggering the listener, the subscriber reads the network information from the "Topology" topic and saves the gathered data in the controller's database by considering the identifier field. It should be noted that the process of saving information into the data store can simultaneously run with the listener-invocation action. Figure 3.4 illustrates the behavior of this block.

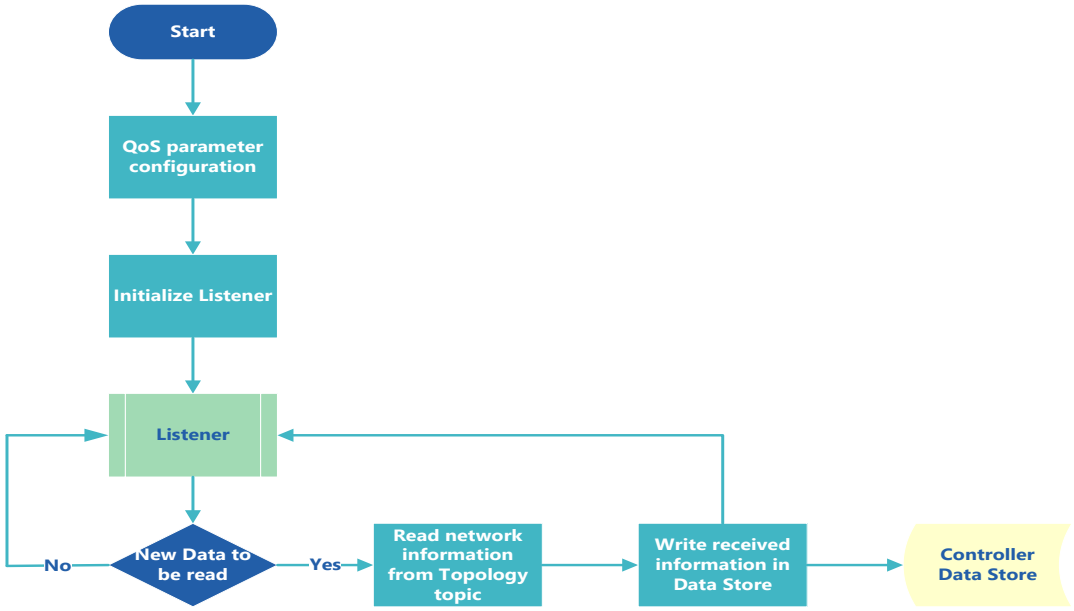


FIGURE 3.4. Subscriber block flowchart.

3.3.3 Synchronization Block

This element is in charge of the synchronization among controllers regarding their role in the network (i.e., GC and AC according to the proposed hierarchy in Section 3.2). Unlike the previous components, the QoS parameter configuration and the listener initialization must consider the controller's role. In the case of the ACs, they can only exchange network information with its GC. In this regard, it is necessary that the GCs communicate among themselves to keep a global network view as well as detect variations in their ACs' status (e.g., alive or failed).

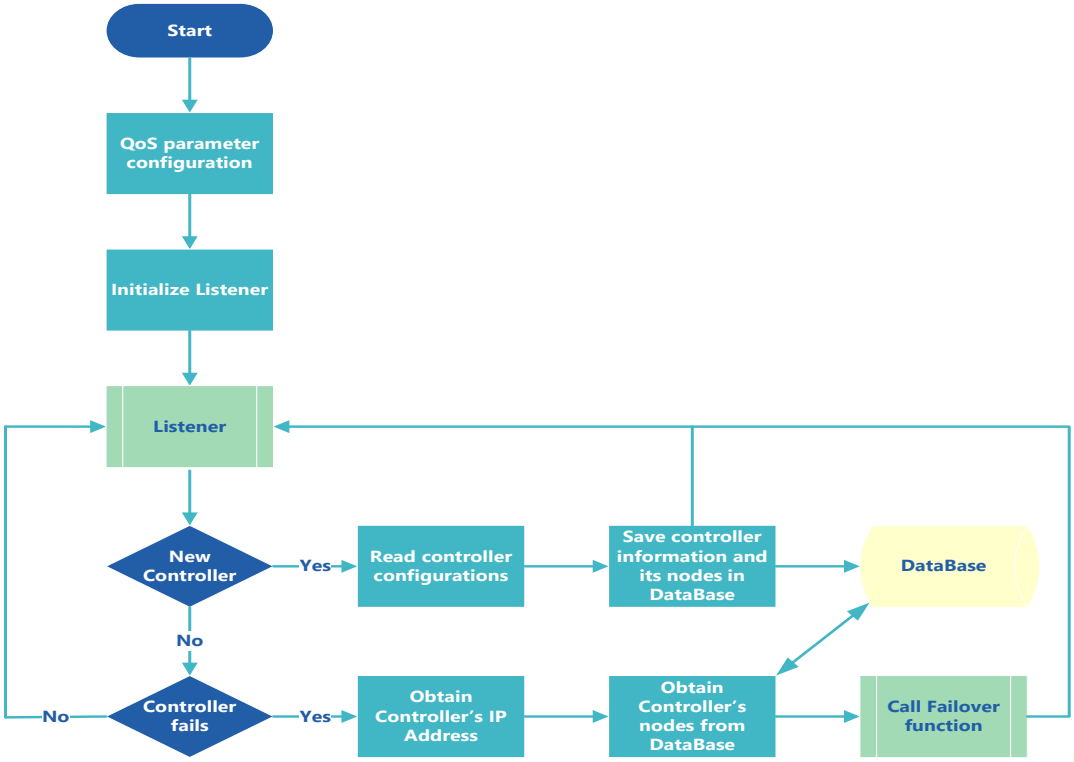


FIGURE 3.5. Synchronization block flowchart.

Additionally, we introduce an authentication procedure that plays an essential role during the discovery phase. More specifically, this process ignores any participant not compatible with the configured security profiles. In this way, the controllers cannot receive information from unregistered entities in the discovery phase.

Once the GC has initialized its listener, it can detect a new AC or a failed one within its associated ACs. In the former case, the GC receives the AC's information and saves it into an internal database. In the latter case, after detecting a failed AC, the GC gathers its IP address to correlate it with its assigned nodes. Finally, the synchronization component can trigger a failover mechanism to reassign those nodes to other ACs. Figure 3.5 depicts the algorithm of this block.

3.4 Testbed Implementation

Considering the proposed SDN controller hierarchy, we built a national SDN testbed composed of two GCs that individually manage two ACs. The GCs were physically distributed in Granada (University of Granada, UGR) and Barcelona (Universitat Politècnica de Catalunya, UPC), and their ACs were placed in the same locations, communicating only with their GCs. Thus, we have two SDN domains as shown in Fig. 3.6. In this figure, the blue dashed line represents the DDS connections over RedIRIS [124].

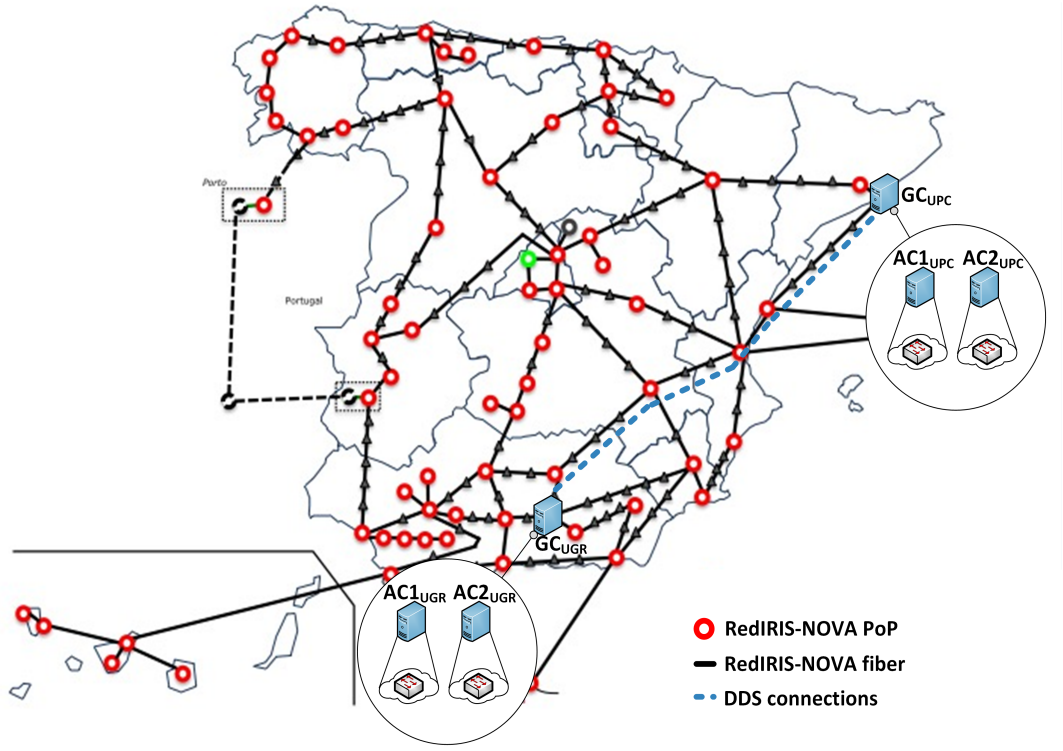


FIGURE 3.6. SDN testbed proposal using DDS to exchange information among controllers.

In the implemented testbed, the GCs were configured to support communication over wide access network (WAN). To this end, we set a public IP address for each GC that would be announced to other GCs during the discovery stage to establish the connection. Additionally, we configured the IP network address translation (NAT) to map the public IP address to the private local access network (LAN) where the controllers were running (SDN domain). Then we utilized port forwarding to exchange the discovery and user data traffic between the private ports used and their corresponding public ports. In this manner, the GCs could share network information and discover each other. Similarly, the ACs were configured to support communications over WAN when their GCs are geographically distant (i.e., other SDN domains). Nevertheless, our premise was to use private LAN for communicating controllers within the same SDN domain.

3.4.1 Hardware and Software Configuration

The SDN controllers were instantiated as virtual machines with 1 CPU and 2 GB of RAM. The UPC's SDN domain runs over an OpenStack-based cloud formed by four servers, providing the following pool of resources: 48 VCPUs and 256 GB of RAM. Similarly, the UGR's SDN domain is hosted by an OpenStack Mitaka environment with the following resources: 4 VCPUs and 32 GB of RAM.

Before implementing the DDS App, we had to choose a controller distribution to use in our testbed and performed a comparative study of different controllers for our selection. The authors of [125] have considered several features in their research such as cross-platform compatibility, southbound and northbound interfaces, OF support, network programmability, efficiency, and partnership. Regarding the study's outcome, we selected the OpenDaylight controller due to its modular framework and wide variety of environments where it can be used. Furthermore, OpenDaylight's applications can collect network information and perform analyses through third-party algorithms, indisputably increasing its applicability in telco use cases. To implement the DDS App, we used the RTI Connext DDS 5.2.3 library, Java Development Kit 1.8, Eclipse Luna, and Maven 3.3.3.

Our testbed uses two main applications: L2Switch and DDS App. The former is a native application of the OpenDaylight controller. It learns about the source media access control (MAC) address through the incoming packets and teleports them to their destinations when they are known. Here, the term "teleport" indicates that the packet is sent to its destination without flooding the network. The implemented DDS App allows communication among controllers to exchange their network information. When the proposed mechanism is installed in the controller, it automatically creates DDS entities (e.g., domains, topics, publishers, subscribers, data writers, and data readers).

The applications above use the MD-SAL data store to perform their functions and so do not interfere with each other. Figure 3.7 depicts the message flows between these applications and other OpenDaylight modules. To better understand this picture, we consider the case of a new flow arriving at one of the switches controlled by the controller (step 1). Since the OFS does not have any information related to the incoming flow in its tables, it sends an OF PACKET_IN message to its SDN controller (step 2). This message is received by the OF plugin. It provides several features to guarantee the exchange of OF messages between the switch and the controller [126]. In step 3, the OF plugin forwards the received message to the packet notification module in the MD-SAL block. This module is responsible for notifying the "on packet received" event to the blocks using this trigger, in our case DDS App and L2Switch (step 4).

Following the logic of our example, the above notification is handled by the packet handler module in the L2Switch App, which decodes the incoming packets and dispatches them appropriately. Thus, it sends the decoded packet to the address tracker module to learn the MAC and IP addresses of the network's entities (step 5). In step 6, the learned information is stored in the corresponding block of the MD-SAL data store (i.e., the operational L2-Address). The packet handler also indicates to the flow writer module to create the MAC-MAC flows when the source and destination MACs are known (step 7). Likewise, the created flows are stored in the data store, saved specifically in the configuration flow block (step 8). In step 9, the flows are translated by the OF plugin to allow an upright configuration in the OFS. Once the packet handler informs the packet transmission (step 10), the OF plugin adds the flows in the switch's table or teleports

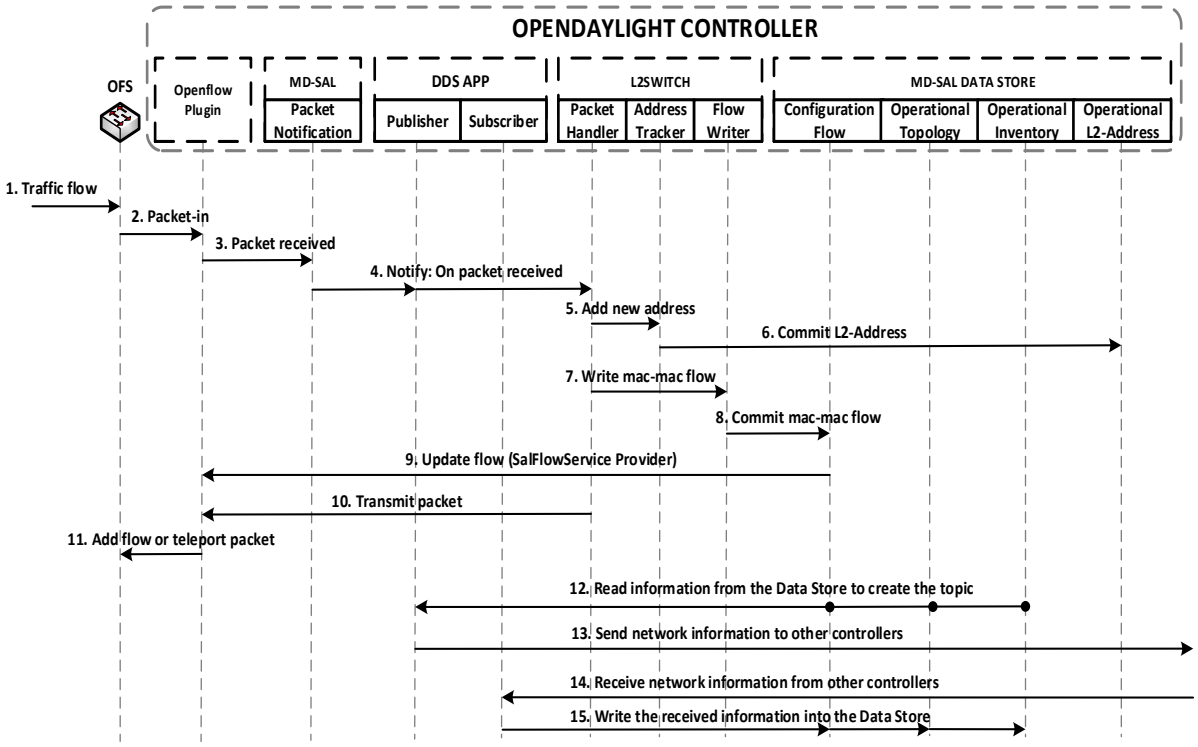


FIGURE 3.7. L2Switch and DDS App communication with OpenDaylight's modules.

the packet according to the created rule (step 11).

Despite the following flow messages appearing after step 11, it does not mean they always occur in that manner since the associated messages of the DDS App can be treated independently of the L2Switch's ones. Once this aspect has been clarified, step 12 is related to the case in which the publisher module of the DDS App obtains information. More specifically, it reads network data from the MD-SAL data store blocks to create the topic, which is used to exchange network information associated with existing nodes, ports, links, and flows (step 13). In contrast, the subscriber module receives the network information from other controllers (step 14) and stores them in the corresponding modules of the MD-SAL data store (step 15).

3.5 Evaluation and Results

To evaluate the behavior of the DDS App in the implemented testbed, we defined two metrics to characterize the controllers' performance in a hierarchical architecture: overhead and latency. The former describes the aggregated data rate while exchanging network information among controllers. The latter represents the amount of time since a controller generates an event until other controllers are aware. This metric is a crucial parameter in distributed systems and can be influenced by the data size, the available bandwidth, the propagation delay, and the processing

time of the controllers. Finally, this section evaluates the CPU consumption by considering the amount of data exchanged among controllers.

3.5.1 Experiment Description

To generate the network information exchanged among controllers, we emulated the underlayer network controlled by each AC using Mininet [127]. In particular, three different networks were used to evaluate our implementation. One was designed to represent a minimal topology formed by two nodes. The remaining networks were taken from the Internet Topology Zoo [128] (i.e., Abilene and BtNorthAmerica networks). We utilized two GCs and two ACs in each scenario, specifically, GC_{UPC}, GC_{UGR}, $AC1_{UPC}$, and $AC2_{UPC}$.

To gather statistically meaningful results, we ran 10 independent experiments per scenario. We started the controller instances for each run and waited a defined time to guarantee the participants' discovery through the DDS App. Then we initialized the network topology by randomly assigning the nodes to the ACs. When the nodes were connected to their ACs, we injected traffic into the network to begin the flow configuration and exchange of network information among controllers.

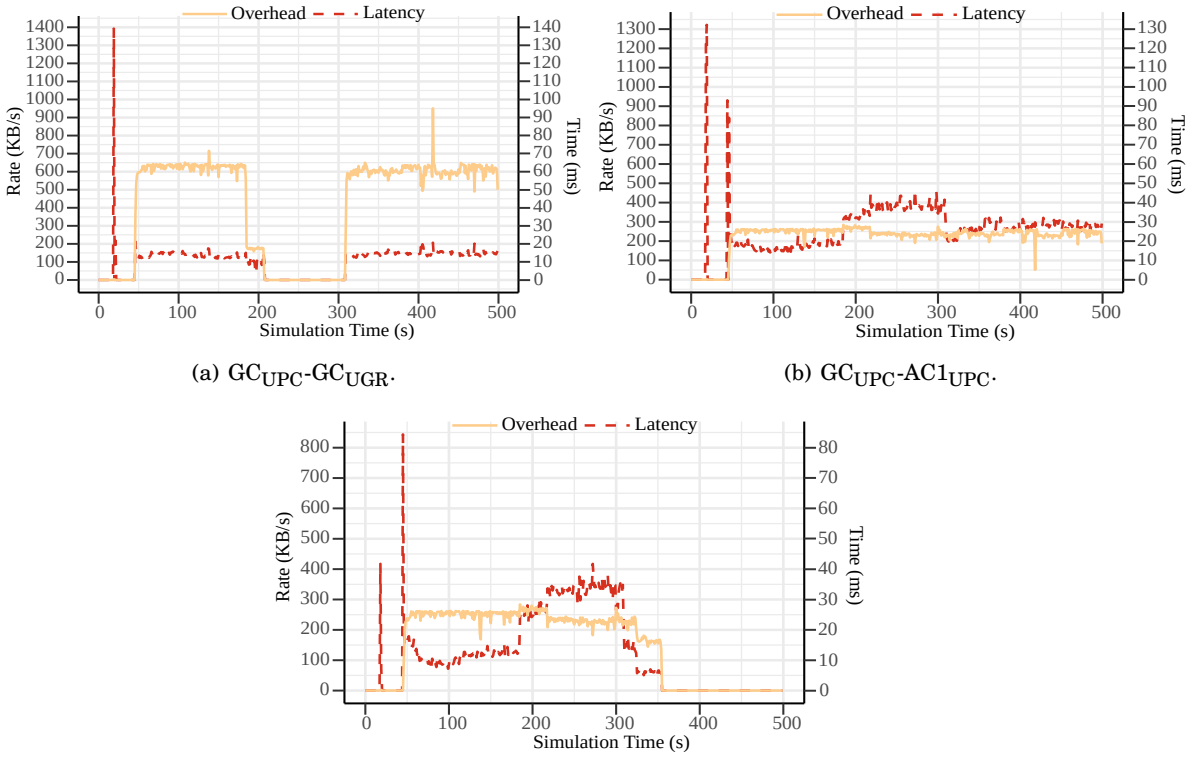
Firstly, since the incoming packets were empty, they did not match any flow rule in the OFS tables. Therefore, the OFSs sent OF PACKET_IN messages to their ACs. After receiving these messages, the controllers determined a path to reach the packet's destination and create a new flow entry by sending an OF FLOW_MOD message to their OFSs. In this manner, the ACs reactively install forwarding rules in the network's switches. To guarantee distributed network information and increase the reliability and fault tolerance of the network, the $AC1_{UPC}$ and $AC2_{UPC}$ shared their assigned nodes and new configured flows with the GC_{UPC}. Similarly, it processed that information and forwarded it to its homologous controller in the other SDN domain, the GC_{UGR}.

The following experiments assume that the propagation delay is a constant value since it depends on a fixed distance between the devices and the controllers. Additionally, we do not limit the available bandwidth to represent an approximated upper bound on the overhead. Thus, we can reduce the controllers' overhead by limiting the available bandwidth.

Several events can occur in the scenario networks, such as nodes and links discovery and flow installation. Therefore, the controllers must notify the changes in the network state. In this regard, the higher the number of consecutive events, the higher the controllers' overhead.

3.5.2 Minimal Scenario

The first scenario is called minimal and is composed of two nodes that have been assigned to different ACs. Through this scenario, we aimed to evaluate the synchronization among controllers in small SDN domains. In this regard, we assess the communication among GCs and the DDS App's behavior by emulating a failure in the GC_{UGR}. The results are depicted in Fig. 3.8.



(a) GC_{UPC}-GC_{UGR}.

(b) GC_{UPC}-$AC1_{UPC}$.

(c) GC_{UPC}-$AC2_{UPC}$.

FIGURE 3.8. Synchronization latency and overhead in minimal network.

This figure shows that the maximum delays (i.e., 80–140 ms) occur during the discovery phase of the controllers (i.e., at the beginning of the experiment). These delays decrease after this process and remain below 50 ms during the rest of the experiment. In the case of the overhead metric, the GCs communication achieved the highest value (i.e., approximately 950 KB/s), as shown in Fig. 3.8(a). In general, GCs communication overhead is around 600 KB/s during most of the experimental time since they exchange all the network information. In contrast, the communication between the ACs and the GC_{UPC} shows lower values (i.e., around 200–300 KB/s), as seen in Figs. 3.8(b) and 3.8(c).

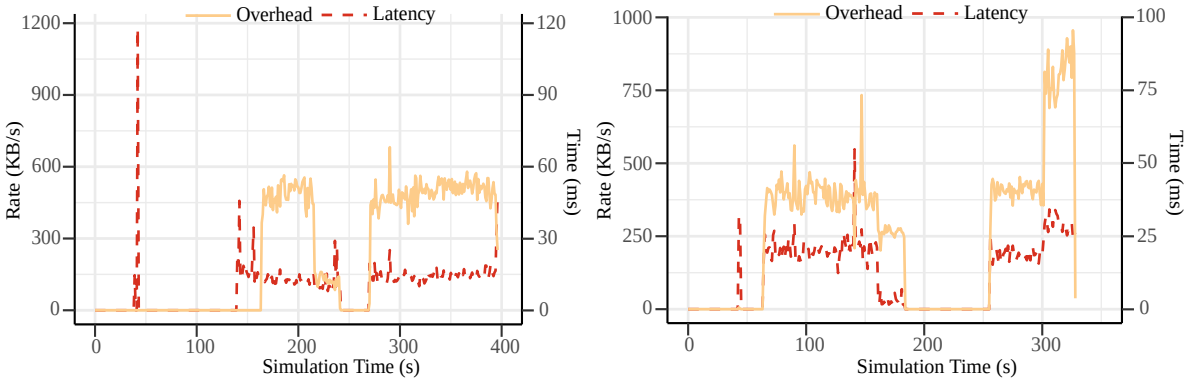
We should note that an interruption in the communication occurs in Fig. 3.8(a) (i.e., at 200 s of the experimental time). This instant corresponds to the GC_{UGR} failure. The connection among GCs is recovered after passing 100 s since that is the initialization time of the GC_{UGR}. In this regard, an upgrade in the hardware capabilities could reduce this time. When the connection is reestablished, the GC_{UPC} shares its network information with the GC_{UGR} to maintain consistency and replicated information in the distributed architecture. In this case, the latency metric does not have the initial peak since the GC_{UPC} already knows its ACs' information and only needs to update the other GC.

To further evaluate inter-controller communication, we analyze a case when one of the ACs

fails. This failure was emulated by turning off the $AC2_{UPC}$ at the end of the test. Therefore, the values of overhead and latency in Fig. 3.8(c) are 0 after passing 350 s of the experimental time. As a consequence of this event, we noticed a spike in the overhead metric around 950 KB/s during the communication among GCs. Similarly, we observed a variation in the GC_{UPC}-$AC1_{UPC}$ communication. More specifically, a small fall in the transmission rate was due to the node reassignment process.

3.5.3 Internet Topology Zoo Scenarios

The second and third scenarios are used to evaluate the DDS App implementation in large networks by analyzing the synchronization delay and overhead among GCs. We selected Abilene and BtNorthAmerica topologies since they have 11 and 36 nodes, respectively. In this way, we allow for several synchronization events as the ACs must manage a significant number of OFSs. Additionally, they must discover more links and install more flows than in the first scenario. Figure 3.9 depicts the results of these experiments.



(a) Abilene network.

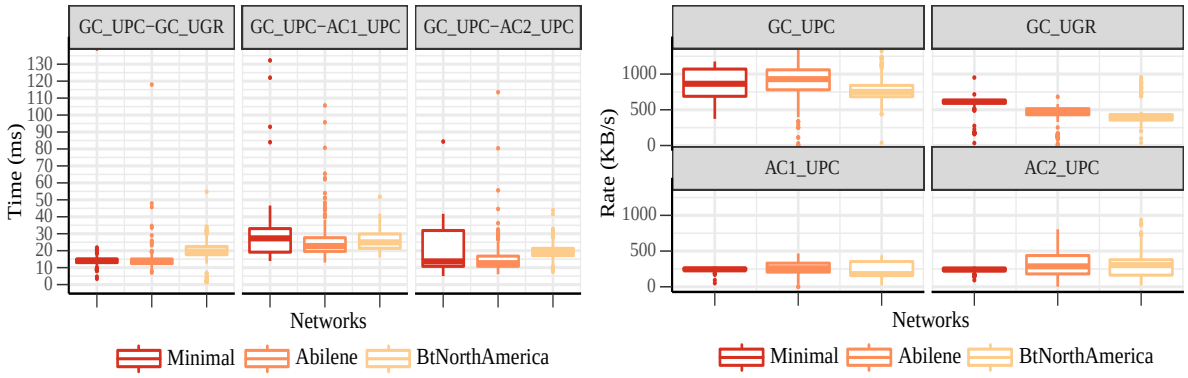
(b) BtNorthAmerica network.

FIGURE 3.9. Synchronization latency and overhead in GC_{UPC}-GC_{UGR} communication.

Similar to the first scenario, higher latency values are obtained during the discovery stage. Overall, the latency values are around 25 ms once the controllers are synchronized. In these experiments, we also emulated a failure in the GC_{UGR}, as reflected in Fig. 3.9, with 0 values of overhead and latency. After restarting GC_{UGR}, the GC_{UPC} updates it on the most recent network information. In addition, we emulated a failure in one of the ACs when running experiments in the BtNorthAmerica network. This failure was illustrated by an increase in the overhead with values up to 950 KB/s, see Fig. 3.9(b). These experiments reflect that the network size does not significantly influence the synchronization latency and overhead related to the GCs communication. It must be noted that the average values of these metrics were similar during most of the experimental time for both scenarios.

3.5.4 Aggregated Results in Analyzed Scenarios

The aggregated results of the performed experiments are depicted in a boxplot representation with a 95% confidence interval. Figure 3.10 depicts the defined metrics for the three studied scenarios. In Fig. 3.10(a), the outliers represent the values of latency related to the discovery phases among controllers.



(a) Synchronization latency.

(b) Synchronization overhead.

FIGURE 3.10. Aggregated results of the 10 experiments per evaluated scenario.

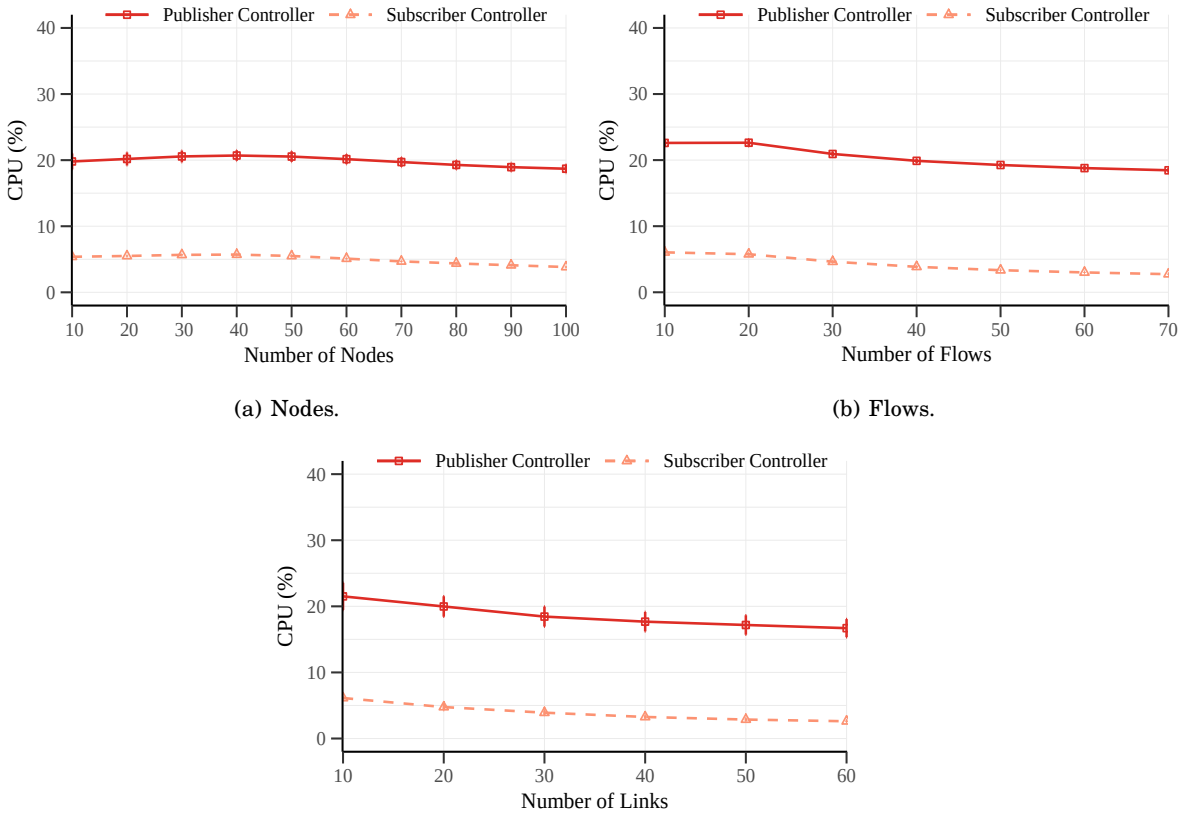
The 75 percentile of the samples regarding the latency values among the GCs requires less than 25 ms for the BtNorthAmerica network and 15 ms for the other networks. Additionally, we obtained higher latency values in the third scenario (i.e., the BtNorthAmerica network) since it generated several events due to the considerable number of nodes. Furthermore, the ACs' processing time affects their synchronization latency with the GC_{UPC}. This behavior is related to the AC's functionality since its crucial role is to configure and manage its assigned nodes. Overall, 75% of the samples require less than 30 ms to synchronize all the events.

We also observed that the GC_{UPC} had the highest synchronization overhead among all the controllers, as shown in Fig. 3.10(b). We expected this result because the GC_{UPC} receives the network information from its ACs and shares it with the GC_{UGR}. Additionally, 75% of the data samples in both ACs require less than 500 KB/s to share the events with the GC_{UPC}.

3.5.5 CPU Consumption Evaluation

This section evaluates the CPU consumption of the controllers when acting as publishers or subscribers. Particularly, we analyze this metric regarding the amount of data that can be sent through the "Topology" topic (i.e., nodes, flows, and links), as shown in Fig. 3.11.

Similar to the previous section, these figures reveal the results of several experiments with a 95% confidence interval. In general, the CPU consumption of a controller that plays the publisher role was around 20%. However, this value had a smooth reduction when the amount of



(a) Nodes.

(b) Flows.

(c) Links.

FIGURE 3.11. CPU consumption for each type of exchanged information.

data increased. Likewise, a subscriber controller displayed similar behavior, although its CPU consumption was approximately 5%. The described performance in publisher and subscriber controllers was aligned with the results obtained in [129–131].

Additionally, the outcome of these experiments reveals that the controller consumes more CPU resources when acting as a publisher than as a subscriber. The differences in this metric between both roles were expected since the publisher controller must read the network information from the controller's data store and create the "Topology" topic before sending any data. Thus, it has an aggregated load compared to the subscriber controller, which saves time and resources because it implements listeners to be aware of new data.

3.6 Conclusion

5G and beyond networks require innovative schemes and solutions that make them scalable and fault-tolerant since most of their use cases pose complex demands. In this vein, this chapter represents our starting point for improving these networks' resilience and fault tolerance. By

proposing a hierarchy of SDN controllers, we offer scalability, flexibility, and programmability to the deployed networks. The proposed architecture is composed of distributed controllers at the top level that manage another group of controllers placed at the bottom level. A publish/subscribe module allows the communication process and network information exchange among controllers through the DDS App.

The three main blocks of the proposed communication mechanism were introduced. A detailed flowchart explained the behavior and execution of each element. Overall, the implemented mechanism allows network information distribution through a configurable mode to announce which data are sent regarding the generated network events, thus achieving a fine-grained sharing approach. In addition, this implementation can auto-discover SDN controllers and update the network information after a controller failure.

To evaluate the performance of the DDS App, we developed a national SDN testbed (Section 3.4) formed by a hierarchical architecture connecting two SDN domains in Barcelona and Granada. Since we used OpenDaylight distribution as an SDN controller, we detailed the relationship of our application with different OpenDaylight modules.

Finally, we emulated three different scenarios in the implemented testbed to evaluate the DDS App according to latency and overhead metrics. The results obtained in terms of latency were low in most cases. Additionally, the CPU consumption of a controller acting as a publisher was higher than when playing the subscriber role. We could conclude that the SDN controllers synchronized network information (e.g., nodes, flows, and links) quickly and lightly through our application.

CHAPTER 4

USE CASE OF A RELIABLE NETWORK SERVICE: SDN-BASED SOLUTION

This chapter is based on:

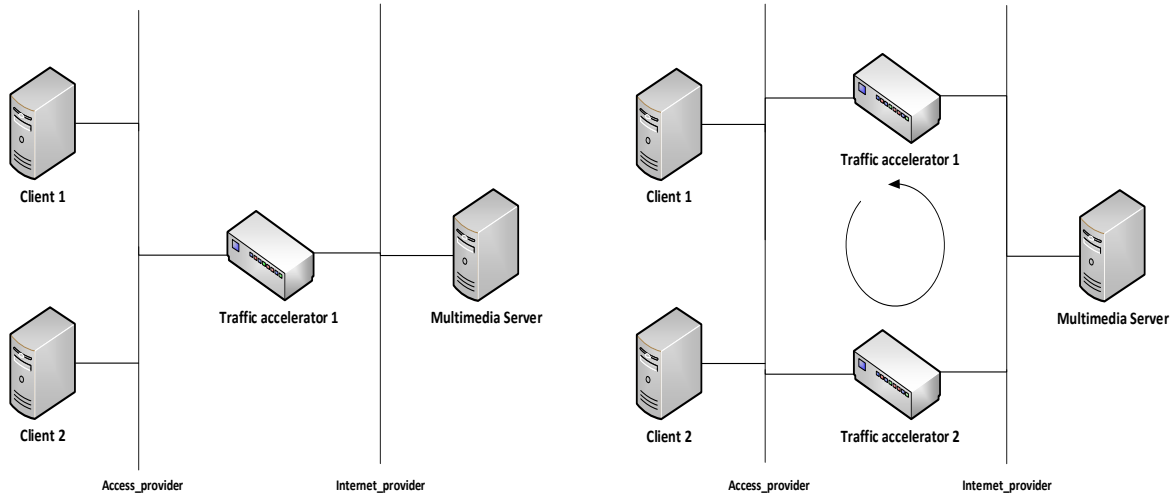
- **A. Llorens-Carrodeguas**, I. Leyva-Pupo, C. Cervelló-Pastor, L. Piñeiro, and S. Siddiqui, "An SDN-Based Solution for Horizontal Auto-Scaling and Load Balancing of Transparent VNF Clusters," *Sensors*, vol. 21, no. 24, Dec 2021.

This chapter presents a specific use case where an SDN-based solution can improve the availability and reliability of a deployed network service. Toward this aim, an SDN controller balances the user's traffic demands among the constituent network functions of the service. It can also scale the assigned resources to avoid underused or overused node resources. Moreover, the implemented DDS application can be used to increase the fault tolerance of the SDN control plane by replicating the network information among the utilized controllers. In this manner, we can ensure resilience at the service and infrastructure levels. Section 4.1 presents a brief overview of the analyzed problem to better understand the studied use case. Subsequently, the proposed solution for the management (i.e., load balancing and horizontal auto-scaling) of a cluster of VNFs is presented in Section 4.2. The presented solution is then validated in Section 4.3 by examining its performance in different situations.

4.1 Problem Description

Network operators are virtualizing their physical network functions to be aligned with the 5G and beyond network use cases. According to the network functions' characteristics, they can be grouped into different categories (i.e., by network functionalities and connectivity, network

security, and network performance). From these groups of functions, the ones associated with network performance (e.g., traffic shaping, rate limiting, and traffic accelerators) are usually transparently placed between the access and internet network providers. More specifically, these network functions do not have a forwarding IP since they are deployed in a BITW manner to avoid altering the communication endpoints. The virtualization concept offers flexibility for the management of network resources as VNFs can be scaled by considering the variations in their processed traffic. In this regard, existing MANO frameworks (e.g., OSM) offer some scaling policies to automatically adjust the VNF's capacity to the traffic demands. However, the scale-out policy of MANO frameworks is characterized by placing new deployed functions in the same subnetwork as the original ones. Additionally, these frameworks do not allow for the modification of previously launched VNFs (e.g., to add new VNF instances or subnetworks). These limitations cause severe problems in the network when scaling transparent functions because network loops may appear. Figure 4.1 illustrates the above problem by analyzing a network service comprised of virtual traffic accelerators.



(a) Single transparent VNF.

(b) Multiple transparent VNFs.

FIGURE 4.1. BITW deployment problem in a NFV scenario.

The initial scenario of the analyzed use case is depicted in Fig. 4.1(a). A single virtual traffic accelerator is deployed in a BITW manner between the access and internet provider networks (i.e., acting as a bridge since the incoming traffic of one interface is sent by the other) to improve the clients' QoE. Eventually, the traffic accelerator's capacity utilization increases with the number of requests, which triggers a scaling-out action when the utilization is above a determined threshold. As a result of the scaling process, the new network scenario is composed of two traffic-accelerator VNFs, as shown in Fig. 4.1(b). It should be noted that two transparent VNFs are connected to the same networks, acting as two bridges and sending packets from one network to the other. Thereby

multicasting or broadcasting messages such as address resolution protocol (ARP) could infinitely loop in the network since there are two active paths between the multimedia server and the clients, thus flooding the network and leading to broadcast storms, multiple packet transmissions, slow and irregular connections as well as network failures.

On the other hand, load balancers are crucial when more than one instance of a given type is deployed in a commodity cluster. Thus, they avoid overload by efficiently distributing traffic to the pool members. Furthermore, load balancers enhance the availability and reliability of the network service by redirecting incoming requests to only healthy VNFs. In this regard, well-known MANO frameworks, such as OSM and ONAP, lack native load-balancing services. Therefore, it is necessary to deploy specific virtual functions (e.g., HAProxy [132]) along with the pool members to provide this feature. Nevertheless, this solution requires a virtual IP in the load balancer to redirect all the incoming traffic to it. This mechanism is recommended when the pool entities are the destination of the incoming traffic. Otherwise, an extra function may need to be placed in the flow path to modify the header packets to reach their final endpoint. This behavior has been found in VIM technologies, such as OpenStack, where its load-balancing service (i.e., Octavia [133]) is also aimed at final services.

Moreover, several types of VNFs (e.g., firewalls and intrusion detection systems) require that the same instance process all the fragments of a given flow during the flow lifetime. Other types of VNFs, like TCP traffic accelerators, may need a bidirectional flow affinity to work correctly.

Under these circumstances, the design of a mechanism to efficiently manage transparent VNF clusters is crucial. The envisioned solution must be able to dynamically scale the cluster's resources. In addition, it must be capable of balancing the flow traffic between transparent VNFs while ensuring bidirectional flow affinity and avoiding packet modifications and extra processing.

4.2 Design Architecture and Implementation

Two redirector VNFs are proposed for the management of transparent clusters. They are placed one at each side of the cluster (uplink and downlink directions) to ensure the bidirectional flow affinity requirement (see Section 4.2.3). Each VNF redirector consists of an OFS and an SDN controller mainly responsible for load-balancing aspects. The SDN controllers are configured to work in an active-standby mode. In this case, the use of two SDN controllers is not strictly necessary, but it is highly recommended to improve the robustness of the solution.

The active controller replicates the network information to the standby one through the DDS application proposed in Section 3.3. Since our application is based on the publish/subscribe paradigm, the active controller will publish the discovered nodes and configured flows as a data stream, also known as the topic (see Section 3.3.1). More specifically, seven fields compose the used topic for this solution: Identifier, NodeId, Port, SourceNode, SourceNodePort, DestinationNode, and DestinationNodePort. The active controller announces whether it will send a node or a flow

using the identifier field. In the case of the standby controller, it will be subscribed to the used topic, thus receiving the network information. This approach avoids the network discovery phase of the standby controller when a failure occurs in the active controller, thus reducing the service downtime and improving the system's fault tolerance.

Since most traffic requests originate in the uplink direction, we placed the master controller facing the access network to reduce the response time. Figure 4.2 provides a general overview of the proposed design as well as the interconnection mode among the solution elements.

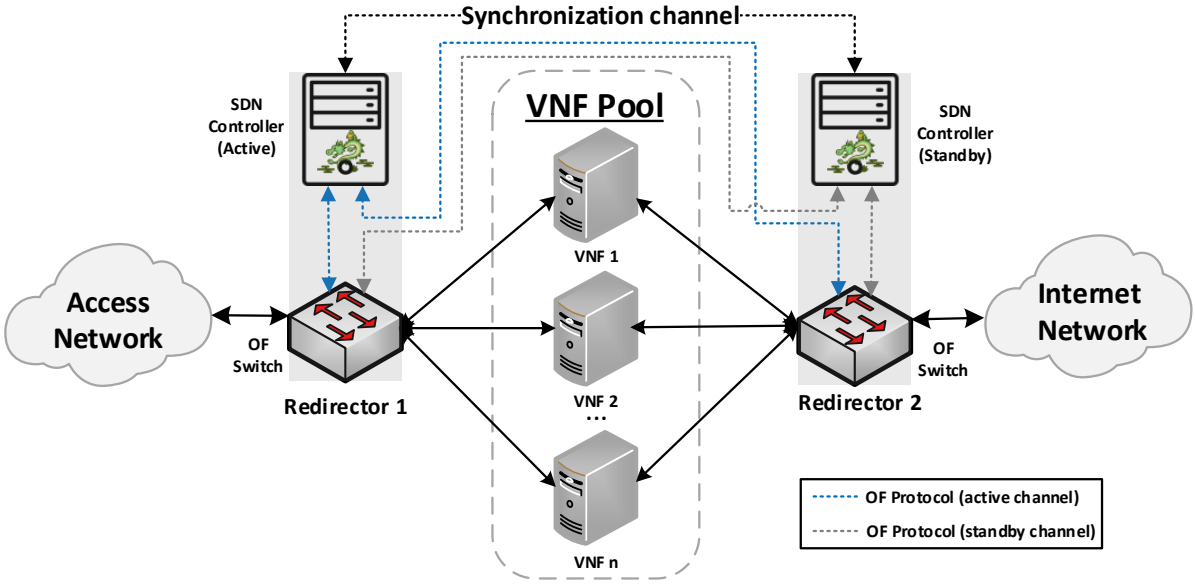


FIGURE 4.2. A high-level view of the proposed solution.

The envisioned VNF redirector application has a modular design (i.e., monitoring, auto-scaling, and load-balancing modules), and the logic of its constituent functions runs on top of the SDN controllers, as shown in Fig. 4.3. The monitoring module is in charge of checking the status of the VNF instances as well as periodically collecting metrics (e.g., CPU consumption and network traffic). The other blocks (i.e., load balancing and auto-scaling) use this information to perform their logic. In the case of the auto-scaling module, it manages the cluster size by triggering scale-in and scale-out actions based on the status and load of the pool members (see Section 4.2.2 for more details). The core of the redirector application is the load-balancing module, which selects the VNF instance to which a traffic request will be assigned for its processing. The selection of the cluster entity depends on the specified load-balancing strategy (e.g., least loaded CPU) and the status and capacity of the cluster's members. Each instance of the cluster is associated with a unique port in the OFSs. We recommend selecting the same port in both switches (a mirror configuration) to simplify the solution complexity.

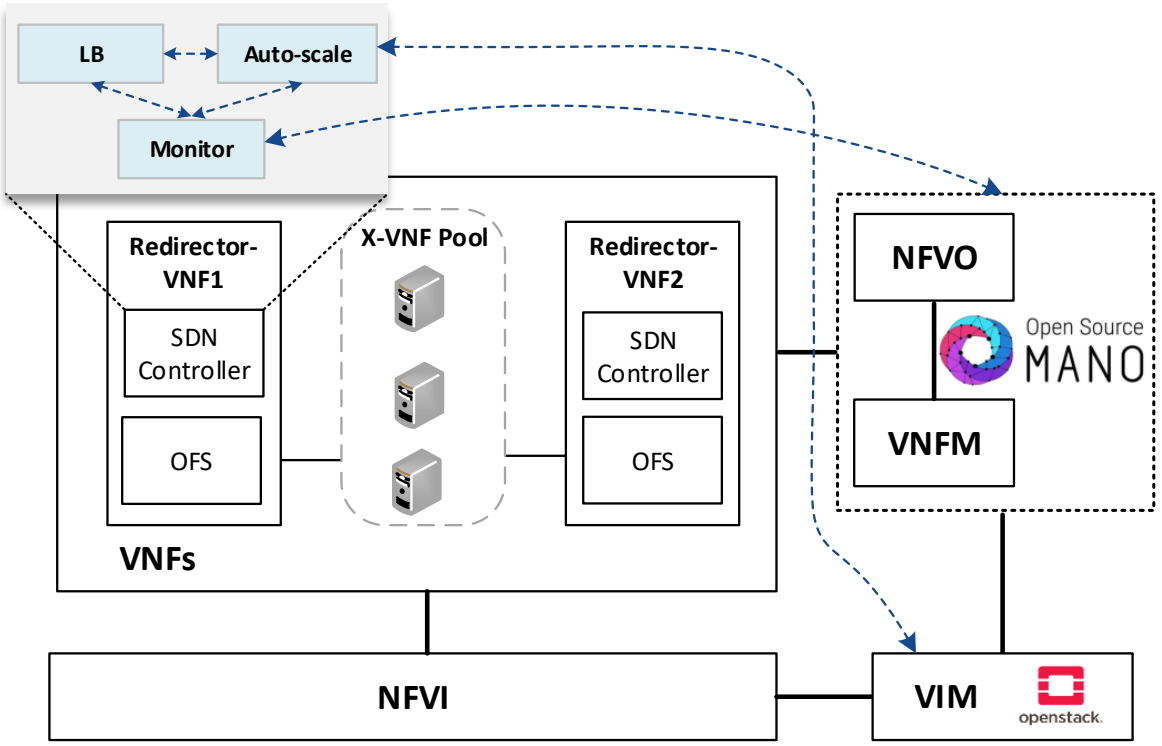


FIGURE 4.3. An overview of the solution components and its communication with the NFV entities.

Finally, our design exploits the parallel processing capacity of multi-cores and a multi-threaded technique to improve the performance of the VNF redirector. This sort of implementation allows faster packet processing in the load-balancing module and avoids delays in its procedure due to the performance of the other modules. The following sections provide more insights into the operation mode of the VNF redirector's modules.

4.2.1 Monitoring Module

Since transparent VNFs work in layer 2 of the Open Systems Interconnect (OSI) model, they are not compatible with well-known health-monitoring methods such as PING, TLS-HELLO, UDP-CONNECT, and TCP-CONNECT because they require directing their queries to an IP address. An approach to overcome this limitation could be using an alternative channel (e.g., a management network) to send the messages associated with one of these health methods. However, this mechanism implies an additional load in the VNF cluster. In this sense, our solution uses information already available to other components (i.e., VNFM) to resolve this shortcoming. The VNFM monitors the health status and performance (e.g., CPU and memory utilization) of VNF instances and network services by managing their lifecycles. The monitoring

module of the OSM framework has a feature (mon-collector) that collects the specified metrics in the VNF descriptors. This item polls the VIM to gather the desired metrics of the deployed VNFs and stores those metrics in its Prometheus time-series database (TSDB).

Our solution implements the monitoring module to make this information available to the redirector application. More specifically, this module communicates with the Prometheus TSDB and gathers the VNF's status and metrics. In this manner, the load-balancing block can be aware of the healthy and unhealthy instances and their available resources. This approach solves the limitation mentioned above and avoids overloading the cluster with frequent health polling through the management network.

This module also gathers information about the deployed network service configuration, in particular, the constituent virtual deployment units (VDUs) of the transparent cluster and their subnetworks. This information is used to match the VDUs with the OFS interfaces to which they are connected. These interfaces are discovered upon the OFS registration on the SDN controllers. In this sense, our monitoring module helps the redirector application discover the network topology.

Additionally, our monitor warns the load-balancing block when it detects an unhealthy instance. Thus, the load balancer triggers a flow-rule updating process that deletes the OF rules associated with the unhealthy instance in the switches table. By sending a delete flow message to the OFS, the controller instructs them to remove the flow rules that contain the specified port (i.e., the one connected with the unhealthy instance). Consequently, the monitoring module avoids network disruption since the load balancer can select another healthy transparent VNF to process the incoming traffic.

4.2.2 Auto-Scaling Module

The automatic scaling of BITW VNFs imposes several challenges to avoid network loops. More specifically, in the case of deploying more than one transparent VNF in the same subnetwork, loops appear given that those instances work as a "wire" in the network. Therefore, transparent VNFs should be deployed on different subnetworks to avoid this behavior. Thus, we can redirect flows to a specific instance since they do not dispose of routing information. However, this approach is not currently possible since MANO frameworks do not allow either specifying different subnetworks when executing auto-scaling policies or adding new VNF instances and subnetworks to already instantiated network services.

Consequently, we diverge from the assumption that the VNF cluster has already been dimensioned to resolve these challenges. Namely, the cluster is launched with each member connecting with one interface in the OFS. In this regard, the OFS interfaces must also be dimensioned according to the pool size. Nevertheless, only the minimum required number of instances remains active to save energy and computing resources; the rest are on standby. Thus, the primary function of this block is to manage the status of the instances (i.e., activate or

deactivate instances) and not to create or remove them. To this end, the auto-scaling module must interact with the VIM. In the case of OpenStack, its Nova service [134] offers several options to manage VNFs (e.g., stop or start, suspend or resume, shelve or unshelve, pause or unpause, and resize), which have different shortcomings in terms of system's resource utilization and the VNF's activation time.

The auto-scaling logic runs during the system lifetime and updates the scaling metric values at each defined period. For the implementation of this logic and to guarantee the proper performance of this module, we define the following parameters:

- Scaling metric: the metric to be monitored (e.g., CPU or memory) and upon which scaling actions will be taken
- Aggregation type: refers to how the scaling metric is gathered (e.g., average or maximum values)
- |VNF|_max: maximum number of active instances in the cluster
- |VNF|_min: minimum number of active instances in the cluster
- Thresholds: upper and lower bounds of the selected metric upon which scale-out or scale-in actions are triggered, respectively
- Threshold time: a minimum amount of time during which the state of the scaling metric concerning the threshold values must sustain to trigger a scaling event (Different threshold times can be defined for scale-in and scale-out actions.)
- Cooldown time: the minimum amount of time that the system must wait after triggering an event before activating another

Figure 4.4 illustrates the programming logic of the auto-scaling block. It starts by setting the minimum number of active members and exposing this updated information to the load balancer. Next, it proceeds to collect samples (i.e., by communicating with the monitor block) of the specified metric until completing a threshold time and computes its aggregation type respecting the set of active members in the VNF pool. Afterward, the auto-scaling conditions are verified by comparing the obtained value with the maximum and minimum thresholds. When meeting one of these conditions, this module confirms the number of active instances before proceeding with the activation or deactivation of an instance. This action helps maintain the number of active instances between the established values. In this context, the scaling procedure is discarded when the scaling action violates any threshold values; thus, the system continues gathering samples.

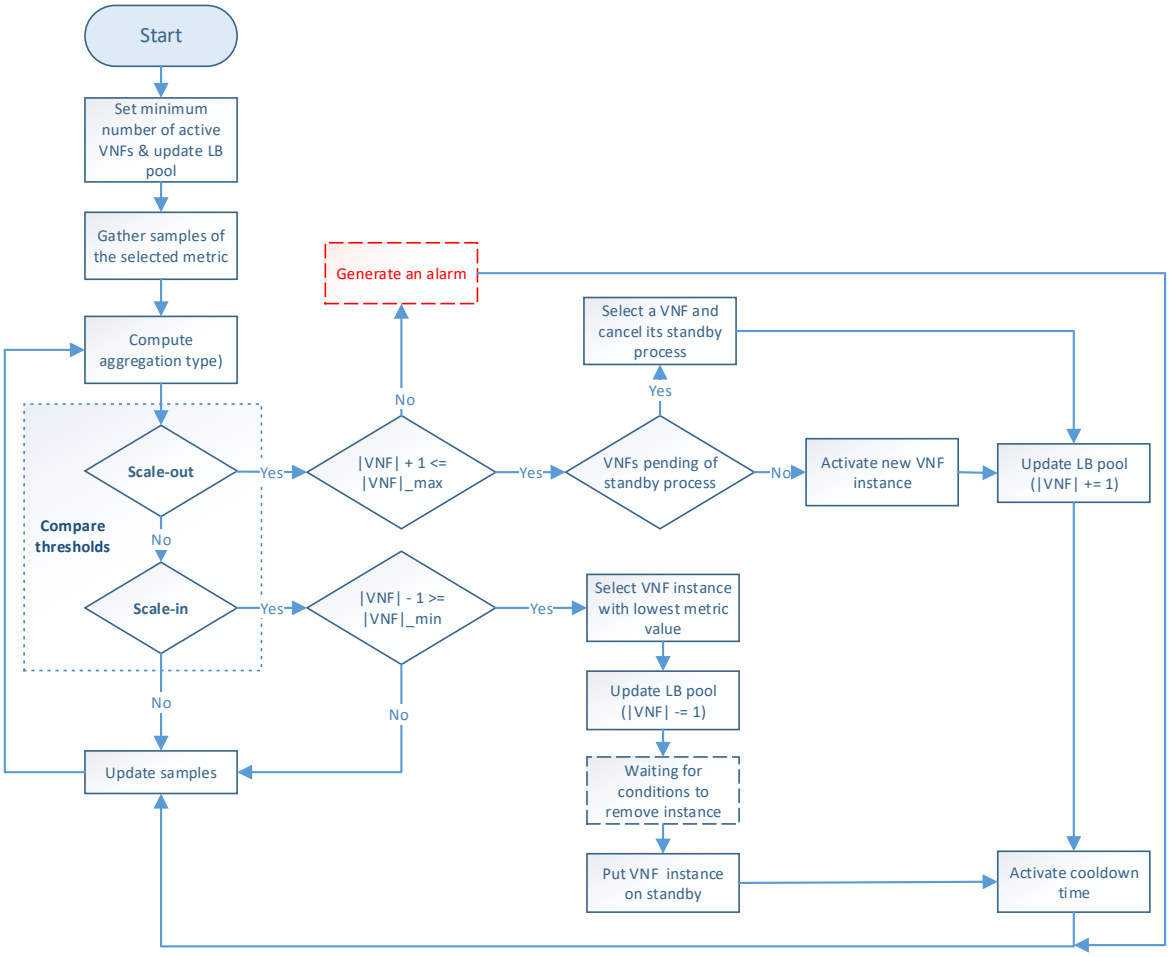


FIGURE 4.4. Flowchart of the proposed auto-scaling procedure.

In addition, an alarm can be activated to notify the system administrator that further actions are needed if the scaling procedure is omitted due to reaching the maximum number of active instances. Otherwise, new instances can be started or existing ones deactivated depending on the triggered condition. According to the VNF service type, we may require a waiting time before putting an active instance on standby to prevent service degradation. When applying scale-out actions, those VNFs in the pre-standby process are selected instead of activating new ones, thus canceling their associated standby process. The set of available VNF instances is reported to the load-balancing module after each scaling decision. We must note that the active members of the cluster are immediately updated for scale-in actions to avoid the assignment of new flows to an instance in a standby process. Finally, a cooldown timer is activated to prevent unnecessary scaling actions caused by possible system instability, and the set of gathered samples is updated.

4.2.3 Load-Balancing Module

The literature review provides two options for maintaining flow affinity when balancing traffic. One approach uses a dedicated load-balancing algorithm based on an IP hashing method. The other utilizes a stick table in memory and a non-deterministic load-balancing algorithm (e.g., round robin or least connections). Load-balancing algorithms are suitable as long as the number of instances involved does not change. Otherwise, more complex techniques (e.g., consistent hashing) must be applied to diminish the adverse effects on the flow affinity mapping when the number of instances changes. This method is recommended for load-balancing applications that do not require synchronization among load-balancer entities but still use the same hashing function.

On the other hand, the second approach requires a global view of the system or synchronization among the load balancers to ensure bidirectional affinity. In this case, no session is redirected when a new instance is added to the cluster, representing an advantage regarding the first approach. Our solution is based on the second technique since we guarantee the flow affinity upon flow entries registered in the OFS tables. Furthermore, the proposed solution does not require any exchange of information between the load-balancing entities (i.e., switch tables) since the flow rules for given traffic are simultaneously created in both switches by their master controller, which has a global view of the cluster.

A drawback of working with flow tables is that the flow waiting time may increase when the traffic is significant. Nevertheless, this effect can be diminished by implementing efficient mechanisms to manage switch tables [135, 136]. Old flow entries related to expired flows can be removed to avoid overloaded flow tables. A configuration recommended in this process is to use the idle_timeout field in OF FLOW_MOD messages. This parameter sets a waiting time before the switch removes idle flow entries when no packet has matched within the specified time. Toward this aim, the OFS must track the arrival time of the last packet associated with the flow.

To better understand the flow rules creation, Fig. 4.5 describes the communication process between the OFSs and the master controller. More specifically, when an incoming flow arrives at a switch (step 1), it searches for the existence of a rule matching the flow in its tables (step 2). If this is the case, the switch applies the actions associated with the flow (step 7). Otherwise, the OFS sends the packet to its assigned master controller through an OF PACKET_IN message (step 3). The controller is responsible for extracting the header packet information and determining the actions that must be applied to the flow (step 4).

Figure 4.6 describes the logic executed in the SDN controller when it receives an OF PACKET_IN message (i.e., step 4 in Fig. 4.5). The controller begins by reading the packet information (e.g., headers and OFS's in_port) to determine how the flow is processed since there are two possible options according to the packet's in_port in the switch. More specifically, when OFS's in_port connects with a cluster member, an OF rule is configured to redirect the traffic from the cluster member port to the one connecting the access provider network. This rule is

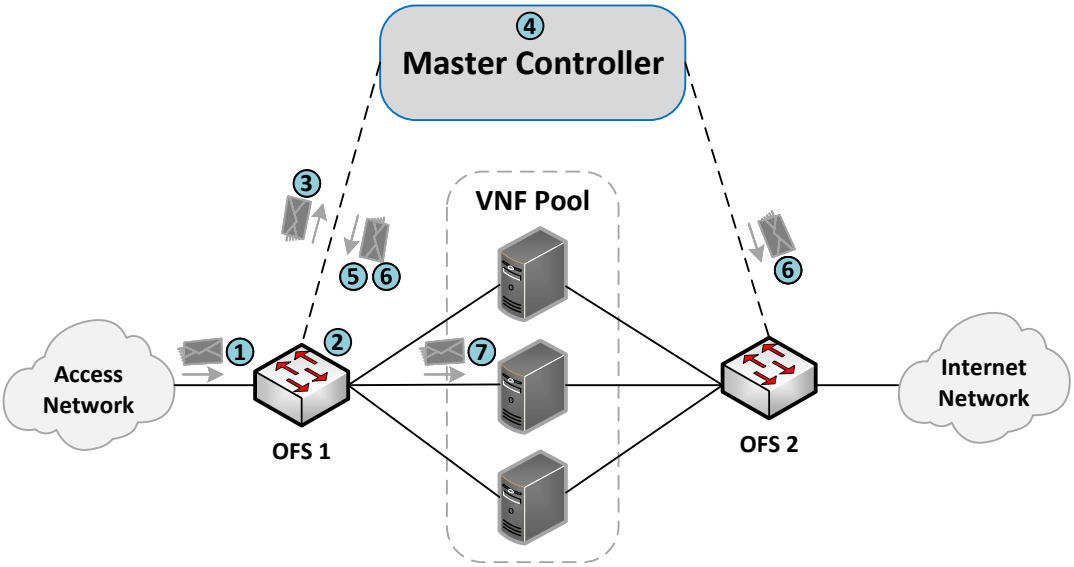


FIGURE 4.5. Communication process between the master controller and the OFSs.

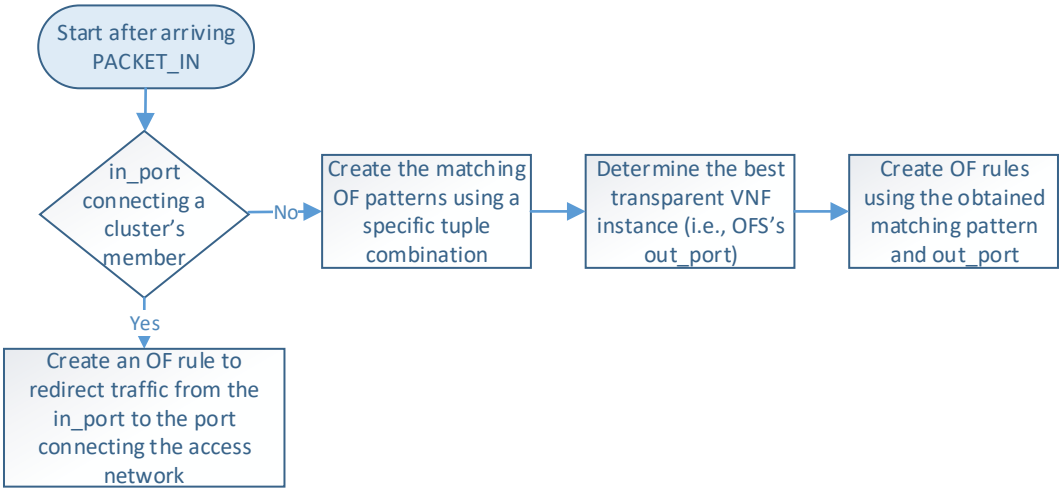


FIGURE 4.6. Flowchart of the proposed load-balancing procedure.

created only once during the operation phase unless the idle_timeout field is specified.

On the other hand, the controller configures actions based on the packet's header information when the in_port connects to the access provider network. In this regard, the load-balancing module creates an OF matching pattern by using a specific tuple combination (e.g., source IP, destination IP, protocol, source port, destination port). Several flow treatments can be defined according to the different tuple combinations and the values of their parameters. Afterward, the load balancer determines the switch's out_port by selecting the best VNF instance according to the specified load-balancing strategy. Then the controller creates a flow rule with the selected matching pattern and the obtained out_port. It should be noted that each transparent VNF

instance connects to a specific switch's port.

After determining the OF rule, the controller sends OF PACKET_OUT and OF FLOW_MOD messages to the switch (steps 5 and 6 in Fig. 4.5). These steps instruct the switch to send the packet through the selected port and configure the specified rule in its table. In addition, a reverse flow rule is proactively configured in the opposite switch by swapping the source and destination fields in the matching pattern and specifying the same action (i.e., OFS's out_port). Following this strategy, our load-balancing module ensures bidirectional flow affinity since it simultaneously creates flow entries in both switches.

Currently, the load-balancing module has three strategies available to distribute traffic among the cluster's members: random, round robin, and least loaded [52, 137]. With the random approach, the controller randomly chooses a VNF instance to redirect incoming flows. This strategy is the most straightforward since the controller must only be aware of the active members in the pool. On the other hand, round robin is one of the most popular load-balancing algorithms due to its simplicity. It distributes incoming traffic requests by sequentially selecting an active pool member to process the incoming traffic. In other words, the controller selects a different member each time a new OF PACKET_IN message arrives. Finally, the least loaded algorithm uses the monitored metrics (e.g., CPU or memory) of the active VNF instances to select the one with minimum load to process new flows.

4.3 Evaluation and Results

This section aims to validate our proposal as a feasible approach to manage traffic distribution and assigned resources according to dynamic traffic demands in a cluster of transparent VNFs.

4.3.1 Experimental Setup

We employed a simplified scenario for the validation of the proposed solution. Specifically, a network service was composed of a VNF cluster, two redirector VNFs, four traffic generators (TGs), and one network. By configuring the VNF interfaces as a bridge, we created a cluster formed by four generic transparently deployed VNFs. The TGs' primary function was to inject traffic between the access and data networks. This traffic had to traverse the cluster in a distributed manner. Additionally, we divided the network domain into smaller subnetworks to deploy each group member on a different subnetwork. In this manner, we ensured that traffic passed only through a single transparent VNF. Moreover, we guaranteed end-to-end (E2E) connectivity between TGs located on different sides of the group. Since the subnetting adoption can introduce several loops in the network when working with transparent VNFs, we automated the network service deployment to enable the Spanning Tree Protocol (STP) in the redirectors and cluster instances to avoid network outages due to loops. Once the master controller had taken over the network control, the STP was deactivated to enable interfaces with blocked status.

Figures 4.7 and 4.8 illustrate the network service topology in a simplified overview and in an OpenStack view, respectively.

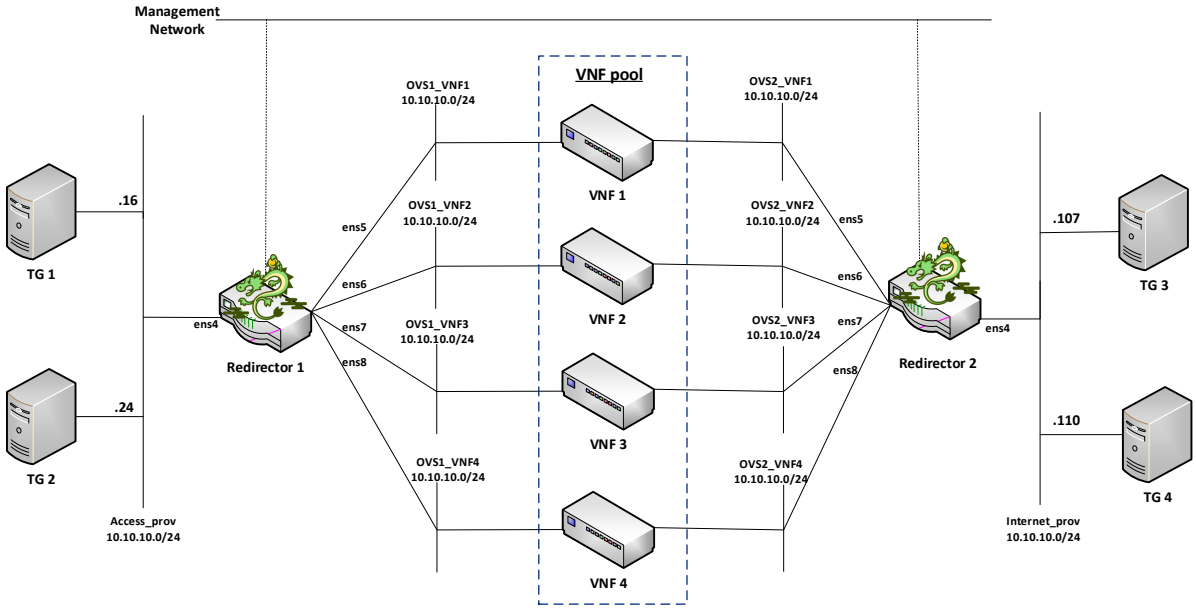


FIGURE 4.7. Simplified overview of the network service topology.

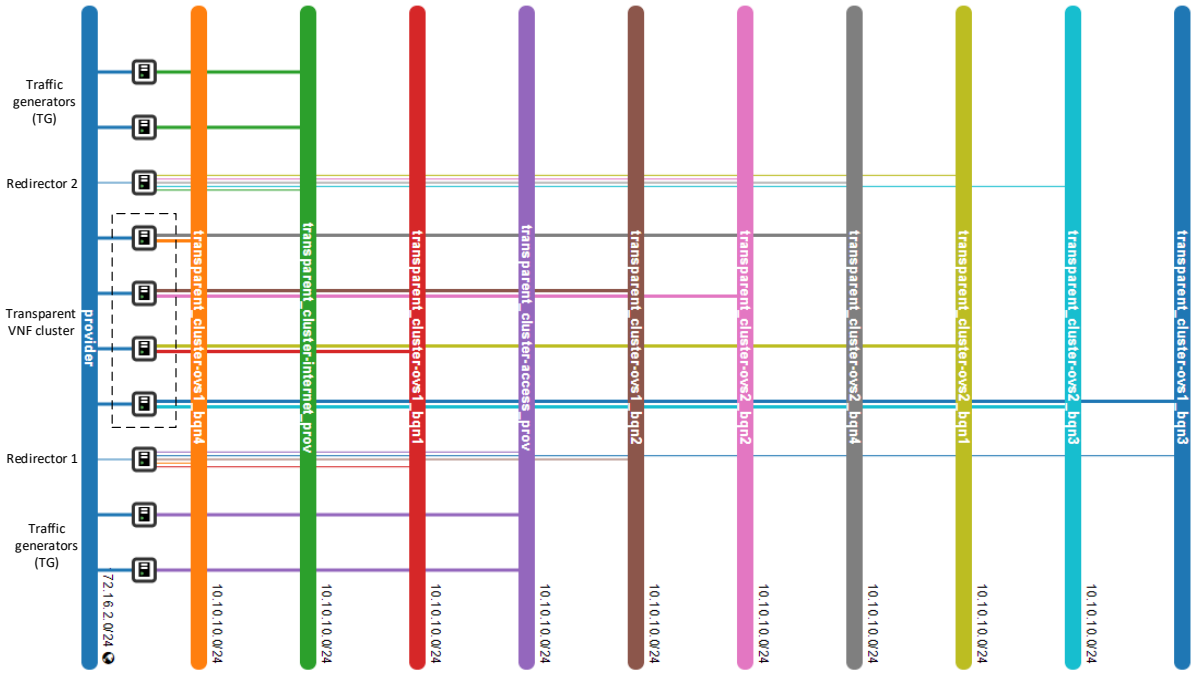


FIGURE 4.8. Network topology overview of the test scenario in OpenStack.

For the implementation phase, we selected Ryu [28] as the SDN controller due to its simple configuration and Python compatibility. The latter was necessary to integrate the redirector application with the OSM client (i.e., version 8.0.4) to gather information about VNF instances through the OSM framework. It should be noted that by using a different SDN controller than the one presented in Section 3.4.1, we illustrate the interoperability of the proposed DDS application concerning the controllers' framework and programming language. Furthermore, we used Open vSwitch v.2.15.1 as OFSs, which were configured to communicate with the controller in an out-of-band mode. It was selected because we used a separate network (i.e., the management network) to connect forwarding devices to the controller and exchange control traffic. Table 4.1 summarizes the hardware and software specifications (version and properties) of the testbed.

Table 4.1: Hardware and software specifications.

	Description	**Specifications**
Server 1	NFVI with OpenStack Controller	Memory: 32 GB RAM DDR4 Processor: Intel Core i7-5820K CPU @ 3.30 GHz OS: Ubuntu Server 18.04.2
Server 2	NFVI with OpenStack Compute	Memory: 16 GB RAM DDR4 Processor: Intel Core i7-5820K CPU @ 3.30 GHz OS: Ubuntu Server 18.04.2
OSM host	HP Compaq 8100 Elite SFF PC	Memory: 8 GB RAM DDR3 Processor: Intel Core i5 CPU 650 @ 3.20 GHz OS: Ubuntu Desktop 18.04.2
OpenStack		Train
OSM		Release EIGHT

4.3.2 Solution Validation

As a proof of concept, we ran several experiments to verify the correct operation of each aspect of the proposed solution. We started by checking the connectivity between the different elements of the system. Figure 4.9 depicts the results of the ping tests between TGs located at access and internet sides. The ping execution was successful since it showed stable E2E traffic (no network loops) through the cluster and low values of time response. The results showed that our solution was working as expected because the flow rules configured by the load-balancing module guaranteed E2E traffic without losing any packets. This test also showed that the monitoring module had updated information about the active cluster members as the load-balancing module selected a healthy instance to configure the flows in the OFSs.

4.3.2.1 Bidirectional Flow Affinity

Since we consider the assurance of bidirectional flow affinity as an essential requirement in our solution, we generated traffic between the access and internet TGs to validate this feature. Toward this aim, the redirector application was configured to create ARP and IP traffic flow

```
ubuntu@injector-vnf:~$ ping 10.10.10.107 -I ens4 -c 5
PING 10.10.10.107 (10.10.10.107) from 10.10.10.16 ens4: 56(84) bytes of data.
64 bytes from 10.10.10.107: icmp_seq=1 ttl=64 time=1.20 ms
64 bytes from 10.10.10.107: icmp_seq=2 ttl=64 time=1.23 ms
64 bytes from 10.10.10.107: icmp_seq=3 ttl=64 time=1.12 ms
64 bytes from 10.10.10.107: icmp_seq=4 ttl=64 time=0.793 ms
64 bytes from 10.10.10.107: icmp_seq=5 ttl=64 time=1.27 ms

--- 10.10.10.107 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 0.793/1.124/1.276/0.178 ms
```

(a) Ping test between TG 1 (IP: 10.10.10.16) and TG 3 (IP: 10.10.10.107).

```
ubuntu@injector-vnf:~$ ping 10.10.10.110 -I ens4 -c 5
PING 10.10.10.110 (10.10.10.110) from 10.10.10.24 ens4: 56(84) bytes of data.
64 bytes from 10.10.10.110: icmp_seq=1 ttl=64 time=2.25 ms
64 bytes from 10.10.10.110: icmp_seq=2 ttl=64 time=1.15 ms
64 bytes from 10.10.10.110: icmp_seq=3 ttl=64 time=1.32 ms
64 bytes from 10.10.10.110: icmp_seq=4 ttl=64 time=0.860 ms
64 bytes from 10.10.10.110: icmp_seq=5 ttl=64 time=1.24 ms

--- 10.10.10.110 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 0.860/1.368/2.253/0.470 ms
```

(b) Ping test between TG 2 (IP: 10.10.10.24) and TG 4 (IP: 10.10.10.110).

FIGURE 4.9. Command line output for ping tests between TGs located in different subnetworks.

rules based on their source and destination addresses. Initially, the switches only had flow rules associated with the interfaces connecting with the transparent VNFs. Figure 4.10 depicts the dynamic configuration of the flow tables in the OFSs.

Additionally, the switches also had a default rule configured when added to the SDN controller. Specifically, this rule indicates to the OFSs that send to the controller any packet that did not match any entries in their flow tables. Consequently, new rules are created according to the matching criteria when new traffic goes through the switches. As shown in Fig. 4.10, the presence of bidirectional flow affinity can be noticed by analyzing both flow tables. It is evidenced that each pair of source-destination addresses was configured to go through the same port (VNF instance) in both switches. Thus, our solution guarantees bidirectional flow affinity by simultaneously creating the rules in the access and internet side switches during the arrival of OF PACKET_IN events in the SDN controller.

4.3.2.2 Load Balancing

We selected a load-balancing strategy based on CPU utilization to evaluate the traffic distribution among the transparent cluster's members (i.e., least loaded). For this test, we generated 20 TCP flows with a bandwidth of 1 Mbps, a duration of 2100 s, and a waiting time of 90 s between each flow. To generate traffic, we implemented a script based on iperf3 [138] to ensure several connections between a pair of TGs located in different subnetworks. In addition, a new

```
ubuntu@vdu-redirector:~$ sudo ovs-ofctl dump-flows br1
 cookie=0x0, duration=587.972s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=583.762s, table=0, n_packets=3, n_bytes=161, priority=500,in_port=ens8 actions=output:ens4
 cookie=0x0, duration=583.362s, table=0, n_packets=349, n_bytes=20200, priority=500,in_port=ens6 actions=output:ens4
 cookie=0x0, duration=481.946s, table=0, n_packets=0, n_bytes=0, priority=500,in_port=ens7 actions=output:ens4
 cookie=0x0, duration=103.395s, table=0, n_packets=0, n_bytes=0, priority=500,in_port=ens5 actions=output:ens4
 cookie=0x0, duration=471.044s, table=0, n_packets=2, n_bytes=84, priority=500,arp,arp_spa=10.10.10.24,arp_tpa=10.10.10.110 actions=output:ens8
 cookie=0x0, duration=470.995s, table=0, n_packets=14, n_bytes=1372, priority=500,ip,nw_src=10.10.10.24,nw_dst=10.10.10.110 actions=output:ens8
 cookie=0x0, duration=389.114s, table=0, n_packets=1, n_bytes=42, priority=500,arp,arp_spa=10.10.10.16,arp_tpa=10.10.10.110 actions=output:ens6
 cookie=0x0, duration=388.998s, table=0, n_packets=9, n_bytes=882, priority=500,ip,nw_src=10.10.10.16,nw_dst=10.10.10.110 actions=output:ens6
 cookie=0x0, duration=228.446s, table=0, n_packets=1, n_bytes=42, priority=500,arp,arp_spa=10.10.10.24,arp_tpa=10.10.10.107 actions=output:ens6
 cookie=0x0, duration=228.130s, table=0, n_packets=9, n_bytes=882, priority=500,ip,nw_src=10.10.10.24,nw_dst=10.10.10.107 actions=output:ens6
 cookie=0x0, duration=206.730s, table=0, n_packets=1, n_bytes=42, priority=500,arp,arp_spa=10.10.10.16,arp_tpa=10.10.10.107 actions=output:ens8
 cookie=0x0, duration=206.581s, table=0, n_packets=9, n_bytes=882, priority=500,ip,nw_src=10.10.10.16,nw_dst=10.10.10.107 actions=output:ens8
 cookie=0x0, duration=587.972s, table=0, n_packets=11, n_bytes=734, priority=0 actions=CONTROLLER:65535
```

(a) Flow table of OFS redirector 1.

```
ubuntu@vdu-redirector:~$ sudo ovs-ofctl dump-flows br2
 cookie=0x0, duration=596.748s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=588.712s, table=0, n_packets=297, n_bytes=15462, priority=500,in_port=ens8 actions=output:ens4
 cookie=0x0, duration=588.304s, table=0, n_packets=349, n_bytes=20164, priority=500,in_port=ens6 actions=output:ens4
 cookie=0x0, duration=486.884s, table=0, n_packets=1, n_bytes=70, priority=500,in_port=ens7 actions=output:ens4
 cookie=0x0, duration=108.333s, table=0, n_packets=0, n_bytes=0, priority=500,in_port=ens5 actions=output:ens4
 cookie=0x0, duration=475.983s, table=0, n_packets=2, n_bytes=84, priority=500,arp,arp_spa=10.10.10.110,arp_tpa=10.10.10.24 actions=output:ens8
 cookie=0x0, duration=475.934s, table=0, n_packets=14, n_bytes=1372, priority=500,ip,nw_src=10.10.10.110,nw_dst=10.10.10.24 actions=output:ens8
 cookie=0x0, duration=394.052s, table=0, n_packets=2, n_bytes=84, priority=500,arp,arp_spa=10.10.10.110,arp_tpa=10.10.10.16 actions=output:ens6
 cookie=0x0, duration=393.936s, table=0, n_packets=10, n_bytes=980, priority=500,ip,nw_src=10.10.10.110,nw_dst=10.10.10.16 actions=output:ens6
 cookie=0x0, duration=233.384s, table=0, n_packets=2, n_bytes=84, priority=500,arp,arp_spa=10.10.10.107,arp_tpa=10.10.10.24 actions=output:ens6
 cookie=0x0, duration=233.068s, table=0, n_packets=10, n_bytes=980, priority=500,ip,nw_src=10.10.10.107,nw_dst=10.10.10.24 actions=output:ens6
 cookie=0x0, duration=211.667s, table=0, n_packets=1, n_bytes=42, priority=500,arp,arp_spa=10.10.10.107,arp_tpa=10.10.10.16 actions=output:ens8
 cookie=0x0, duration=211.518s, table=0, n_packets=9, n_bytes=882, priority=500,ip,nw_src=10.10.10.107,nw_dst=10.10.10.16 actions=output:ens8
 cookie=0x0, duration=596.752s, table=0, n_packets=7, n_bytes=454, priority=0 actions=CONTROLLER:65535
```

(b) Flow table of OFS redirector 2.

FIGURE 4.10. Dynamic flow rule creation in the OFS redirectors with bidirectional flow affinity.

matching criterion was configured in the SDN controller to redirect traffic based on the five-tuple parameters.

Figure 4.11 summarizes the CPU load of each member of the transparent cluster. By analyzing the CPU utilization of the two active VNF instances (i.e., blue and orange lines), it can be noted that their values increased and decreased in a balanced manner according to the number of active flows passing through them. The blue and orange VNFs had average values of 1.01% and 1.11%, respectively. Thus, the average imbalance achieved by the system was below 10%. This difference was mainly caused by an odd number of flows in the system. Similarly, it was reduced by assigning the successive incoming flows to the least-loaded VNF instance.

4.3.2.3 Auto-Scaling Actions

The capability to increase or decrease the network functions according to specific metrics is one of the main advantages that virtualization offers to network operators. This section evaluates our proposed solution by studying its behavior during different scaling actions. Similar to the previous experiments, the transparent cluster started with two active members, and the rest were on standby. The TGs in the access side generated 50 TCP flows with a bandwidth of 20 Mbps, a connection time of 15000 s, and a waiting time of 300 s between each flow. Thus, we confirmed there would be enough flows to increase the CPU load of the active cluster's members, thereby triggering the scaling actions.

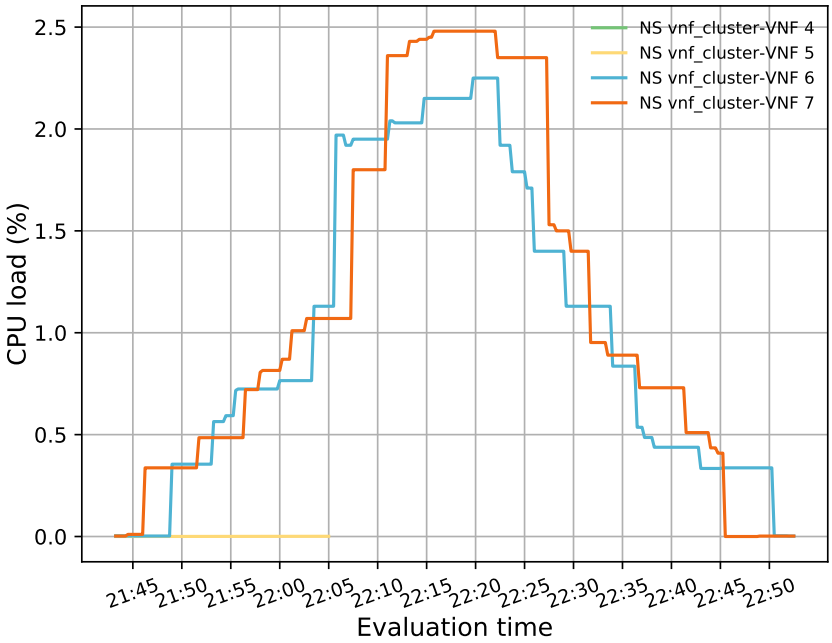


FIGURE 4.11. CPU load distribution among members in the transparent VNF cluster.

Figure 4.12 illustrates the evolution of the test over time. This experiment established minimum and maximum thresholds of 20% and 50% in the average CPU utilization to trigger scale-in and scale-out actions. We also considered a threshold time of 120 s and a cooldown time of 300 s. A total of four scaling actions were executed (i.e., two scale-out and two scale-in) during this test.

At the beginning of the experiment, only two instances were active (i.e., blue and orange lines) with very low utilization. However, the CPU load increased since new packets were injected into the system as time passed. At around 22:40 h, a scale-out action was triggered as the average CPU load was above 50%. As a result of this procedure, a new VNF instance was activated (green line) in the first moment. Meanwhile, a second instance (yellow line) was also started some minutes later (after 23:00 h) since the average CPU load of the active members once again passed the maximum established threshold. We noted that the aggregation metric associated with the new instances took some time to be reflected in the Prometheus TSDB. This behavior is a consequence of OpenStack's granularity (i.e., 300 s). Therefore, the starting value of their CPU usage is different from 0% because they were already processing traffic.

During the second half of the experiment, the oldest flows assigned to the initially deployed VNFs, represented in blue and orange lines, started reaching their specified lifetime. A decrease in the load of the initial instances was noticed due to the termination of these flows. Similarly, the two last activated instances (i.e., green and yellow lines) underwent a reduction in their loads around 03:45 h. The average CPU load in the transparent cluster started decreasing until the minimum established threshold was reached. At this point, a scale-in action was performed. As

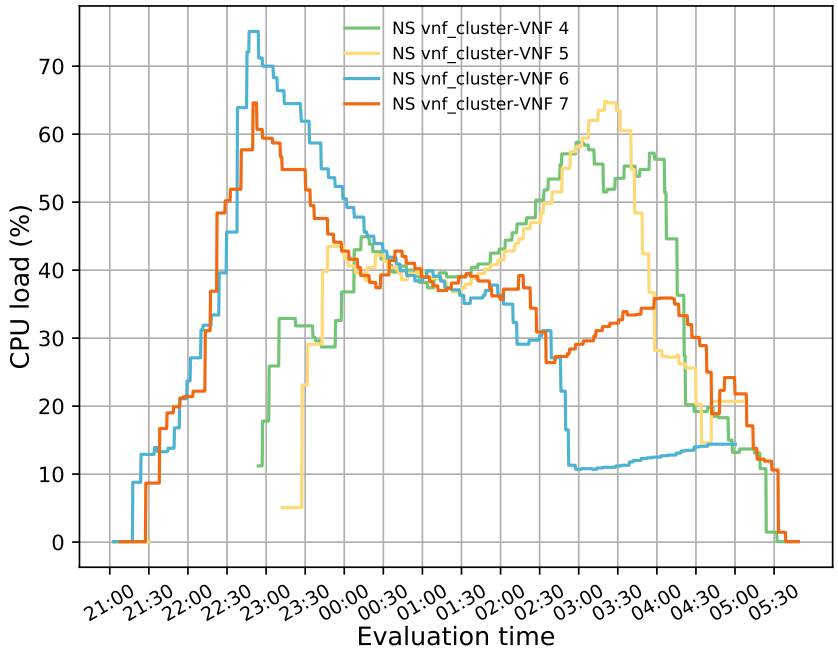


FIGURE 4.12. CPU load distribution among members in the transparent VNF cluster while auto-scaling actions were performed.

a result, the instance represented by the blue line was deactivated, and its existing flows were reassigned to the other VNFs. Another scale-in action was triggered after the defined cooldown time because the average CPU usage was still below 20%. Thus, the transparent VNF depicted in yellow was deactivated.

The description above is summarized in Fig. 4.13, which depicts the variation of the average CPU load in the cluster and its number of active members during the experiment. This figure shows that a new instance was activated to decrease this parameter every time the average CPU was above the maximum threshold (i.e., 50%). In contrast, in the moments when the average CPU was below 20%, the number of cluster instances was reduced until reaching the minimum number of active VNFs, expressly, by the end of the experiment. Here it should be noted that the number of active instances remained constant during the experiment while the average CPU metric was within the established thresholds.

During the execution of this experiment, there was a reduction in the CPU load of the initially deployed VNFs when new instances were activated, even though their allocated flows were invariably maintained. This behavior was due to an increase in the system congestion levels, which the establishment of new connections caused. In this regard, TCP implements a congestion window (CWND) to avoid congestive collapse. This mechanism limits the number of outstanding unacknowledged packets in transit on the network associated with a pair of source and destination endpoints. More specifically, when the network traffic increases, each client injecting TCP packets increases its CWND to reduce its forwarding rate [139]. Since the

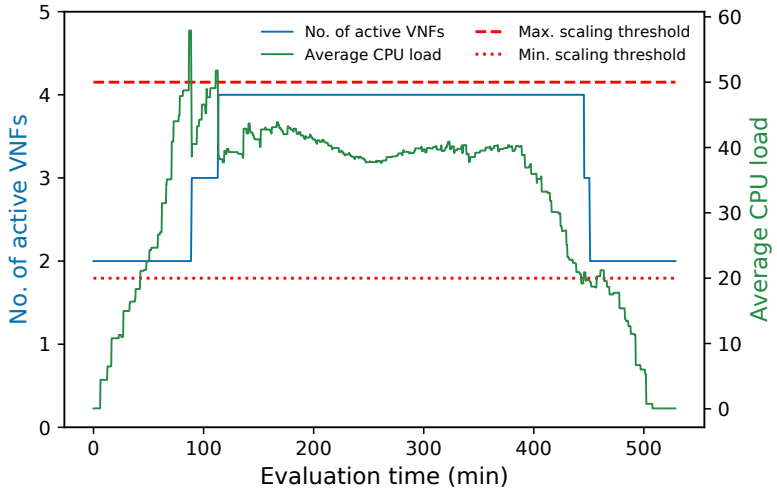


FIGURE 4.13. Number of active members in the cluster and average CPU utilization versus time.

transmission of TCP streams consumes most of the CPU resources [140], the aforementioned reductions influence the CPU load of the BITW VNFs. Therefore, the newly arrived flows were assigned to the last activated instances because they were the least loaded. Thus, the CPU load variation of the instances was mainly due to the arrival of new flows and not to the connections already established.

Figure 4.12 shows how in the middle of the test (i.e., 00:30–01:30), the CPU load of the cluster's members was more balanced, although it had slight variations because the incoming traffic was evenly distributed across all the instances with similar utilization levels. However, this behavior did not last too long. The CPU load of the last activated instance increased, while that of the initial VNFs decreased. With this action, the CWND of the active connections was reduced while increasing their send rates, resulting in a higher CPU load on scaled instances. The latter was not noticeable in the first pair of deployed instances due to completing most of its allocated sessions.

4.3.2.4 Health Monitoring

The primary purpose of the health-monitoring test was to assess the system's responsiveness when a member of the transparent VNF cluster fails. To this end, we generated 20 TCP flows with a bandwidth of 20 Mbps. We also set a connection time of 6000 s and a waiting interval of 300 s between each generated flow. In addition, we established a low granularity (300 s) in OSM and OpenStack to aggregate the collected metrics. Similar to the previous experiments, the transparent cluster started with two active VNFs to which the incoming TCP traffic was evenly distributed according to their CPU load. After 40 min of the simulation, we emulated a failure in a VNF instance (i.e., the one with id 7 in the network service) by putting it on standby.

Figure 4.14 depicts this modification in the selected instance state since its value changed from 1 to 0 just before 16:25 h (1 means healthy and 0 unhealthy). Furthermore, it can be observed that a new instance was activated almost immediately (i.e., in less than a minute) to replace the failed VNF, thus meeting the minimum number of active members in the pool.

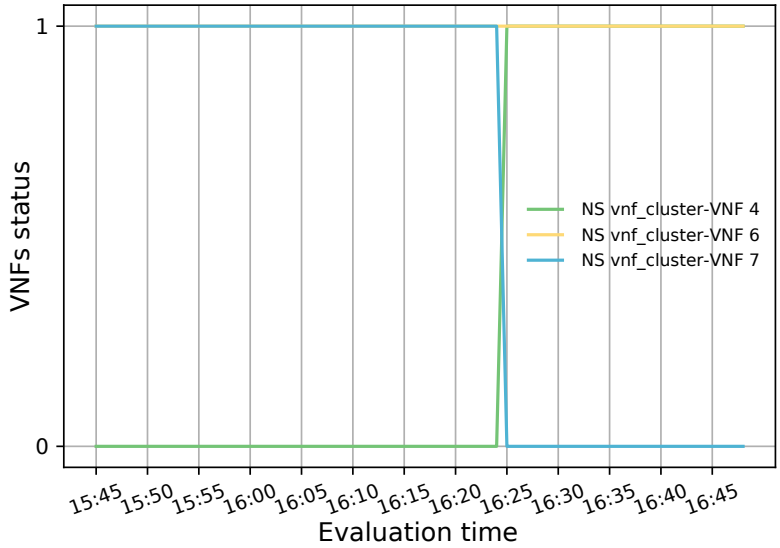


FIGURE 4.14. Status of each member belonging to the transparent cluster.

Figure 4.15 represents the load distribution on each active VNF as well as the activation of a new instance after the failure event. This new instance had higher CPU utilization than the failed one because it processes part of the affected traffic and the newly generated traffic. The latter is evidenced by the flow rules created in the redirectors (e.g., OFS redirector 1) before and after the failure, see Figs. 4.16(a) and 4.16(b), respectively.

By comparing both graphs, we observed that most of the traffic assigned to the failed instance was redirected to the new VNF (i.e., the flow with the destination port [tp_dst] 5003, 5006, and 5002) along with part of the new incoming traffic (i.e., flow with tp_dst 5009). Since the new instance was not active when the traffic arrived, just a small portion of the affected traffic (i.e., flow with tp_dst 5004) was reassigned to the other VNF. From Fig. 4.16(b), it can also be seen that the flows assigned to the healthy instance (VNF connected to the ens6 interface of the OFSs) maintained their affinity throughout the experiment.

Finally, there were several significant delays in publishing the CPU metric in the Prometheus TSDB by the modules in charge of it (e.g., mon-collector and ceilometer[1]). We believe that better performance can be achieved by using higher granularity values (e.g., every second or minute instead of every five minutes). However, these values imply the modification of various parameters

[1]OpenStack project used as a data collection service to provide customer billing, resource tracking, and alarming capabilities across all OpenStack core components.

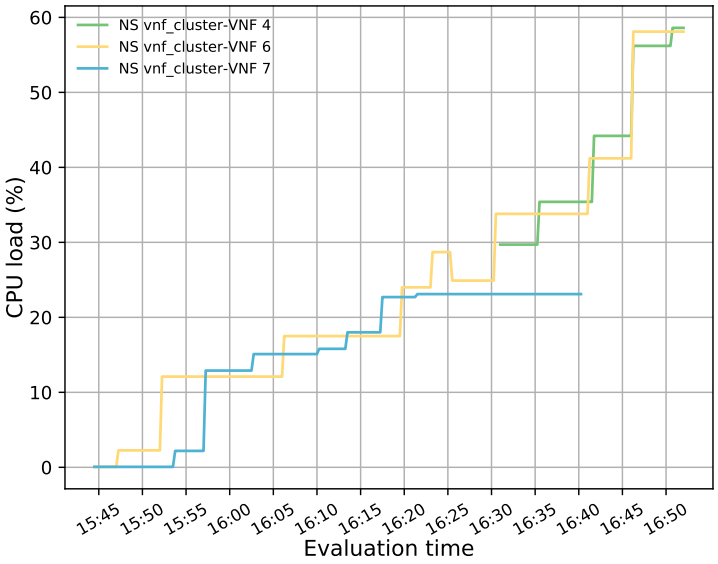


FIGURE 4.15. CPU utilization of each active member in the transparent VNF cluster.

```
ubuntu@vdu-redirector:~$ sudo ovs-ofctl dump-flows br1
 cookie=0x0, duration=2565.896s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=2560.631s, table=0, n_packets=3545672, n_bytes=233998450, priority=500,in_port=ens8 actions=output:ens4
 cookie=0x0, duration=2560.631s, table=0, n_packets=3985643, n_bytes=263057338, priority=500,in_port=ens6 actions=output:ens4
 cookie=0x0, duration=2535.455s, table=0, n_packets=2, n_bytes=140, priority=500,in_port=ens5 actions=output:ens4
 cookie=0x0, duration=2531.933s, table=0, n_packets=1, n_bytes=70, priority=500,in_port=ens7 actions=output:ens4
 cookie=0x0, duration=2493.745s, table=0, n_packets=1, n_bytes=70, priority=500,in_port=LOCAL actions=output:ens4
 cookie=0x0, duration=2331.111s, table=0, n_packets=64, n_bytes=2688, priority=500,arp,arp_spa=10.10.10.25,arp_tpa=10.10.10.104 actions=output:ens8
 cookie=0x0, duration=2330.646s, table=0, n_packets=0, n_bytes=0, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=60423,tp_dst=5004 actions=output:ens8
 cookie=0x0, duration=2031.080s, table=0, n_packets=0, n_bytes=0, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=44671,tp_dst=5005 actions=output:ens6
 cookie=0x0, duration=1929.292s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=47347,tp_dst=5001 actions=output:ens6
 cookie=0x0, duration=1929.174s, table=0, n_packets=2056876, n_bytes=5886658921, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=54993,tp_dst=5001 actions=output:ens6
 cookie=0x0, duration=1629.268s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=39167,tp_dst=5002 actions=output:ens8
 cookie=0x0, duration=1628.900s, table=0, n_packets=1736480, n_bytes=4969712131, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=55221,tp_dst=5002 actions=output:ens8
 cookie=0x0, duration=1329.087s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=59099,tp_dst=5003 actions=output:ens8
 cookie=0x0, duration=1328.815s, table=0, n_packets=1415822, n_bytes=4051946995, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=60757,tp_dst=5003 actions=output:ens8
 cookie=0x0, duration=1028.588s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=59439,tp_dst=5004 actions=output:ens6
 cookie=0x0, duration=1028.408s, table=0, n_packets=1096775, n_bytes=3138897391, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=58795,tp_dst=5004 actions=output:ens6
 cookie=0x0, duration=729.293s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=49827,tp_dst=5005 actions=output:ens6
 cookie=0x0, duration=728.583s, table=0, n_packets=773374, n_bytes=2213158765, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=34029,tp_dst=5005 actions=output:ens6
 cookie=0x0, duration=429.265s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=45675,tp_dst=5006 actions=output:ens8
 cookie=0x0, duration=429.139s, table=0, n_packets=455449, n_bytes=1303363663, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=43679,tp_dst=5006 actions=output:ens8
 cookie=0x0, duration=128.923s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=39709,tp_dst=5007 actions=output:ens6
 cookie=0x0, duration=128.563s, table=0, n_packets=134886, n_bytes=385926337, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=38051,tp_dst=5007 actions=output:ens6
 cookie=0x0, duration=2565.903s, table=0, n_packets=23, n_bytes=1610, priority=0 actions=CONTROLLER:65535
```

reassigned flow rule

(a) Before the failure of the VNF instance 7.

```
ubuntu@vdu-redirector:~$ sudo ovs-ofctl dump-flows br1
 cookie=0x0, duration=3115.045s, table=0, n_packets=0, n_bytes=0, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=3109.780s, table=0, n_packets=4938790, n_bytes=325942834, priority=500,in_port=ens8 actions=output:ens4
 cookie=0x0, duration=3109.780s, table=0, n_packets=6417291, n_bytes=423548743, priority=500,in_port=ens6 actions=output:ens4
 cookie=0x0, duration=3084.604s, table=0, n_packets=2, n_bytes=140, priority=500,in_port=ens5 actions=output:ens4
 cookie=0x0, duration=3081.082s, table=0, n_packets=99732, n_bytes=6591337, priority=500,in_port=ens7 actions=output:ens4
 cookie=0x0, duration=3042.894s, table=0, n_packets=1, n_bytes=70, priority=500,in_port=LOCAL actions=output:ens4
 cookie=0x0, duration=2580.229s, table=0, n_packets=0, n_bytes=0, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=44671,tp_dst=5005 actions=output:ens6
 cookie=0x0, duration=2478.441s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=47347,tp_dst=5001 actions=output:ens6
 cookie=0x0, duration=2478.323s, table=0, n_packets=2584710, n_bytes=7397286277, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=54993,tp_dst=5001 actions=output:ens6
 cookie=0x0, duration=1577.737s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=59439,tp_dst=5004 actions=output:ens6
 cookie=0x0, duration=1577.557s, table=0, n_packets=1624444, n_bytes=4649044129, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=58795,tp_dst=5004 actions=output:ens6
 cookie=0x0, duration=1278.442s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=49827,tp_dst=5005 actions=output:ens6
 cookie=0x0, duration=1277.732s, table=0, n_packets=1301172, n_bytes=3723688681, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=34029,tp_dst=5005 actions=output:ens6
 cookie=0x0, duration=678.072s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=39709,tp_dst=5007 actions=output:ens6
 cookie=0x0, duration=677.712s, table=0, n_packets=662650, n_bytes=1896353353, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=38051,tp_dst=5007 actions=output:ens6
 cookie=0x0, duration=378.350s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=56919,tp_dst=5008 actions=output:ens6
 cookie=0x0, duration=377.867s, table=0, n_packets=342643, n_bytes=980321365, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=42755,tp_dst=5008 actions=output:ens6
 cookie=0x0, duration=31.924s, table=0, n_packets=8, n_bytes=674, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=32927,tp_dst=5009 actions=output:ens7
 cookie=0x0, duration=30.733s, table=0, n_packets=29971, n_bytes=85542175, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=39727,tp_dst=5009 actions=output:ens7
 cookie=0x0, duration=22.944s, table=0, n_packets=24138, n_bytes=69075966, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=60757,tp_dst=5003 actions=output:ens7
 cookie=0x0, duration=22.667s, table=0, n_packets=23791, n_bytes=68077260, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=43679,tp_dst=5006 actions=output:ens7
 cookie=0x0, duration=22.396s, table=0, n_packets=23499, n_bytes=67244352, priority=500,tcp,nw_src=10.10.10.25,nw_dst=10.10.10.104,tp_src=55221,tp_dst=5002 actions=output:ens7
 cookie=0x0, duration=31.524s, table=0, n_packets=4, n_bytes=168, priority=500,arp,arp_spa=10.10.10.25,arp_tpa=10.10.10.104 actions=output:ens7
 cookie=0x0, duration=3115.052s, table=0, n_packets=33, n_bytes=6520, priority=0 actions=CONTROLLER:65535
```

new flow rule
reassigned flow rule

(b) After the failure of the VNF instance 7.

FIGURE 4.16. Flow table of redirector 1.

and configuration files at both OSM and OpenStack levels, which is not trivial and lacks detailed documentation.

4.4 Conclusion

In this chapter, an SDN-based solution was proposed to dynamically manage a cluster of transparent VNFs. More specifically, our proposal implements a modular application that runs on top of SDN controllers to perform load-balancing and auto-scaling decisions. This application can also be integrated with the DDS application presented in Chapter 3 to increase the fault tolerance of the system while performing the functions mentioned above.

Regarding the presented solution, the load-balancing module guarantees bidirectional flow affinity without packet modification by simultaneously configuring OF rules in the switches compounding the proposed approach. In contrast to most of the literature reviewed on this topic, our solution was implemented in a real environment using two well-known frameworks, OSM and OpenStack, instead of simulation tools such as Mininet.

The results of the evaluation validated the feasibility of the solution, which successfully redirects E2E traffic through the transparent cluster without losing any packets. In addition, the bidirectional flow affinity of the solution was demonstrated by finding each pair of source-destination addresses attached to the same port in the switches tables. Similarly, several experiments found the effectiveness of the load-balancing and auto-scaling blocks.

Furthermore, the performance of the monitoring module was studied through a health test. This experiment reflected the activation of a new instance to meet the minimum number of active members in the cluster after detecting a failed member.

CHAPTER 5

ENERGY-FRIENDLY SCHEDULER FOR EDGE-COMPUTING SYSTEMS

This chapter is based on:

- **A. Llorens-Carrodeguas**, S. G. Sagkriotis, C. Cervelló-Pastor, and D. P. Pezaros, "An Energy-Friendly Scheduler for Edge Computing Systems," *Sensors*, vol. 21, no. 21, Oct 2021.

In Chapter 3, the agent entity was introduced as an element of the SDN controller hierarchy by considering the edge-computing paradigm. These agents can be represented as edge nodes. Therefore, providing them with mechanisms to extend their lifetime is crucial since most are battery-powered. Additionally, existing scheduling processes of prominent platforms (e.g., Kubernetes) do not integrate energy measurements of the participant devices in the placement decisions. This limitation can underutilize the available energy resources or attempt deployments bound to fail due to insufficient resources. Moreover, the capacities of the node performing the scheduling decisions are not included in the pool of resources for VNF deployment. This exclusion results in underutilized computational and energy capacity.

In this context, this chapter presents a scheduler to improve the resource utilization of energy-constrained edge nodes and thus increase their resilience. Section 5.1 presents the problem statement and the notation and system model used in our proposal. To this end, we consider a reference architecture that captures the environment characteristics mentioned above. We introduce the proposed scheduling algorithm and describe the functions of its modules in Section 5.2. Finally, we discuss the results in Section 5.3.

5.1 Energy-Constrained Edge Nodes Cluster: Problem Description

Within the reference architecture depicted in Fig. 5.1, a controller node is deployed alongside a set of computing nodes (N_c) in a commodity cluster of SBCs. Incoming event requests are scheduled and deployed according to their deadline and resource requirements. We consider the computing nodes energy-constrained devices with fixed computational resources. Thus, the proposed scheduler must place events by considering the remaining battery of each node with the ultimate goal of increasing the lifetime of the cluster and, therefore, its resilience. The controller node must be capable of coordinating and managing the deployed services and tasks. It must also monitor each node's battery status, including its battery levels.

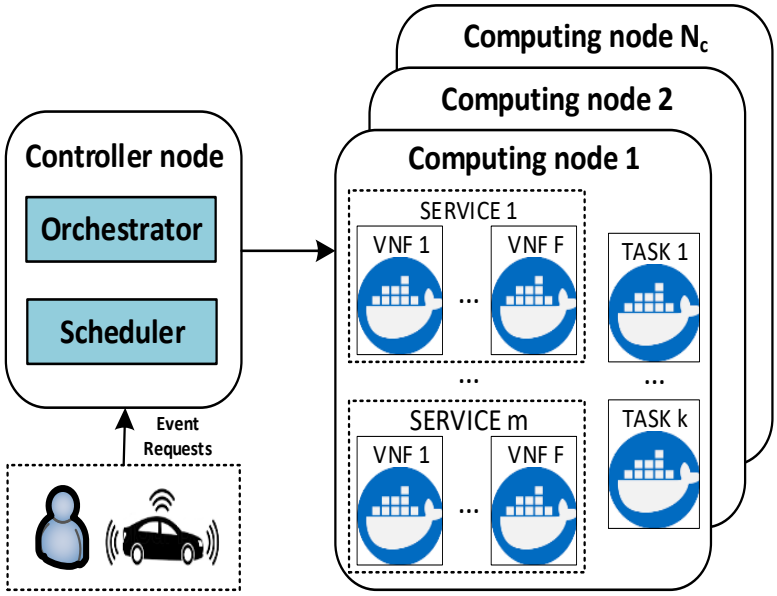


FIGURE 5.1. Reference architecture formed by a controller node and multiple computing nodes.

5.1.1 Problem Modeling and Notation

The cluster is treated as a pool of available resources that must be managed to avoid violating deadlines or energy constraints. We consider two types of events scheduled in the cluster: tasks and network services. The former is a set of instructions that require a predefined and fixed value of execution time, such as log rotation[2] associated with a particular running service, processing rows from a service database table, and backing up a service database before its deletion. Executing a task consumes specific required resources that are released upon completion. In the

[2]Process of compressing logs files that are older than a particular time and deleting ancient ones.

case of a network service event, it is composed of a set of VNFs that can reserve resources for an arbitrary amount of time that is not known in advance.

The event requests are distributed among computing nodes that are geographically distant across multiple locations. Thus, we can include use cases that require fog-to-fog communication or other geo-distributed applications. A well-known use case is a set of drones on-boarding SBC nodes to enable emergency service communications (e.g., IP telephony) between islands during natural disasters such as earthquakes, fires, and floods. In this situation, each drone would run different VNFs: access point, domain name system (DNS), and access router [141, 142].

The scheduling algorithm examines the incoming events in an online manner. In other words, they are treated as they arrive and by considering the current resource capacity of the cluster. The events are placed in a set of deployable units within computing nodes (P). P represents all the virtual nodes[3](p) created on all the physical nodes where events can be scheduled (N). Each virtual node $p \in P$ is identified with an ID. The parameter p_i^n indicates that virtual node p_i is placed in physical node n, where $n \in N$. Any given network service (S) is formed by a sequence of VNFs (F), where each function f must be processed on a set of physical nodes. These functions must be scheduled one after the other in a specified sequence. Each virtual node p created can only process one function at a time. Similarly, any given task (T) will be processed on the selected physical node, where a virtual node p will be created to execute the requested task. Each event has a demanded rate (r_e) that the selected node must meet. Additionally, the processing capacity of the node (c_n) must satisfy the event's demanded rate (i.e., $c_n \geq r_e$).

All the VNFs and tasks have a running time parameter (t_r) that denotes the amount of time that must pass before an event can be considered complete. When $t_r > 0$, the event runs during the specified time. If this parameter is 0 or not specified, the event will be executed during the system's lifetime. Both types of events also have a starting time (t_s) and a completion time (t_c). The former represents the precise instant when the selected node starts processing the event. The latter specifies when the VNFs or the tasks finish their execution. Thus, we can calculate the execution time (t_e) of an event through the following equation:

$$t_e = t_c - t_s \tag{5.1}$$

In addition, the event's time of arrival (t_a) is stored to calculate the time that an event is in the system (t_t). Thus, the first parameter (t_a) denotes the time when the controller node received the scheduling request. Additionally, the t_t parameter represents the total time starting from when an event arises until its completion and can be calculated as follows:

$$t_t = t_c - t_a \tag{5.2}$$

Since our system will receive event requests with an unknown arrival time, we introduce a priority queue to the controller node. According to user demands, the events can have different

[3]The virtual nodes (p) are equivalent to Kubernetes pods.

priorities and are sorted by considering specific criteria. The priority queue works as a centralized event allocation system to coordinate several nodes. More specifically, the scheduler takes the highest element in the list and chooses the best physical node to deploy a virtual node running the event (task or VNF). The priority queue introduces an unavoidable delay in the assignation node procedure because the controller node schedules events one by one. In this regard, we define the waiting time (t_w) of an event as the amount of time from its arrival until its execution starts. This parameter can be obtained from the following equation:

$$t_w = \begin{cases} t_s - t_{a_{f=1}} & \text{if the event is a service} \\ t_s - t_{a_T} & \text{if the event is a task} \end{cases} \tag{5.3}$$

Waiting and total time parameters are considered evaluation metrics that capture the scheduler's performance. From Equation (5.3), we obtain the waiting time for each event. In the case of a service, we need the arrival time of the first network function as it represents the beginning of the service.

Finally, a deadline (d) is defined for processing a given event. In the case of a network service, the processing of its last function must be completed by this time. Otherwise, the scheduler incurs a service level agreement (SLA) violation. Table 5.1 provides a list of notations related to the system model.

Table 5.1: System model notation.

Notation	**Description**
P	Set of deployable units of computing nodes
N	Set of physical nodes where events can be scheduled
N_c	Set of computing nodes ($N_c \subset N$)
S	Network service request arriving to controller node
F	Sequence of VNFs compounding a network service request
T	Task request arriving to controller node
p	Each virtual node created on the physical nodes to run the events
n	Each physical node where virtual nodes are created
p_{f_i}	Indicates the virtual node where function f_i is running
p_T	Indicates the virtual node where task T is running
f_i	Each VNF forming part of a network service
r_e	Demanded rate of each task and VNF
c_n	Processing capacity of each physical node ($n \in N$)
t_r	Running time of an event before considering it completed
t_s	Starting time of an event when being processed in the selected node
t_c	Completion time of an event in the selected node
t_e	Execution time of an event in the assigned node
t_a	Arrival time of an event request in the controller node
t_t	Total time of an event in the system
t_w	Waiting time of an event in the priority queue
d	Deadline for processing a given event

5.2 Proposed Scheduling Solution: SOCCS

In this section, we propose our SOC and capacity-based scheduler (SOCCS). It processes event requests and determines the best node in an SBC cluster to run them based on the estimations of remaining battery and CPU usage in the nodes. The proposed scheduler is formed by three main modules: the SOC estimator, the monitor, and the scheduler. The relationship among these elements is depicted in Fig. 5.2, which shows how these blocks communicate with the orchestrator (represented as blue lines) and the communication between the orchestrator and the cluster elements (represented as green lines). In this representation, the set of nodes where events can be scheduled (N) comprises the controller node and the computing nodes ($N = N_c + 1, N_c \subset N$). The following sections explain the functions of each block.

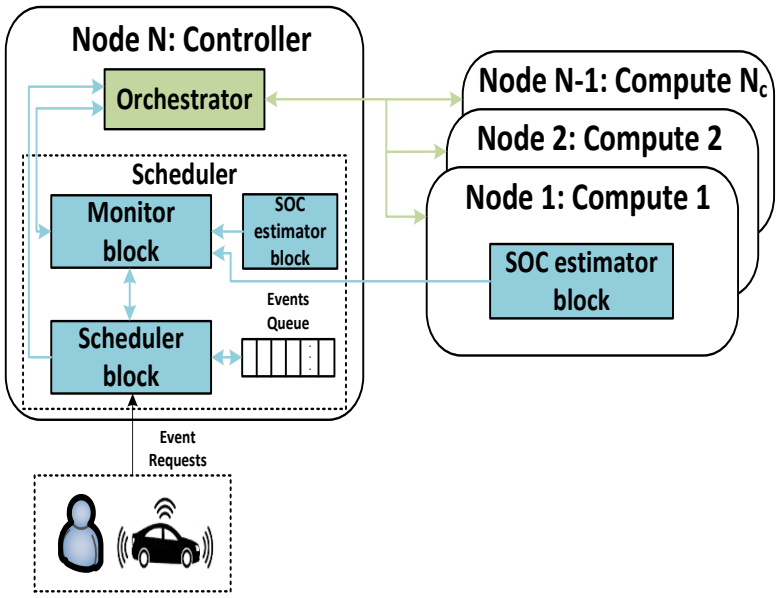


FIGURE 5.2. Scheduling module design and the relationship among its different blocks.

5.2.1 SOC Estimator Block

This element runs in every node comprising the SBC cluster. This module receives the necessary information from the measurement equipment and calculates the SOC using the coulomb counting method [85].

The SOC estimation can be dependent on specific characteristics of the batteries (e.g., state of health and model) according to the estimation model used [143]. Such factors are related to the battery hardware. Thus, the estimators must be updated when there is a change in the hardware. The battery hardware-dependent estimations are slightly evident in the coulomb counting method. This mechanism monitors the total electric charge that a battery absorbs or releases during the charging or discharging phases.

The estimation of the SOC can be achieved by dividing the percentage of the released electric charge in the battery by the one that entered it. We denote the released capacity when the battery is completely discharged as $Q_{releasable}$ and the rated capacity as Q_{rated}. Then the SOC percentage can be obtained as follows:

$$SOC = \frac{Q_{releasable}}{Q_{rated}} \cdot 100\% \tag{5.4}$$

The proposed estimator block adopts the coulomb counting method due to its accurate yet straightforward approach. The Q_{rated} term is obtained by considering the actual electric charge that the battery can deliver over several charge-discharge cycles to get more accurate results. Similar to the proposed methodology in [85], we can find the coulombic efficiency (η) of the rated capacity. Additionally, we used the maximum releasable capacity (Q_{max}). Thus, Equation (5.4) is adjusted to:

$$SOC = \frac{Q_{max}}{\eta \cdot Q_{rated}} \cdot 100\% \tag{5.5}$$

The SOC is given by Equation (5.5) when the battery is fully charged. However, during a discharging phase, we should know the percentage of the capacity relative to the Q_{rated} term, which is denoted as the depth of discharge (DOD). The DOD is obtained from a measured charging and discharging current (I_b) in an operating period τ and then subtracted from the total SOC, as shown in the following equations:

$$\Delta DOD = \frac{-\int_{t_0}^{t_0+\tau} I_b(t)\,dt}{\eta \cdot Q_{rated}} \cdot 100\% \tag{5.6}$$

$$DOD(t) = DOD(t_0) + \Delta DOD \tag{5.7}$$

$$SOC(t) = 100\% - DOD(t) \tag{5.8}$$

The DOD is an accumulated value, as shown in Equation (5.7). We can estimate the battery's SOC by using Equation (5.8) at any time. The estimation process is based on the measured voltage and current. This block knows the battery operation mode according to the value and direction of the operating current. During the discharging phase, the DOD adds up the drained charge until reaching the Q_{max} value when the battery is exhausted (i.e., $SOC = 0\%$). In contrast, the DOD counts down the accumulated charge in the charging phase until the battery is fully charged (i.e., $SOC = 100\%$). The SOC estimator block uses the procedure mentioned above to estimate the battery's SOC in each node where it is running. Finally, it sends the estimated value to the monitor block.

5.2.2 Monitor Block

This module is in charge of monitoring all the virtual nodes that have been created. Similarly, it monitors the events scheduled by the scheduling algorithm described in Section 5.2.3. It also

checks the status and usage of the physical nodes by communicating with the orchestrator. More specifically, this module tracks the CPU and memory consumption of each node, as summarized in Procedure 5.1.

Procedure 5.1: Update Nodes

```
1 forall n ∈ N do
2     Update node usage by adding up the CPU and memory utilization of all the virtual nodes
        placed in this node
3     if n_usage_CPU ≥ Usage_CPU_max and n_status is scheduled then
4         n_status ← unscheduled
5     if n_usage_CPU < Usage_CPU_max and n_status is unscheduled then
6         n_status ← scheduled
```

This procedure updates the resource utilization of each node within the SBC cluster. It determines a node's resource utilization by calculating the whole usage of its virtual nodes in terms of CPU and memory (line 2). We assume the cluster's nodes as candidates to place a created virtual node. Thus, this procedure checks if a current node's utilization has not reached its defined maximum capacity (line 3). When the maximum capacity has been reached, the node's status is marked as *unscheduled*, excluding the node from the candidate selection process in the scheduling algorithm (line 4). In contrast, line 5 verifies that the current node's usage is below its maximum value. If so, the node's status is set to *scheduled* in line 6 if it was previously marked as *unscheduled*. The monitor block uses the procedure above, and the behavior of this module is described in Algorithm 5.1.

Algorithm 5.1: Monitor Process

```
1  Event_rejected ← 0 (Amount of rejected events)
2  Event_violations ← 0 (Amount of deadline violation in events)
3  while True do
4      forall p ∈ P do
5          if p_status is Running then
6              Get CPU and memory usage from metrics server
7              Save CPU and memory values in p_usage
8          else
9              if p_status is Succeeded and p_tc > p_d then
10                 Event_violations = Event_violations + 1
11             if p_status is Failed then
12                 Event_rejected = Event_rejected + 1
13             Delete the virtual node running the event to release it resources
14             Remove p from P
15             Remove S or T from l_S or l_T accordingly
16     Procedure 1: Update Nodes
```

The monitor block's procedure begins by initializing two parameters (lines 1–2) that run

during the system's lifetime (line 3). It checks several parameters (e.g., p_{status}, p_{t_c}, p_d) for all the virtual nodes created ($p \in P$) to verify if specific conditions have been satisfied. When a virtual node is running, this module gathers its CPU and memory utilization from a metrics server (e.g., Prometheus) and stores these values in p_{usage} (lines 5–7). Otherwise, the event is considered to be in one of two possible states: *succeeded* or *failed*. When an event has completed its execution, the virtual node where it was running is marked as succeeded. If the event completes its execution after its deadline (line 9), the algorithm updates the number of deadline violations in line 10. When the condition related to the failed virtual nodes is satisfied (line 11), the algorithm updates the number of rejected events in line 12. Afterward, the algorithm releases the resources used and updates the respective parameters (lines 13–15). In the last step (line 16), the updating nodes procedure is called. The rejected events and deadline violation parameters are later used as evaluation metrics of the proposed scheduler. These metrics are analyzed in detail in Section 5.3.

Finally, the monitoring block receives, in a parallel process to Algorithm 5.1, the SOC battery information sent by the SOC estimator block. This information is saved in $n_{usage_{SOC}}$. Consequently, the scheduling block can obtain the utilization of a node in terms of CPU, memory, and SOC by reading the stored values in n_{usage}.

5.2.3 Scheduler Block

This module determines the best node where an event can run according to the SOC prediction. It receives the event requests one after the other and appends them to a priority queue. Simultaneously, this block takes the events from the priority queue individually to determine the node where each event will run by considering the remaining battery estimations and CPU usage. The SOC prediction is determined through a regression model [144] explained in Section 5.2.3.1. Procedure 5.2 summarizes the SOC prediction methodology used by Algorithm 5.2.

Procedure 5.2: SOC prediction

Input: p^0, n
Output: SOC_{value}
1 $SOC_{value} \leftarrow 0, data \leftarrow \varnothing$
2 **if** *n is controller* **then**
3 $\quad cpu \leftarrow n_{usage_{CPU}} + p^0_{CPU_{req}}$
4 $\quad pkt_{in} \leftarrow n_{pkt_{in}} + \frac{n_{pkt_{in}}}{\sum_{p \in P}}$
5 $\quad pkt_{out} \leftarrow n_{pkt_{out}} + \frac{n_{pkt_{out}}}{\sum_{p \in P}}$
6 $\quad data \leftarrow cpu, pkt_{in}, pkt_{out}, n$
7 **else**
8 $\quad cpu \leftarrow n_{usage_{CPU}} + p^0_{CPU_{req}}$
9 $\quad data \leftarrow cpu, n$
10 $SOC_{value} \leftarrow SOC_{pred_{model}}(data)$
11 **return** SOC_{value}

This procedure uses the trained prediction model to forecast the SOC value that a node would have if a specific virtual node running an event were assigned to it. Procedure 5.2 takes as input terms the current node and the virtual node to be scheduled. Then it extracts and determines the required information to predict the SOC value using the regression model. In line 1, the SOC prediction procedure initializes the output variable and creates an empty set to store the data used by the prediction model. Next, the procedure checks if the current node is the controller node since the data used by the model is determined by the node's type (line 2). Section 5.2.3.1 explains the reasons for making this differentiation in the input data to the model. Lines 3–6 determine and store, in $data$ variable, the values required by the model in the case of the controller node. If the virtual node was deployed in a physical node, its expected CPU usage is calculated by adding the current CPU usage and the required CPU of the event running in the virtual node (line 3). Once the virtual node is scheduled, lines 4 and 5 determine the expected overall number of packets exchanged between the controller and the computing nodes. The obtained values are then stored in the dataset (line 6). In the case of computing nodes, the procedure only factors in the expected CPU usage and saves this value in the dataset (lines 7–9). Finally, the prediction value is derived from the SOC regression model considering the stored data (line 10).

Algorithm 5.2 is executed during the scheduling algorithm's lifetime with existing elements in the priority queue (line 1). Then the algorithm gets the nodes' capacity from the monitor block before analyzing any possible candidate to assign an event (line 2). After gathering the usage of the nodes, the algorithm takes the first element in $l_{priority}$ and initializes the list of possible candidates to host the new virtual node (lines 3–5). The following steps define the potential candidate SBCs where virtual nodes can be placed. Each physical node with a battery percentage above a minimum predefined value and a scheduled status is added to $l_{candidate}$ (lines 6–8). After these steps, the algorithm removes the controller node from the $l_{candidate}$ list when it is present in the list and there is at least one available computing node (lines 9–10). Thus, the algorithm increases the controller's longevity and avoids extra processing for deploying an event on it.

In the case that the previous condition is not met (line 9), we analyze two possibilities with the same outcome (line 11). The first possibility corresponds to when no node can host a virtual node. The second one is more complex since the controller is the only available node, and the event to schedule will run during the system's lifetime. For both cases, the associated event (i.e., service or task) is rejected, and $Event_{rejected}$ metric is updated (lines 12–22). Accordingly, the created virtual node is removed to release its resources, and P, l_S, and l_T are updated. In the case of a service event, the algorithm checks each former VNF. If the function has already been deployed, it is deleted to release the associated resources. The function is also removed to avoid being considered by the scheduler when found in the priority queue, thus saving further resources. After not meeting the conditions mentioned above (lines 9 and 11), we have at least one node in $l_{candidate}$ (line 23). It should be noted that the controller node can be in $l_{candidate}$ when there are no more available computing nodes and the events to be placed have a specified

Algorithm 5.2: Event Scheduler

```
1  while len(l_priority) > 0 do
2      Get nodes' capacity information
3      p^0 ← First element of l_priority
4      Update l_priority
5      l_candidate ← ∅
6      forall n ∈ N do
7          if n_usage_SOC > SOC_min_threshold and n_status is scheduled then
8              l_candidate ← l_candidate + n
9      if l_candidate > 1 and n^controller ∈ l_candidate then
10         Remove n^controller from l_candidate
11     else if l_candidate == 0 or (l_candidate == 1 and n^controller ∈ l_candidate and p^0_{t_r} == 0) then
12         if p^0_event is Service then
13             forall f ∈ F do
14                 if p_{f_i} is deployed then
15                     Delete p_{f_i} to release its resources
16                 if p_{f_i} ∈ l_priority then
17                     Remove p_{f_i} from l_priority
18         else
19             Delete p_T to release its resources
20         Remove p from P
21         Remove S or T from l_S or l_T accordingly
22         Event_rejected = Event_rejected + 1
23     else
24         n^best ← First element in l_candidate
25         forall n ∈ l_candidate do
26             if SOC_pred_model ≠ ∅ then
27                 n_SOC_pred ← Procedure 2: Predict SOC(p^0, n)
28                 Calculate n_score using Equation (5.9) with n_SOC_pred
29             else
30                 Calculate n_score using Equation (5.9) with n_usage_SOC
31             if n_score > n^best_score then
32                 n^best ← n
33         Bind p^0 to n^best
```

running time ($t_r > 0$). In this manner, our proposal scheduler reduces the number of rejected events, as demonstrated in Section 5.3.

Algorithm 5.2 continues by assuming the first element in $l_{candidate}$ as the best node (line 24). This node is then analyzed to determine if any candidate node has a better score than this current best node (lines 25–32). Afterward, the algorithm verifies if the SOC predictor model exists (line 26). If this is the case, the predictor model calculates the SOC of the current candidate node through Procedure 5.2 (line 27).

The output of the SOC prediction procedure is used in Algorithm 5.2 to calculate the node score through Equation (5.9) (line 28). By using this equation, the algorithm tries to maximize the node score by selecting the node with the highest SOC and the minimum CPU usage.

$$n_{score} = \alpha_1 \cdot (SOC/100) + (1 - \alpha_1) \cdot (1 - \mathbb{E}_{CPU}/Usage_{CPU_{max}}) \tag{5.9}$$

In Equation (5.9), the value α_1 is an adjustable positive weight with values between 0 and 1. The SOC term corresponds to either a predicted value (i.e., using the SOC regression model) or a real measured value from the SOC estimator block. The expected CPU usage ($\mathbb{E}_{CPU}$) represents the new CPU usage that the analyzed node would have if a virtual node running an event were scheduled to it. More specifically, this value is calculated by adding the current CPU usage of the node and the required CPU usage of the event. If the SOC predictor model is not available, the steps are similar to the previous ones but using the current SOC of the node to calculate its score (lines 29–30). After calculating n_{score}, the algorithm checks if this value is higher than the best node score (line 31). In that case, the current node is taken as the new best node (line 32). Finally, the scheduler block communicates to the orchestrator to bind p^0 in n^{best} (line 33).

Algorithm 5.3 represents the main process of the scheduler application since it initializes the other processes (i.e., the monitor process, the event scheduler and the regression model handler [see Section 5.2.3.1]) (lines 5–7). This algorithm begins by initializing the event lists and the set of virtual nodes, which can be updated by both the scheduler and monitor blocks (lines 1–3). In addition, it initializes the SOC prediction model used by Algorithm 5.2 and the dataset to train the model in Algorithm 5.4 (line 4). The $data\_training$ variable is updated in each cycle of Algorithm 5.1. The primary process of the proposed scheduler waits for any event request while the event scheduler module is running (line 8). In line 9, the algorithm adds the event request to the corresponding list when it arrives. Then this algorithm creates a virtual node with the requirements for the event and adds it to the set of virtual nodes (line 10). After creating the virtual node, the events' ranking in the priority queue ($l_{priority}$) (lines 11–13) is determined according to two factors: $delay(p)$ and $wait_{queue}(p)$. The former represents the amount of time that the scheduler can delay the execution of an event without missing its deadline, as shown in Equation (5.10). The latter is the waiting time of the event before being processed by the scheduler, see Equation (5.11). We denoted the current system time as t_{now} in both equations. It should be noted that the smaller the $delay(p)$, the faster the created virtual node will be executed.

$$delay(p) = p_d - t_{now} - p_{t_r} \tag{5.10}$$

$$wait_{queue}(p) = t_{now} - p_{t_a} \tag{5.11}$$

Considering the previous definitions, we calculate the ranking score for the virtual node running an event (p_{rank}) as follows:

$$p_{rank} = \beta_1 \cdot delay(p) - (1 - \beta_1) \cdot wait_{queue}(p) \tag{5.12}$$

Algorithm 5.3: Main process

1 $P \leftarrow 0$
2 $l_S \leftarrow 0$ (List of running services)
3 $l_T \leftarrow 0$ (List of running tasks)
4 $SOC_{pred_{model}} \leftarrow \varnothing$, $data\_training \leftarrow \varnothing$
5 **Algorithm 5.1:** Monitor Process
6 **Algorithm 5.2:** Event Scheduler
7 **Algorithm 5.4:** Regression Model Handler
8 **while** *True* **do**
9 Add S or T to l_S or l_T accordingly to the event request
10 Create p for S or T and add it to P
11 Determine maximum delay to process p through Equation (5.10)
12 Determine the time before putting p into priority queue through Equation (5.11)
13 Calculate p_{rank} using Equation (5.12)
14 Add p to $l_{priority}$
15 Sort $l_{priority}$ by virtual node ranking

The β_1 term represents an adjustable positive weight with values between 0 and 1. A virtual node with the lowest ranking must be executed first. Thus, the algorithm updates the priority list and sorts the queue according to the calculated ranking of the virtual node (lines 14–15).

5.2.3.1 SOC Regression Model

Section 5.2.1 describes a methodology under the umbrella of direct calculation and model-based methods to obtain the battery SOC. In contrast to these methods, the data-driven ones do not require an equivalent circuit or electrochemical mechanism model to describe battery behaviors. Thus, this type of method can estimate the battery SOC through sampled data by finding a relationship between the data and the SOC measurements. Some examples of data-driven methods include autoregression moving average (ARMA), artificial neural network (ANN), and support vector machine (SVR) [145]. These techniques can cause a significant computational burden when treating enormous amounts of training data. They must also be trained in an initial state before their hyperparameters can be adjusted. Therefore, these mechanisms might not be feasible for use cases where an SBC battery-powered cluster must also process service and task requests.

In the case of regression models, their coefficients are determined from available training data by minimizing the root mean square error (RMSE) between the predicted and measured values. The RMSE represents the standard deviation of the prediction errors, thus showing how concentrated the data is around the line of best fit [144]. In general, regression models can be classified into two types: polynomial and linear. The former may include higher powers of one or several predictor variables, see Equation (5.13). The latter may involve the interaction effects of two or more variables, therefore representing an example of a multiple linear regression model, see Equation (5.14) [144]. A detailed analysis to select the model regression used in our proposal

is shown in Section 5.3.2.

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + ... + \beta_k x^k \tag{5.13}$$

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{12} x_1 x_2 + ... + \beta_{mn} x_m x_n \tag{5.14}$$

In this chapter, we adopt a regression model for the SOC estimation as the computational characteristics of SBCs might not be capable of supporting complex algorithms such as ARMA, ANN, and SVR. The proposed regression model is trained with an initial dataset formed by collected metrics during a defined period. We update the model coefficients when the RMSE metric is above a predefined threshold to improve its accuracy. Algorithm 5.4 is responsible for calculating and updating these coefficients.

Algorithm 5.4: Regression Model Handler

```
1  while True do
2      Wait time to check regression model estimators
3      if SOC_pred_model is ∅ then
4          SOC_pred_model ← train_model(data_training)
5          data_training ← ∅
6      else
7          predictions ← SOC_pred_model(data_training)
8          Determine RMSE between predictions and SOC_real in data_training
9          if RMSE > error_threshold then
10             SOC_pred_model ← update_model(data_training)
11             data_training ← ∅
```

This algorithm verifies the SOC regression model after waiting a predefined time (line 2). Then it checks if the model has not been training (line 3). If this is the case, the algorithm trains the model with the existing training dataset ($data\_training$), and afterward, this dataset is initialized to gather new data for future model examinations (lines 4–5). In the case of an existing model (line 6), the algorithm studies its coefficients to determine if they must be updated (lines 7–11). Using the existing gathered data, the algorithm makes several SOC predictions based on the trained model (line 7). Subsequently, it determines the RMSE between the predictions and the measured SOC in $data\_training$ (line 8). Later, the calculated RMSE is compared to a predefined threshold to determine if it is above this value (line 9). If this condition is met, Algorithm 5.4 updates the SOC regression model by considering the existing dataset (line 10). After this step, the training dataset is restarted, and newly gathered data is added through the monitoring process (line 11).

5.3 Evaluation and Results

To evaluate the performance of our proposed scheduling algorithm, we built a testbed formed by a cluster of four Raspberry Pi 4 Model B [14]. This equipment represents a regular IoT device

that can connect with a variety of sensors and offer edge-processing capabilities. We used the UM24C module [146] to measure the power consumption of each node. The power sources for the Pi devices are batteries with a capacity of 10000 mAh. Figure 5.3 depicts the described testbed. We deployed Kubernetes 20.04 as the management framework for virtualized services and tasks. Namely, these events run as Docker containers within pods and are placed into the devices by considering their available capacity according to predefined requirements. The proposed scheduling algorithm (i.e., SOCCS) was implemented using Python 3.6.8 and deployed within Kubernetes. More specifically, it was deployed with similar privileges and roles to the native scheduler of Kubernetes. Therefore, the descriptors of the events to be deployed must indicate the scheduler name (i.e., SOCCS or Kube-scheduler) that must attend the request.

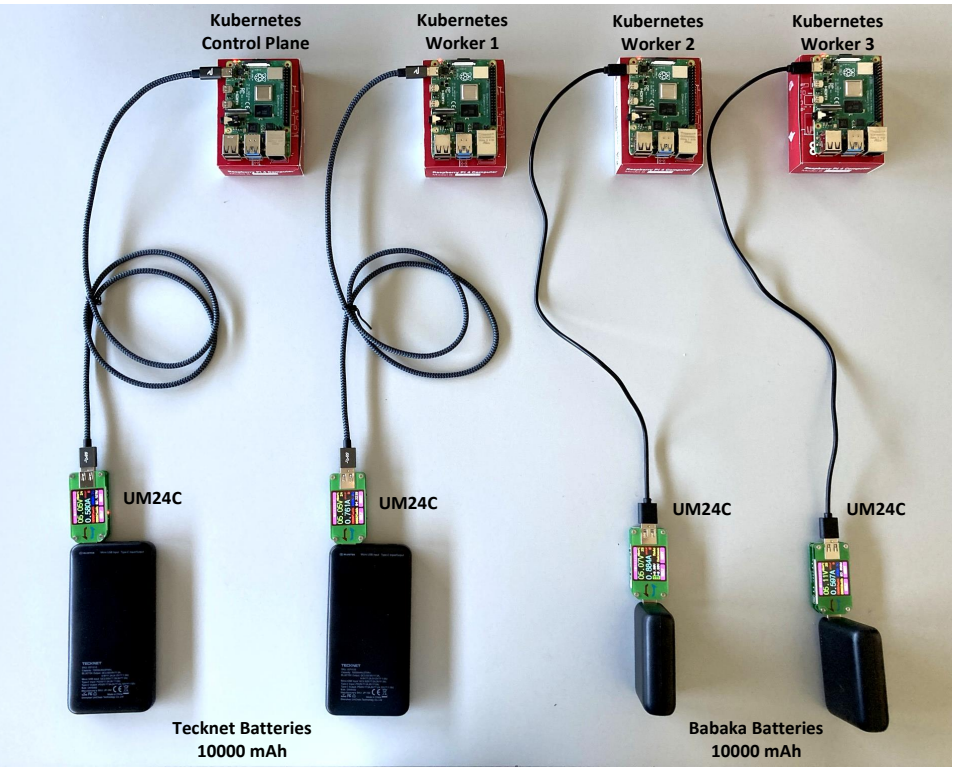


FIGURE 5.3. SBC cluster using Raspberry Pi devices running Kubernetes.

In our evaluation scenarios, the services and tasks to be scheduled arrive one at a time following a Poisson distribution. We explore different event arrival rates that range from 2 to 12 events per time unit. Table 5.2 summarizes the main parameters used for creating the services and tasks. More specifically, they were selected randomly from the values following a uniform distribution. The evaluation parameters are defined based on typical workloads derived from the literature. Notably, the CPU usage is measured in *CPU* units and is expressed as an absolute quantity. Thus, 100 milli-CPU and 0.1 CPU are identical amounts of CPU usage in a single-core, dual-core, or 48-core machine.

Table 5.2: Evaluation parameter ranges based on testbed.

Parameter	**Values**
Number of VNFs in a service	5 - 10
Processing capacity per node (MIPS)	500 - 3000
CPU capacity per node (milli-CPU)	4000
Memory capacity per node (Ki)	7998464
Required processing rate per event (MIPS)	100 - 500
Required CPU per event (milli-CPU)	150 - 250
Required memory per event (Ki)	200 - 500

5.3.1 Utilizing Unused Controller Resources

This section analyzes the impact of deploying selected events to the controller (i.e., Kubernetes master node). Different from previous works in cloud and edge computing [92, 147, 148], this work included the controller node in the scheduling process to deploy event requests. The reviewed papers do not consider deploying them in the controller node because they speculate this will cause additional processing overhead in the controller, thus increasing the total time of the events in the system. However, we can reduce the adverse effects of deploying events in the controller by selecting those with a fixed execution time. In particular, the controller can host a new event using the released resources of the previous one. Therefore, it increases the number of overall scheduled events and the overall acceptance ratio. In this manner, our scheduler effectively uses all the available resources in the cluster.

To analyze the impact of this decision, we ran several experiments for both types of scheduling (including/excluding the controller). The results are presented with a confidence interval of 95%. Figure 5.4 depicts the average number of successfully scheduled and rejected events and deadline violations for different event generation rates. We considered successfully scheduled events the ones that did not exceed their deadlines. In these experiments, we assume that surpassing a deadline would not lead to a task or service interruption since maintaining a required QoS is a desirable parameter but not mandatory [94].

The results show that the number of successfully scheduled events was relatively similar for both criteria (i.e., deploying and not deploying in the controller node) for all the event generation rates. However, there is a noticeable difference in rejections and deadline violations for generation rates of 8, 10, and 12. In the case of events assigned to the controller, we observe a reduction of 50%, 61%, and 66% in the number of rejected events, respectively. These reductions are more significant than the increments in the deadline violations, which can be up to 16%, 42%, and 49% for the same generation rates.

Figure 5.5(a) shows the events' average waiting time for both criteria. The waiting time increased when services and tasks were deployed to the controller (soft orange line) with regard to the other criterion (strong red line). This behavior resulted from placing them one by one using the resources released by the previous event. Consequently, no extra load was added to the

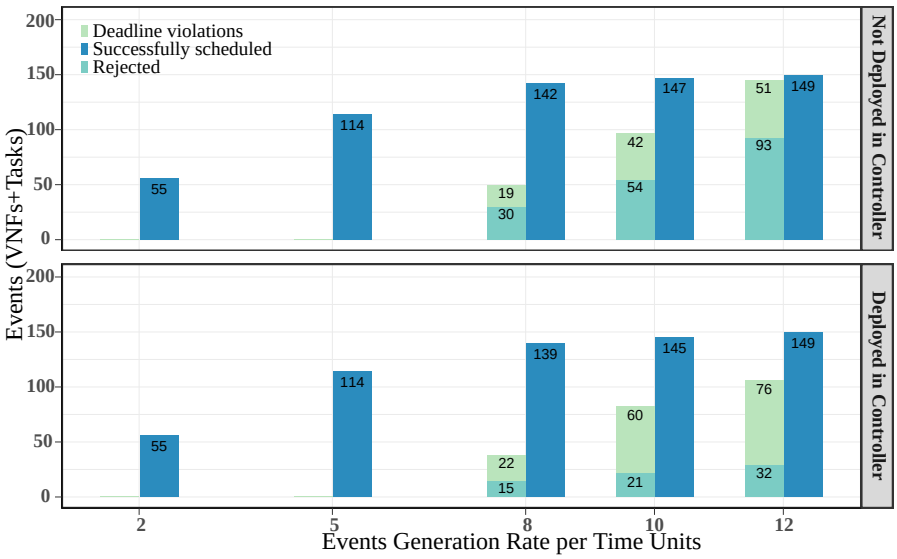
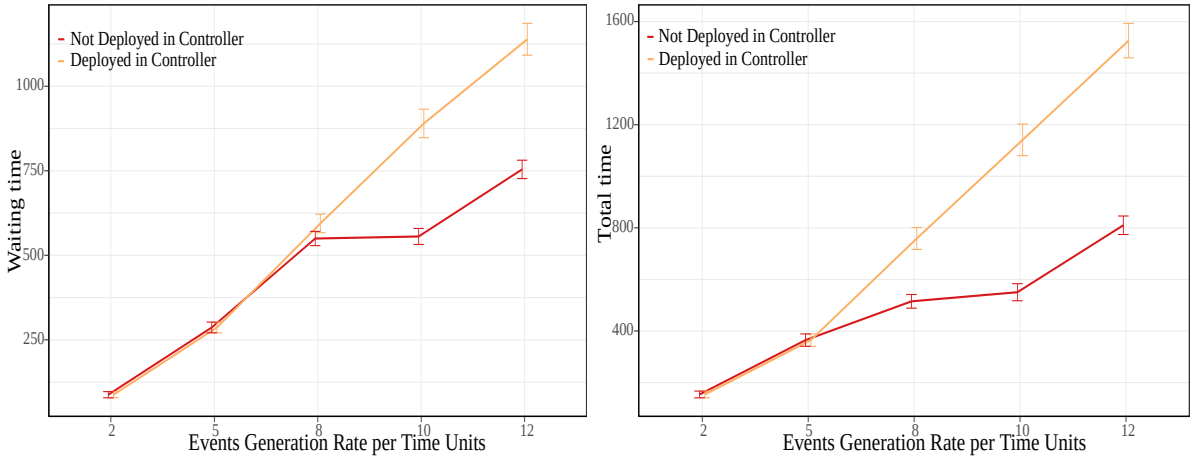


FIGURE 5.4. Number of deadline violations, rejected events, and successfully scheduled events.



(a) Waiting time.

(b) Total time.

FIGURE 5.5. Average metrics time while deploying or not deploying events in the controller node.

controller, and possible saturation in the system was thus avoided. Similar results were obtained for the average total time of the events since it includes the waiting time, as shown in Fig. 5.5(b).

We minimize rejections and effectively utilize all available resources using the controller node to deploy selected events. Figure 5.6 illustrates the average acceptance ratio of events for both criteria. For a generation rate of 2 and 5 events per time unit, both cases have an acceptance rate of 100%. However, the performances differed when the generation rates were greater than 5 events per time unit. More specifically, the events' acceptance ratio was increased by 11%, 28%,

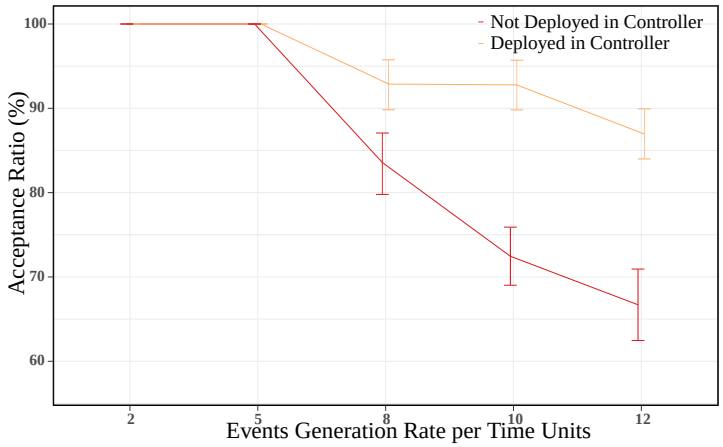


FIGURE 5.6. Average event acceptance ratio with and without deploying events in the controller node.

and 30% for generation rates of 8, 10, and 12 events per time unit, respectively, when deploying events in the controller node.

Based on these results, we confirm that using the controller node to deploy specific events guarantees a higher event acceptance ratio than using its resources solely for scheduling tasks. A significant reduction in rejected events proved the higher acceptance ratio. Nevertheless, it was at the cost of more significant deadline violations. Overall, the advantages outweigh the drawbacks when deploying services and tasks in the controller node (i.e., Kubernetes master node) in a resource-constrained environment. In this regard, the proposed scheduler has been implemented by taking these results into account.

5.3.2 SOC Regression Model

Several regression models were studied before starting the training phase to choose the one that best fits our study case. Figure 5.7 shows three polynomial models of first, second, and third orders, which employ the CPU usage as a predictor to estimate the SOC in a compute node. In this graphic, we included the adjusted R^2 parameter to describe the accuracy of the models. This parameter reflects the variation number of predictors and does not automatically increase when more predictors are added to the model. According to this metric, the best model was the third-order one, represented by the blue line. Nevertheless, given the negligible accuracy difference of the third-order model and the computation overload required, we used the second-order polynomial to balance accuracy and computation cost.

The experiments performed show that the controller node has a different behavior than the compute ones since its primary function is to schedule service and task requests. This role does not demand as much CPU usage as the execution of services and tasks. Figure 5.8(a) depicts the controller's behavior. Namely, the CPU usage reaches its higher value of around 1900 milli-CPU

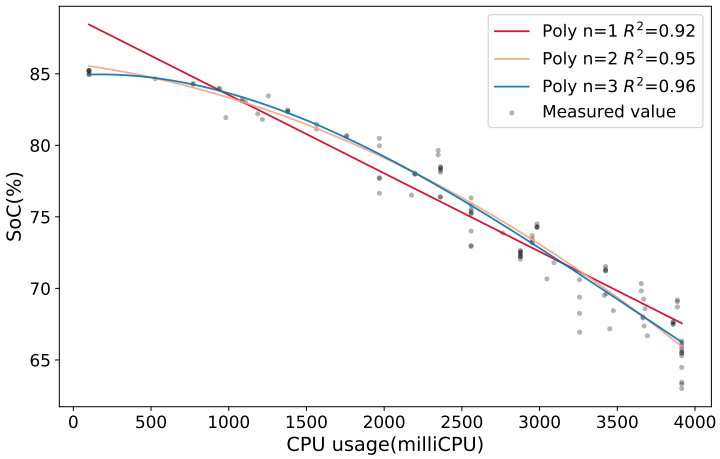
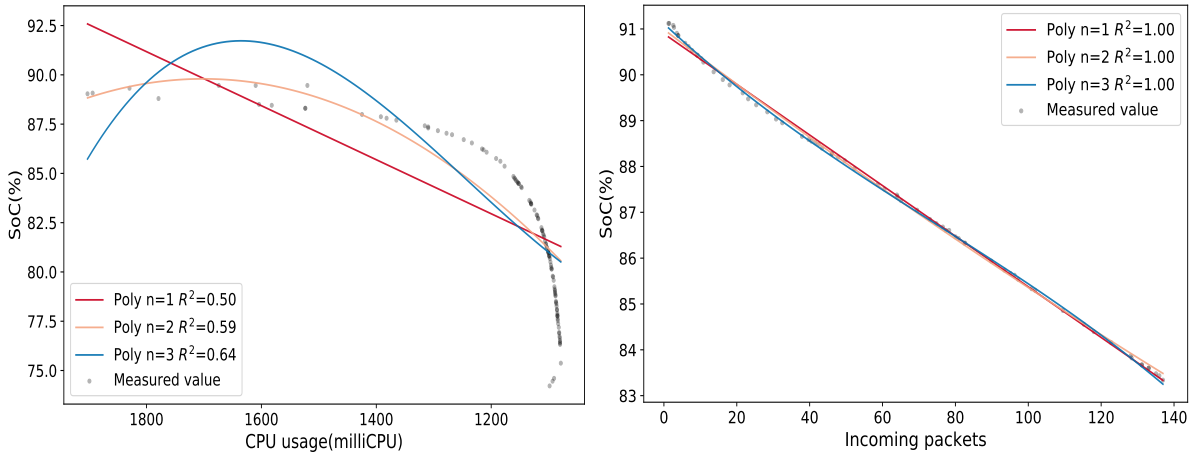


FIGURE 5.7. Multiple regression models for the SOC estimation based on CPU usage for a compute node.

and then starts to decrease while the SOC begins to decline. Considering the adjusted R^2 values, we can perceive that none of the analyzed models, based on CPU usage, fits the data correctly because the values were below 0.70.



(a) Model based on CPU usage.

(b) Model based on incoming packets.

FIGURE 5.8. Multiple regression models of the SOC for the control plane node.

Consequently, we analyzed another metric different from CPU usage. The selected metric was the number of incoming packets in the controller node since it receives user requests and worker messages. The polynomial models studied are shown in Fig. 5.8(b), which illustrates that the models perfectly fit the data as their adjusted R^2 was 1.

The CPU utilization represents a boundary parameter for the SBCs since a node cannot process new requests when reaching its maximum value, thus affecting its performance. Therefore,

we must also consider CPU usage in the regression model for the controller node. In this regard, Fig. 5.9 reflects a three-dimensional representation of the regression model based on incoming packets and CPU usage. The model coefficients are estimated by finding the minimum sum of squared deviations between the blue plane and the measured values. It should be noted that the least square regression line becomes a plane with two estimated slope coefficients when having two predictors. Considering the adjusted R^2, this model fits our data since it has the highest possible value.

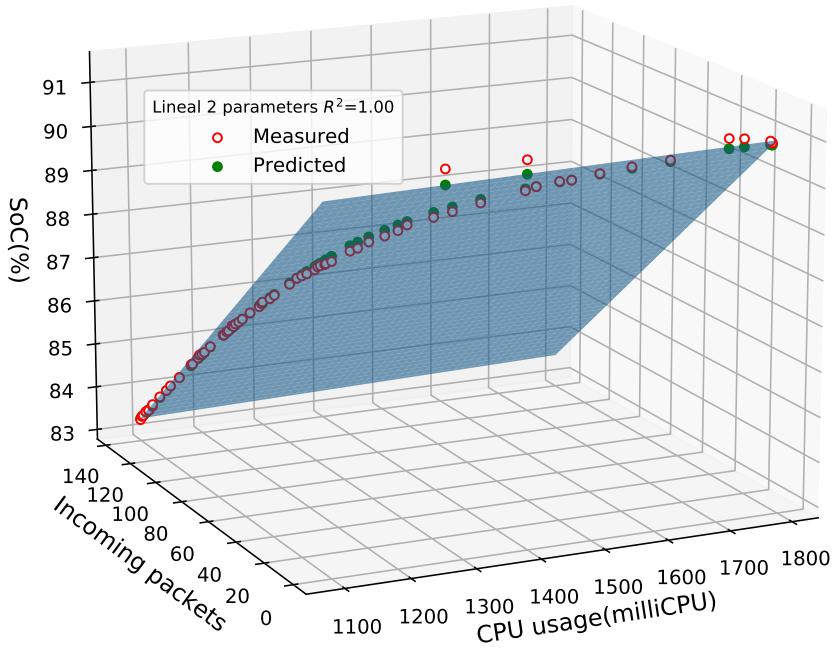


FIGURE 5.9. Lineal regression model with CPU usage and incoming packets as predictors for the controller node.

Finally, the SOC regression model for any node n in the SBC cluster ($n \in N$, where $N = N_c + 1$) can be expressed as follows:

$$SOC_{pred_{model}} = \delta_0 + \delta_1 \cdot pkt_{in_n} \cdot ctrl + \delta_2 \cdot cpu_n + \sum_{\forall i \in N_c} (\delta_{3_i} \cdot compute_i) \cdot cpu_i^2 + \delta_{4_i} \cdot compute_i \quad (5.15)$$

The model coefficients (δ) were obtained from the available training dataset. We introduced the categorical variable $compute_i$ to represent each compute node in the model ($i \in N_c$). The SOC of the node to be predicted takes a value of 1, and the others take a value of 0 (e.g., $compute_1$=1, $compute_2$=0, ..., $compute_{N_c}$=0). In the case of the controller node ($n = N$), all the categorical variables are 0 (e.g., $compute_1$=0, $compute_2$=0, ..., $compute_{N_c}$=0). Regarding the pkt_{in} term, we added a binary indicator $ctrl$ to distinguish whether the node is a controller or not. In this sense, it takes a value of 1 when the controller's SOC is predicted; otherwise, it is 0. Thus, our SOC regression model is transformed into a lineal model with two predictors (i.e., pkt_{in_n} and

cpu_n). In contrast, a second-order polynomial model based on CPU utilization was used for the compute nodes since the pkt_{in} term was discarded due to its low value regarding CPU usage.

5.3.3 Comparison among Scheduling Algorithms

To the best of our knowledge, no other online scheduler has been built for improving the acceptance ratio of requested events in an energy-constrained SBC cluster. Therefore, our proposed scheduling algorithm, SOCCS, was compared with the well-known greedy least loaded scheduler (LLS) and the native Kubernetes scheduler (KS). The former algorithm allocates the different events to the node with the highest available buffer capacity. In this manner, the node with the least CPU usage is chosen. In the case of the KS algorithm, it ranks the feasible nodes and selects the one with the highest-ranking score. In other words, the more resources the services and tasks use in a node, the lower its ranking is. These algorithms were analyzed through the following metrics: scheduled and rejected events, deadline violations, waiting and total time, acceptance ratio, and battery consumption. We ran several experiments for each generation rate to ensure the reliability of the results. They show a confidence interval of 95% for all the analyzed scheduling algorithms.

5.3.3.1 Average Number of Scheduled and Rejected Events

Figure 5.10 presents the results in terms of requested, scheduled, and rejected events (i.e., tasks and services). We note that LLS and SOCCS achieved similar results when the generation rate was low (i.e., 2 and 5) since they deployed all the requested services (rejected services were 0). In contrast, the KS had around 2 rejected services (red bar) for a generation rate of 5 events per unit time. We also observed that KS rejected around 2 tasks (salmon bar) for high generation rates (i.e., 8, 10, and 12), while the other algorithms deployed all the requested tasks (mid-blue bar).

As mentioned in Section 5.1.1, the generated services are formed by several network functions (VNFs). Figure 5.11 shows the results for their constituent VNFs. Analyzing the rejected VNFs for low-value generation rates shows that KS rejected around 8 VNFs (apricot bar) for 5 events per time unit. In contrast, the other approaches could schedule all the requested network functions for low-value generation rates. Regarding the scheduled VNFs (windows blue bar), our scheduler outperformed LLS and KS algorithms by 19% and 17%, 8% and 7%, and 5% and 2%, respectively, for the high-value generation rates. Furthermore, SOCCS reduced the rejected VNFs with respect to LLS by 11%, 12%, and 12% for 8, 10, and 12 events per time unit, respectively. For the same generation rates, these reductions were higher than KS, which was demonstrated by the values up to 16%, 23%, and 18%.

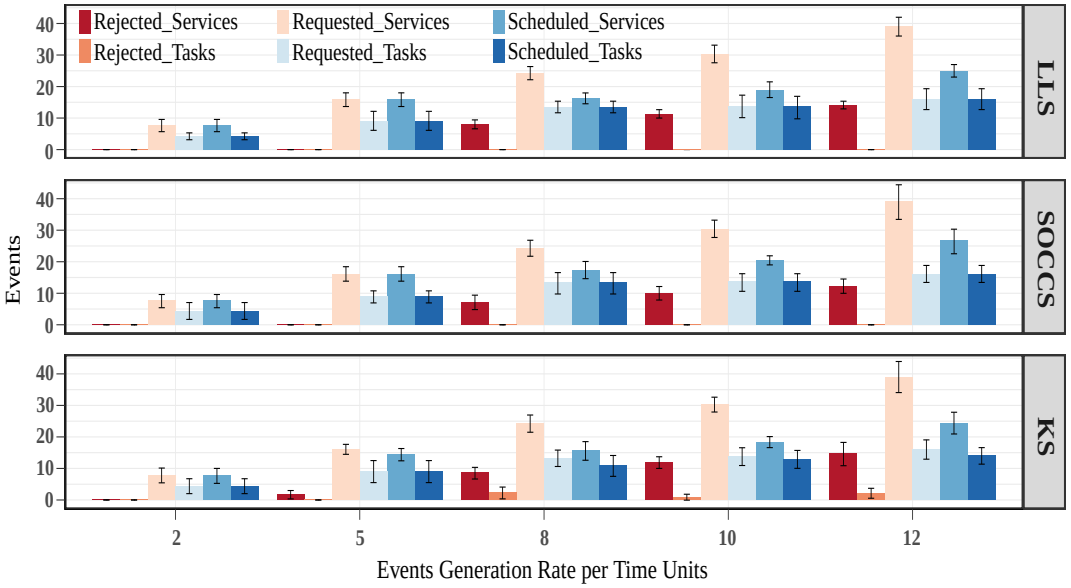


FIGURE 5.10. Number of requested, scheduled, and rejected events (i.e., services and tasks) for all the scheduling algorithms.

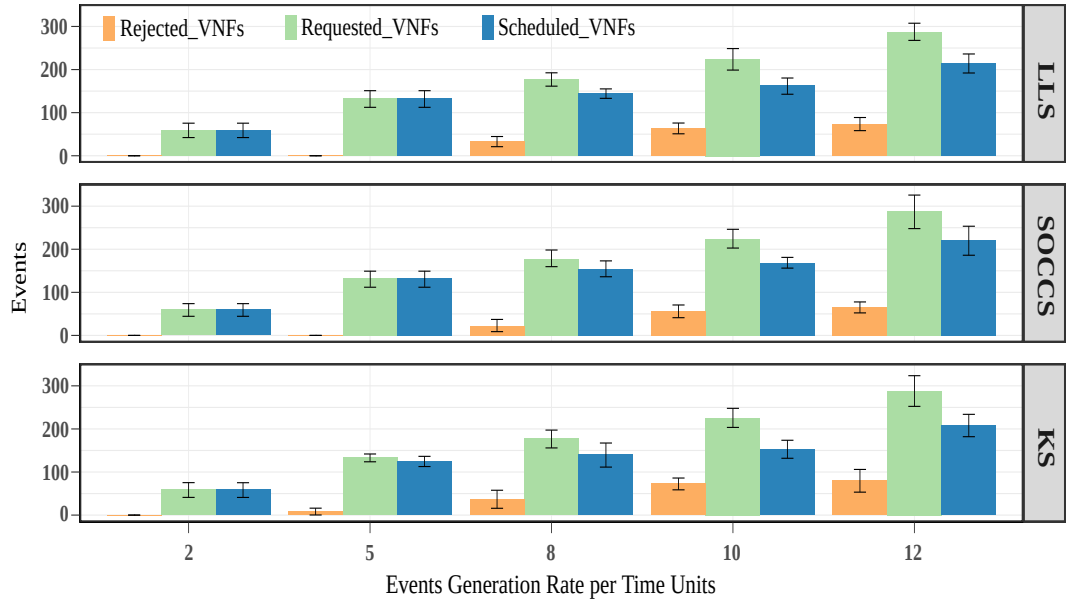


FIGURE 5.11. Number of requested, scheduled, and rejected VNFs for all the scheduling algorithms.

5.3.3.2 Average Acceptance Ratio

The acceptance ratio denotes the proportion between scheduled and requested events. Figure 5.12 depicts the average value of this metric for each generation rate. We observed that KS had the

worst performance for all the generation rates (e.g., its worst value is above 72%). The other algorithms (i.e., LLS and SOCCS) had similar results for low-value generation rates. Nevertheless, their differences were apparent for arrival rates of 8 or more events per time unit. Our proposed algorithm increased the acceptance ratio by around 2% with respect to LLS. Thus, SOCCS had the best performance.

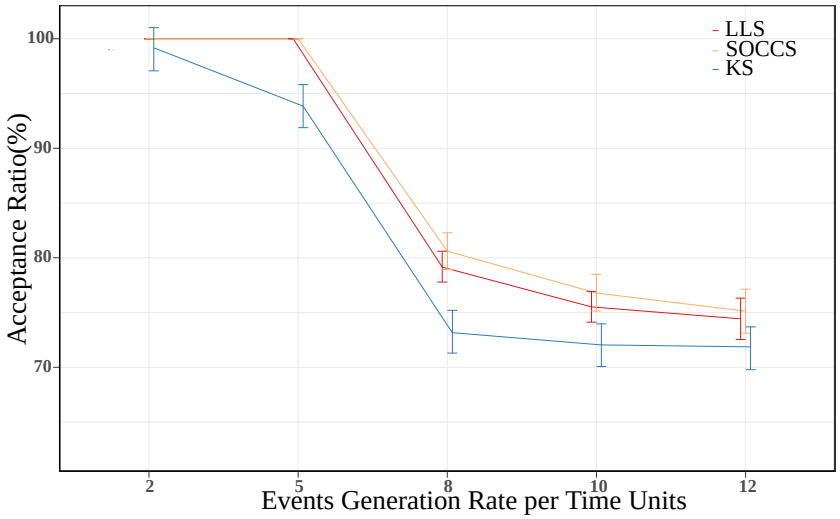


FIGURE 5.12. Event acceptance ratio for each scheduling algorithm.

5.3.3.3 Average Number of Successfully Scheduled Events and Deadline Violations

Figure 5.13 provides a deeper insight into the number of scheduled events since it separates the events with deadline violations from the successful ones. Our algorithm generally avoided more deadline violations than the others for all the generation rates. For the highest generation rate (i.e., 12 events per time unit), SOCCS decreased the number of deadline violations by 75% and 83% compared with LLS and KS, respectively.

5.3.3.4 Average Waiting and Total Time

As shown in Section 5.3.1, the values of the waiting and total times when the scheduler deploys events in the controller node are higher than the case in which they are not placed in this node. An analysis of Fig. 5.14 affirms that SOCCS had the lowest increment for both metrics. More specifically, our scheduler reduced the waiting time for the highest generation rate by 42% and 53% with regard to LLS and KS, respectively (see Fig. 5.14(a)). Additionally, SOCCS also decreased the total time by 34% and 53% in comparison with LLS and KS, respectively (see Fig. 5.14(b)).

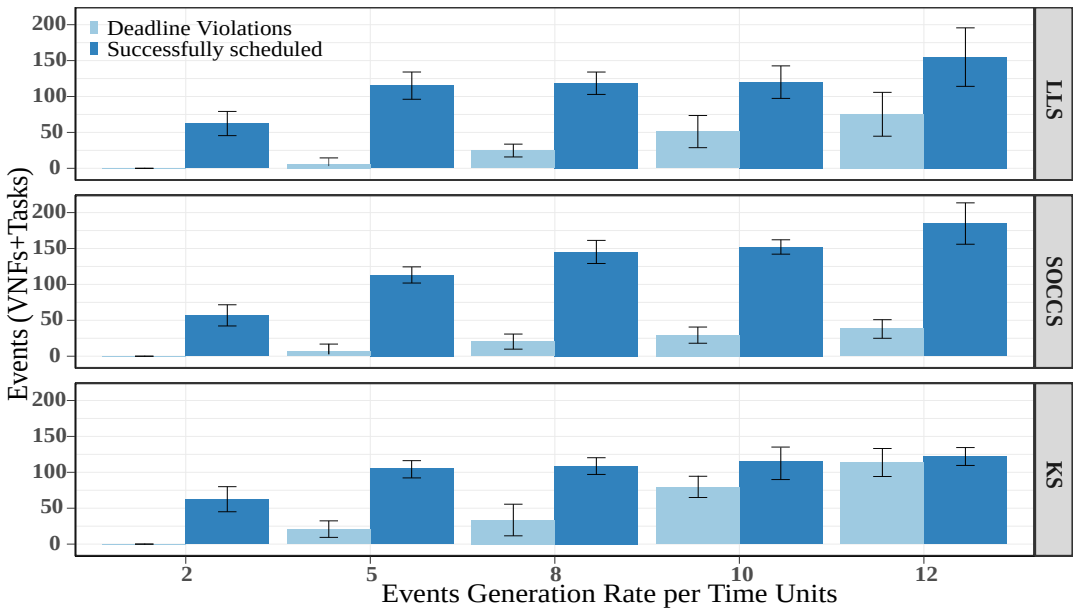
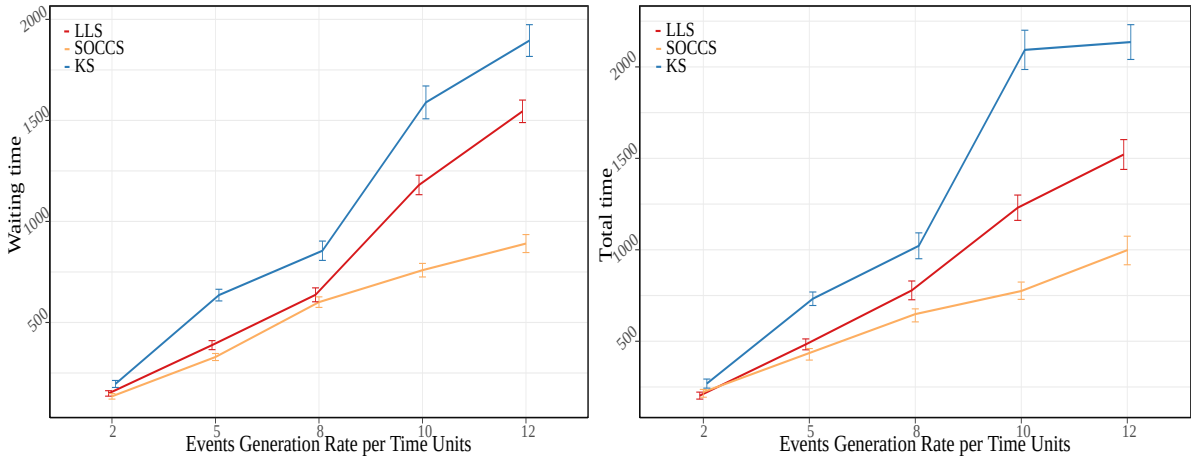


FIGURE 5.13. Number of successfully scheduled events and deadline violations for each scheduling algorithm.



(a) Waiting time.

(b) Total time.

FIGURE 5.14. Time metrics for all the scheduling algorithms.

5.3.3.5 Average Battery Consumption

The battery consumption was obtained by calculating the difference between the initial and final measured SOC in each experiment. Figure 5.15 depicts the average battery consumption for each node when running different scheduling algorithms to deploy events.

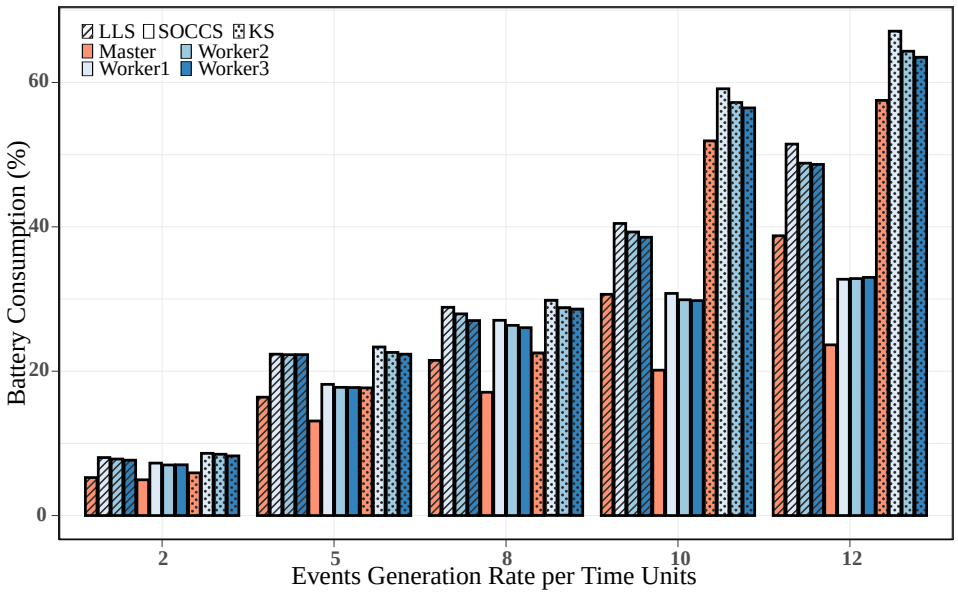


FIGURE 5.15. Battery consumption for each node while running different scheduling algorithms.

In general, the greater the event arrival rate, the higher the battery consumption. Comparing the three algorithms shows that the KS had the highest battery consumption for all the generation rates. In contrast, SOCCS had the lowest battery consumption because it always selected the node with the highest score to assign the requested event. Namely, it chose the node with the maximum SOC and minimum CPU usage. In addition, Section 5.3.3.4 revealed that our proposed scheduler performs faster than the baseline algorithms because the total time of the events in the system was notably reduced.

Consequently, our proposal reduced the cluster's operation time to process the requested events, which indisputably had a lower impact on battery consumption than the other schedulers. Considering the highest generation rate, SOCCS saved up to 39% and 59% of the battery consumption in the controller node compared with LLS and KS, respectively. Similarly, it decreased the consumption in Worker1 by 36% and 51% concerning the same algorithms. Regarding the workers' imbalance in battery consumption, our proposed algorithm had the best performance. For a generation rate of 12 events per time unit, SOCCS registered an imbalance of 0.25 between the maximum and minimum average battery consumption. This value was much lower than those obtained in LLS and KS algorithms (2.82 and 3.61, respectively).

5.4 Conclusion

In this chapter, we proposed SOCCS as a scheduling algorithm to assign events to an SBC cluster by taking into account the predicted battery consumption and used resources (e.g., CPU

and memory utilization). Specifically, a regression model was used to make the predictions by establishing the relationship between the SOC and the CPU usage. The proposed scheduler was implemented and evaluated in a Raspberry Pi cluster running Kubernetes.

Unlike the revised literature, we analyzed the case of employing the unused controller resources to deploy specific events. The results revealed that such consideration is a good criterion in a resource-constrained environment. More particularly, we increased the acceptance ratio by around 30% for the highest generation rate when deploying events in the controller node. This increment in the acceptance ratio was also related to a significant reduction in the rejected events, up to 66% for the same generation rate. Consequently, our proposed scheduler could better use the available resources in the cluster.

When comparing the proposed solution with other algorithms, the results showcase that SOCCS outperformed the baseline algorithms with regard to analyzed metrics. Concretely, our proposal reduced the rejected VNFs up to 23% for the highest generation rate. Its high values of acceptance ratio highlight this result. Furthermore, the proposed scheduler decreased the number of deadline violations in the scheduled events by 83%. This outcome was corroborated by the low values in the waiting and total time of the events when using SOCCS. Finally, our proposed solution saved around 59% of battery consumption in the controller node and 51% in the compute ones for the highest generation rate, thus increasing the lifetime of the cluster's elements and, therefore, their resilience.

CHAPTER 6

DQN-based Intelligent Controller for Multiple Edge Domains

This chapter is based on:

- **A. Llorens-Carrodeguas**, C. Cervelló-Pastor, and F. Valera, "DQN-Based Intelligent Controller for Multiple Edge Domains," Submitted to a Journal, 2022.

This chapter presents an improvement of the solution proposed in Chapter 5 in that the described approach supports its execution in large-scale environments. Indeed, we introduce multi-cluster edge systems where event requests can be deployed across all the participant nodes. To do so, a global entity must gather the node's information (e.g., CPU, memory, and SOC) from the participant devices to select the best nodes where event requests can be deployed. In this regard, we incorporate an ML method to guarantee the best decisions for the proposed global entity, thus ensuring the system's resilience. The proposed strategy is aligned with the European project AI@EDGE [149], which works on artificial intelligent closed-loop automation mechanisms and distributed and decentralized connect-compute platforms, among several other breakthroughs.

The following sections explain the proposed solution in detail. Section 6.1 presents the problem statement and the notation and system model used in our proposal. It also explains the considered reference architecture that captures the environment characteristics mentioned above. We describe the intelligent controller solution and the functions of its modules in Section 6.2. Finally, the results are discussed in Section 6.3.

6.1 Problem Statement and System Model

This section presents the problem addressed as well as the notation and system model used.

6.1.1 NFV System Description

To capture the environment characteristics above and formally define the problem under investigation, we consider the reference architecture depicted in Fig. 6.1. Under this architecture, a global SDN controller is deployed to manage a region compounded by several cluster nodes. Thus, we denote K as a set of clusters ($k \in K$). Each cluster k is formed by a set of physical nodes (N) where VNFs (f) can be scheduled. The cluster nodes are considered energy-constrained devices, representing edge nodes with a fixed amount of computational resources.

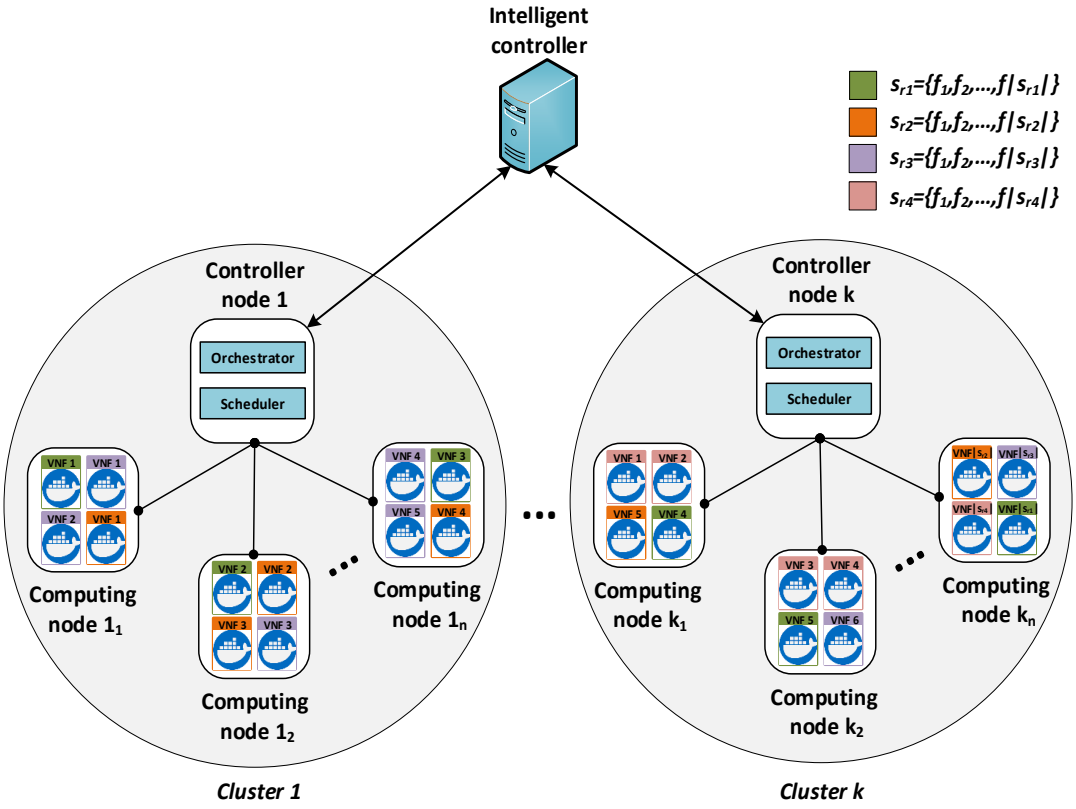


FIGURE 6.1. Reference NFV system formed by several clusters of edge nodes which an intelligent controller manages.

The virtual functions (f) are placed in a set of deployable units within computing nodes (P). Each virtual node $p \in P$ is identified with an ID. We include a global parameter $p_i^{k_n}$ to indicate that virtual node p_i has been placed in physical node n of cluster k, where $n \in N$ and $k \in K$.

Each physical node n in cluster k has resource and power capacities. The former contains the computing resources (e.g., CPU and memory). The latter includes the power source that keeps the

device working. Therefore, we denote the resource capacity of each node by $C_{k_n} = (C_{k_n}^{CPU}, C_{k_n}^{Mem})$, which represents the available resources concerning CPU and memory. Moreover, we indicate the available energy capacity of each node as E_{k_n}, which is expressed in terms of the SOC. Additionally, each node has a total output bandwidth represented as W_{k_n}.

Similar to the considerations of Chapter 5, the system can process two types of events, tasks and services, although we focus on the latter because it represents the most complex case since several virtual functions compound the services. More specifically, this kind of event is comprised of a sequence of VNFs, $F = \{f_1, f_2, \ldots, f_{|F|}\}$.

In this regard, each function instance $f \in F$ has a resource demand in terms of CPU and memory denoted by $D_f = (D_f^{CPU}, D_f^{Mem})$.

We denote S_R as a set of service requests arriving at one of the controller nodes in the system. Each request $s_r \in S_R$ must be directed through the VNFs compounding the request by considering its requirements. The service-related s_r is expressed as follows:

$$s_r = \{f_1, f_2, \ldots, f_{|s_r|}\}, f_i \in F, i = 1, 2, \ldots, |s_r|.$$

Each service request s_r has specific QoS demands, such as the bandwidth requirement W_r and deadline d_r for processing the given request. In addition, each VNF f in the request has a running time parameter (t_r) to show the time that must pass before considering it complete. When $t_r > 0$, the event runs during the specified time. Otherwise, the event will be executed during the system's lifetime when this parameter is 0 or not specified.

Finally, for each request $s_r \in S_R$, we use a binary variable x_{s_r,k_n}^i to indicate the placement decision of each VNF f_i belonging to the requested service. More specifically, $x_{s_r,k_n}^i = 1$ when VNF f_i is successfully placed on node $n \in N$ of cluster $k \in K$; otherwise, $x_{s_r,k_n}^i = 0$. Table 6.1 provides a list of notations related to the system model.

6.1.2 Problem Modeling

To address real-time network variations due to requests received with an unknown arrival time, we use the concept of time slot, denoted by τ. The controller verifies the nodes' status at each time slot τ. Additionally, it can receive service requests, make deployment decisions, and update the network's states. In this regard, we define $C_{n,\tau}$ as the available capacity of node n at time slot τ. Likewise, $W_{n,\tau}$ represents the available bandwidth of node n at time slot τ.

We further define $S_{R,\tau} \subset S_R$ to address the possibility of receiving several requests at time slot τ. Thus, simultaneous service requests can be treated as they arrive by considering the event's ranking in the priority queue. The event's ranking is calculated based on two factors: $delay(f)$ and $wait_{queue}(f)$. The former represents the time that the virtual function's execution f can be delayed without missing its deadline (see Equation (6.1)). The latter represents the waiting time of the VNF f in the queue (see Equation (6.2)). In this equation, t_a reflects the

Table 6.1: System model notation.

Notation	Description
K	Set of clusters
N	Set of physical nodes where events can be placed
P	Set of deployable units in the computing nodes
S_R	Set of network service requests arriving at controller
F	Sequence of VNFs compounding a network service request
k	Each cluster formed by a set of physical nodes (N)
n	Each physical node where virtual nodes are created
p	Each virtual node created on the physical node to run the events
$p_i^{k_n}$	Indicates that virtual node p_i is placed in node n of cluster k
s_r	Each network service request formed by a sequence of VNFs
f_i	Each VNF compounding a network service
C_{k_n}	Available resource capacity of node $n \in N$ of cluster $k \in K$
E_{k_n}	Available energy capacity of node $n \in N$ of cluster $k \in K$
D_f	Resource demand of service instance $f \in F$
W_{k_n}	Total output bandwidth of node $n \in N$ of cluster $k \in K$
W_r	Bandwidth requirement of service request $s_r \in S_R$
d_r	Deadline for processing a given request
t_r	Running time of a given request before considering it complete
t_a	Arrival time of a given request to controller
t_s	Starting time when the selected nodes process a given request
$x^i_{s_r,k_n}$	1 if the VNF f_i is successfully deployed on node $n \in N$ of cluster $k \in K$, 0 otherwise
$a_{s_r,\tau}$	1 if service request $s_r \in S_R$ is active in time slot $[t_s, t_s + t_r]$, 0 otherwise
$\eta^f_{k_n,\tau}$	Number of VNF instances $f \in F$ that are placed on node $n \in N$ of cluster $k \in K$ in time slot $[t_s, t_s + t_r]$
$\nu_{k_n,\tau}$	1 if any VNF instance is placed on node $n \in N$ of cluster $k \in K$ in time slot $[t_s, t_s + t_r]$, 0 otherwise
$z^{s_r}_{k_n,\tau}$	1 if any VNF of request $s_r \in S_R$ is placed on node $n \in N$ of cluster $k \in K$ in time slot $[t_s, t_s + t_r]$, 0 otherwise
y_{s_r}	1 if request $s_r \in S_R$ is deployed, 0 otherwise

arrival time of the service request.

$$delay(f) = f_{d_r} - t_{now} - f_{t_r} \tag{6.1}$$

$$wait_{queue}(f) = t_{now} - f_{t_a} \tag{6.2}$$

Considering the previous definitions, we calculate the ranking score for the virtual function (f_{rank}) as follows:

$$f_{rank} = \beta_1 \cdot delay(f) - (1 - \beta_1) \cdot wait_{queue}(f) \tag{6.3}$$

To represent whether request $s_r \in S_R$ is still in service at time slot τ, we use the binary variable $a_{s_r,\tau}$ as follows:

$$a_{s_r,\tau} = \begin{cases} 1 \ , \ t_s \leq \tau < (t_s + t_r) \\ 0 \ , \ \text{otherwise} \end{cases} \tag{6.4}$$

where t_s represents the starting time when the selected nodes start processing the service request.

Since we consider multiple-instance deployments of VNFs in the same node, we must know the number of VNFs $f \in F$ placed on node $n \in N$ of cluster $k \in K$ at time slot τ. For this, we utilize the variable $\eta^f_{k_n,\tau}$ to reflect this value (see Equation (6.5)).

$$\eta^f_{k_n,\tau} = \sum_{\forall s_r \in S_R} \sum_{1 \leq i \leq |s_r|}^{f} x^i_{s_r,k_n} \cdot a_{s_r,\tau} \tag{6.5}$$

Similarly, we indicate when any VNF instance is placed at time slot τ on node $n \in N$ of cluster $k \in K$ through the binary variable $\nu_{k_n,\tau}$ as follows:

$$\nu_{k_n,\tau} = \begin{cases} 1 \ , \ \sum\limits_{\forall f \in F} \eta^f_{k_n,\tau} > 0 \\ 0 \ , \ \sum\limits_{\forall f \in F} \eta^f_{k_n,\tau} = 0 \end{cases} \tag{6.6}$$

After these specifications, we can formally present our model, defined as $< \mathbf{S}, \mathbf{A}, \mathbf{R}, \gamma >$. In this model, $\mathbf{S}$ represents the set of discrete states, $\mathbf{A}$ is the set of discrete actions, $\mathbf{R}$ is the reward function, and $\gamma \in [0,1]$ is a discount factor for future rewards. Thus, the core elements used in the model are defined as follows:

State: The state $s \in \mathbf{S}$ at time t (s^t) is represented as a vector consisting of the remaining resources and bandwidth of each node and the requirements of the current VNF to be placed. Thus, state s^t is defined as:

$$s^t = (C^t, E^t, W^t, R^t),$$

where C^t defines the remaining resources of each node; therefore $C^t = (C^t_1, C^t_2, \ldots, C^t_{|N|})$. In addition, the remaining SOC of each node is described as $E^t = (E^t_1, E^t_2, \ldots, E^t_{|N|})$. The remaining bandwidth of each node is represented by $W^t = (W^t_1, W^t_2, \ldots, W^t_{|N|})$. Finally, the requirements of the VNFs to be scheduled are defined as $R^t = (D_i, W_{r_i}, t_{r_i}, d_{r_i}, PF_{r_i})$, where D_i are the VNF's demanded resources on the node, W_{r_i} represents the bandwidth demand, the running time of the service request is established by t_{r_i}, d_{r_i} defines the deadline for processing the given request, and PF_{r_i} is the number of VNFs in $s_r \in S_R$ waiting to be deployed.

Action: We denote action $a \in \mathbf{A}$ as a binary vector. In more detail, each position in the vector corresponds to a possible action. When the first position is 1, it indicates that no node will be selected (i.e., $a^t = [1,0,0,0,...,0]$ do nothing). The value of 1 in one of the remaining positions implies the selected node (i.e., $a^t = [0,0,1,0,...,0]$).

Reward Function: We define the reward function as the weighted sum of objectives that we want to achieve jointly. The mathematical representation of this function is explained in the following section.

State Transition: The state transition is defined as (s^t, a^t, r^t, s^{t+1}), where s^t is the current network state, a^t is the action taken (i.e., do nothing or place VNF), and s^{t+1} is the new network state after receiving the reward r^t.

6.1.3 Problem Formulation

This section presents the mathematical formulation of the multi-cluster service deployment problem, which considers all the clusters' nodes to deploy the constituent virtual functions of a service request.

We consider that multiple VNFs belonging to different service requests can be placed at the same node in time slot τ while it has available resources. Such consideration is known as VNF consolidation [150–152] and is represented as follows:

$$\sum_{\forall f \in F} \eta^{f}_{k_n,\tau} \cdot D_f \leq C_{k_n} \tag{6.7}$$

In addition, we use $z^{s_r}_{k_n,\tau}$ to indicate that any VNF of request $s_r \in S_R$ are placed in node $n \in N$ of cluster $k \in K$ at time slot τ. Then:

$$z^{s_r}_{k_n,\tau} = \begin{cases} 1 \ , \ \sum\limits_{i=1}^{|s_r|} x^{i}_{s_r,k_n} \cdot a_{s_r,\tau} > 0 \\ 0 \ , \ \sum\limits_{i=1}^{|s_r|} x^{i}_{s_r,k_n} \cdot a_{s_r,\tau} = 0 \end{cases} \tag{6.8}$$

Thus, we establish that the bandwidth demand of all requests passing through node $n \in N$ of cluster $k \in K$ cannot exceed its total output bandwidth, represented as follows:

$$\sum_{\forall s_r \in S_R} W_r \cdot z^{s_r}_{k_n,\tau} \leq W_{k_n} \tag{6.9}$$

Similar to Chapter 5, we contend that surpassing the deadline of the service request would not lead to a service rejection since maintaining a required QoS is a desirable metric but not mandatory [94]. In this vein, we use a binary variable y_{s_r} to indicate whether s_r is deployed or not. Then:

$$y_{s_r} = \begin{cases} 1 \ , \ \sum\limits_{i=1}^{|s_r|} \sum\limits_{n \in N} x^{i}_{s_r,k_n} = |s_r| \\ 0 \ , \ \sum\limits_{i=1}^{|s_r|} \sum\limits_{n \in N} x^{i}_{s_r,k_n} < |s_r| \end{cases} \tag{6.10}$$

By accounting for the previous considerations, we aim to achieve several objectives. First, we intend to minimize the resource cost of used nodes ($\mathbb{RC}$), which can be expressed as follows:

$$\begin{aligned} &min \sum_{k \in K} \sum_{n \in N} v_{k_n,\tau} \cdot (\beta_C C_{k_n} + \beta_W W_{k_n}), \\ &\text{s.t. (4), (5), (6), (7),} \end{aligned} \tag{6.11}$$

where β_C and β_W are the node resource and bandwidth unit costs, respectively.

Our second objective is to maximize the system's lifetime ($\mathbb{LT}$) by selecting nodes with appropriate levels of SOC to save as much energy as possible. Thus,

$$\begin{aligned} &max \sum_{k \in K} \sum_{n \in N} E_{k_n}, \\ &\text{s.t. (1), (2), (4), (5), (6), (7), (8), (9).} \end{aligned} \tag{6.12}$$

Our third objective is to maximize the number of deployed services ($\mathbb{DS}$) to benefit the customer's requests. This objective is expressed as follows:

$$\begin{aligned} &max \sum_{s_r \in S_R} y_{s_r}, \\ &\text{s.t. (1), (2), (4), (5), (6), (7), (8), (9).} \end{aligned} \tag{6.13}$$

Once the objectives are determined, we can define the reward function as a weighted sum of the resource costs, the lifetime of the system, and the number of deployed service requests, as expressed in Equation (6.14).

$$r^t(s^t,a^t) = \begin{cases} \zeta \cdot \mathbb{LT} + \xi \cdot \mathbb{DS} - \phi \cdot \mathbb{RC}, & s_{r_i} \text{ or } f_i \text{ is deployed} \\ 0 & , s_{r_i} \text{ or } f_i \text{ is rejected} \end{cases} \tag{6.14}$$

The adjustable positive weights $\zeta, \xi, \phi \in [0,1]$ allow a trade-off between the different deployment decisions.

6.2 DQN-Based Intelligent Controller Solution

In this section, we propose our deep Q-network (DQN)-based intelligent controller (DQNIC) to deploy virtual functions of a service request among several clusters. After receiving a multi-deployed service request from one of the assigned clusters, the proposed solution selects the best nodes among all the clusters by considering remaining battery estimations and CPU usage. As shown in Fig. 6.2, each cluster has a scheduler in charge of deploying local service requests by using the SOC and capacity-based scheduler (SOCCS) presented in Chapter 5. Since our proposal requires the local schedulers' available information (e.g., CPU, memory, and SOC) to deploy services by considering a multi-cluster placement, we incorporate a mechanism in the local schedulers to guarantee their communication with the intelligent controller. This mechanism is also proposed for the intelligent controller and is based on DDS (see Chapter 3). In this way, our proposal is formed by five main elements: the global scheduler, the global monitor, the data writer, the data reader, and the DDS monitor. These modules have been grouped into two categories according to their functions: DQN-based scheduler and global DDS.

Figure 6.2 depicts the relationships among the constituent modules of the proposed solution. The orange line represents the DDS communication between global and local entities as well as their relationship with the schedulers' blocks; the blue line reflects the connection between the schedulers' elements and how the local schedulers communicate with the orchestrator. Finally, the green line shows the interaction between the orchestrator and the cluster nodes. The functions of the DQNIC's blocks and local DDS are explained in the following sections.

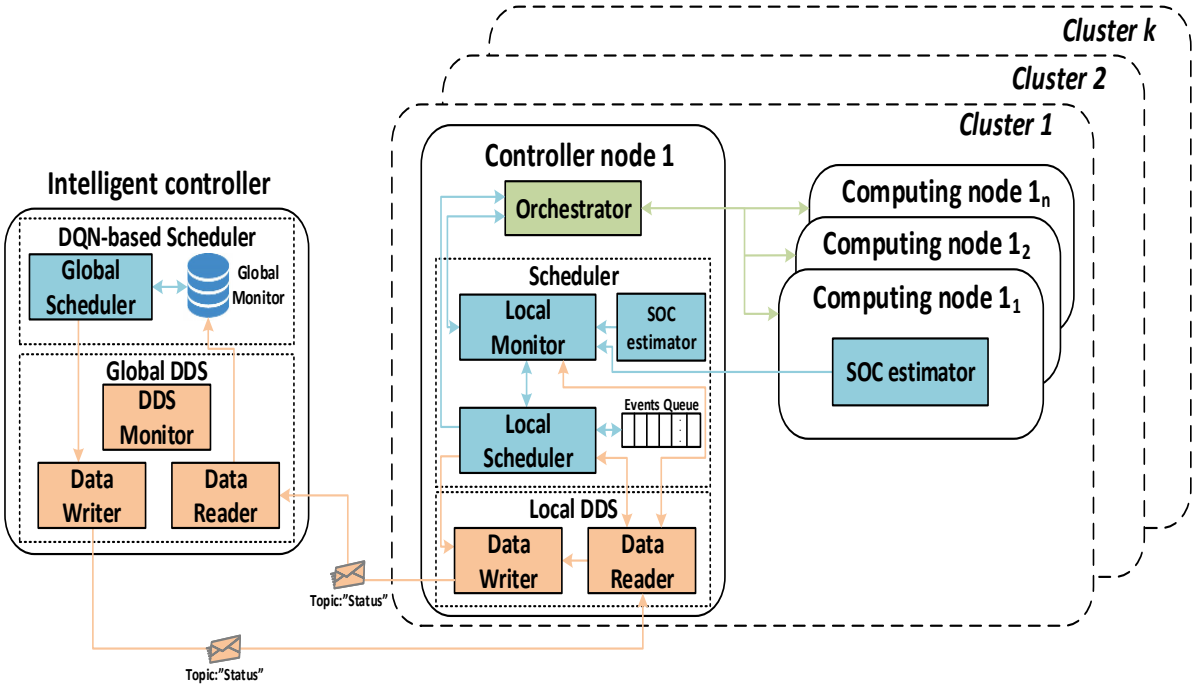


FIGURE 6.2. Intelligent controller solution design and the relationships among its constituent modules.

6.2.1 Local DDS

The local DDS implementation is composed of two blocks: *data reader* and *data writer*. The former reads the information sent by the DQNIC regarding the selected computing node where the controller must deploy the current VNF. The controller node of a cluster will place a VNF in its domain when it detects its name in the identifier field of the "Status" topic. This topic represents a data stream composed of eight fields where the identifier one is used to announce the kind of information that the *local data writer* will send. This element is responsible for publishing data regarding the nodes and service status by sending information about service requests, service and VNF deployments, and service rejections. Additionally, the *local data writer* periodically shares the remaining battery and CPU utilization of the cluster's nodes.

6.2.2 Global DDS

In contrast to the local DDS, the global implementation is formed by three blocks: *DDS monitor*, *data writer*, and *data reader*. The monitor automatically discovers compatible controller nodes according to the configured security profiles, guaranteeing that the DQNIC only receives information from registered entities during the discovery phase (see Chapter 3). Moreover, the *DDS monitor* tracks the controller nodes' status by detecting either a new participant or a failed one.

The *global data writer* is triggered by the *global scheduler* to send, through the "Status" topic, the selected node where the analyzed VNF must be placed. By indicating the controller node that manages the selected compute in the identifier field, it allows the reception of its decision only by the required cluster's master node.

In correspondence with the information sent by the *local data writer*, the *global data reader* collects the nodes' status in terms of CPU, memory usage, and SOC estimation. It also receives multi-deployment service requests and several placement states. In addition to gathering the stated information, the *global data reader* is responsible for storing it on the corresponding global monitor's buffer for further use during the DQN training.

6.2.3 Global Monitor

This block represents a collection of storing data structures used by the different constituent elements of DQNIC. More specifically, the *global data reader* stores the received information related to the nodes' status and service requests in this element. The *global scheduler* reads the stored data from the *global monitor* to create the input data forwarded through the DQN. Moreover, this block is used to save the state transitions (i.e., s^t, a^t, r^t, s^{t+1}), which are utilized afterward to train the DQN algorithm.

6.2.4 Global Scheduler

This element represents the most crucial component in the DQNIC solution. It determines the best node where a particular VNF can be placed according to the system's current state. In this manner, the *global scheduler* chooses what it considers the best action. This element improves its decisions by considering past experiences and obtaining rewards from the system after each action. This behavior is due to the RL algorithm running on this element.

A well-known RL solution is Q-learning, which selects actions according to the Q-values stored in its dimensional Q-table. This solution's main weakness is its lack of generality and scalability despite its powerful and straightforward capabilities [153]. Therefore, this solution is inapplicable in large-scale networks.

The DQN introduces a neural network to estimate the Q-value function to overcome this limitation. The architecture of this network is formed by an input layer, an output layer, and several hidden layers, as shown in Fig. 6.3. The input layer comprises the state vector, and the output layer is the actions' probability distribution.

By considering a DQN model to estimate the possible actions that DQNIC can take, we implement the primary process of *global scheduler* as shown in Algorithm 6.1. This algorithm runs simultaneously with the communication process managed by the *global DDS*. Thus, we avoid delays in the algorithm's execution due to the performance of other modules.

The primary function of Algorithm 6.1 is to select the best node in which to place a VNF request by considering an input state. To this end, we created a DQN network according to the

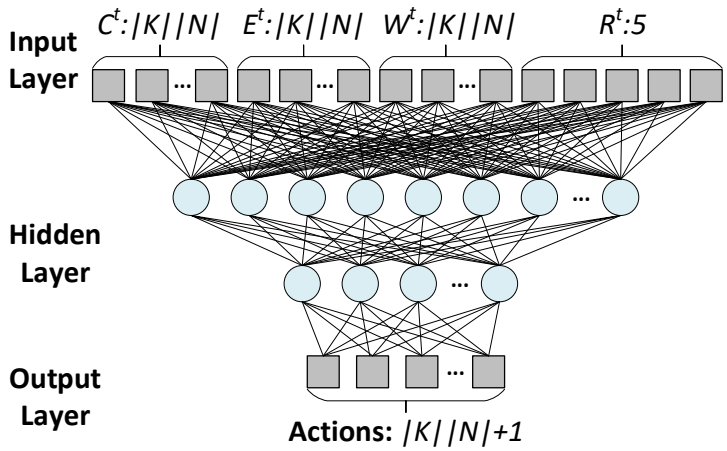


FIGURE 6.3. Neural network design to estimate the Q-value function.

Algorithm 6.1: DQNIC training process.

1 Create a DQN model according to the number of inputs and outputs or load a pre-trained model
2 Initialize action-value function Q with random weights Θ
3 Initialize target action-value function $\hat{Q}$ with weights $\Theta^- = \Theta$
4 Initialize action space ($|K||N|+1$), RL-agent and replay buffer (D) with a determined size
5 **while** $trainSteps < exploreSteps$ **do**
6 Create input state (nodes' status and VNF request) using the stored information in *global monitor*
7 **if** *RL-agent's strategy is exploration* **then**
8 RL-agent selects a random action a^t from action space with probability ϵ
9 **else**
10 RL-agent selects action $a^t = max_a Q(s^t, a; \Theta)$
11 Trigger *global datawriter* to indicate VNF's deployment to the selected node in action a^t
12 Wait for acknowledgment of VNF's deployment from selected node
13 Calculate reward r^t using Equation (6.14)
14 Get next state (s^{t+1}) and store transition (s^t, a^t, r^t, s^{t+1}) in D
15 **if** $envSteps > observedSteps$ **then**
16 Sample random mini-batch of transitions (s^j, a^j, r^j, s^{j+1}) from D
17 **forall** *Transitions in mini-batch* **do**
18 **if** *Episode terminates at step* $j+1$ **then**
19 Set $y^j = r^j$
20 **else**
21 Set $y^j = r^j + \gamma * max_{a'} \hat{Q}(s^{j+1}, a'; \Theta^-)$
22 Perform gradient decent step on $(y^j - Q(s^j, a^j; \Theta))^2$ with respect to the network parameters Θ
23 Every X steps reset $\hat{Q} = Q$

number of inputs (e.g., node's status and VNF request) and outputs (e.g., set of clusters' nodes in the system) in the system's model (line 1). This step also includes the possibility of loading a pre-trained model if one exists. Later, we initialize the DQN model's parameters with random

weights (lines 2 and 3). It should be noted that we use two DQN networks (i.e., double DQN model although we called DQN model for convenience) to make our training more stable since the values of $Q^t(s^t, a^t)$ and $Q^t(s^{t+1}, a)$ provided by the Bellman equation [154] have only one step between them. Thus, it is difficult for a neural network to distinguish between them, which can alter the estimation values of nearby states after updating the network's parameters. Therefore, we introduce the so-called target Q-network to back-propagate its predicted Q values and train the main Q-network with fixed weights to stabilize the computation of $maxQ(s^{t+1}, a)$ term in the Bellman equation. Following the algorithm's execution, we define the set of possible actions to take, which includes the existing nodes of the system plus the possibility of doing nothing (line 4). Moreover, this step comprises the initialization of the RL agent and the replay buffer, where the system's transitions will be stored.

In line 5, we define the running condition of this algorithm by indicating the number of training steps we want to consider. During this process, the algorithm creates the input state for each interaction with the environment by considering the stored information in *global monitor* (line 6). To make the model easier to train, we use the input data normalization method to format the input data into a small range (i.e., $[0,1]$). For the computing resource inputs (e.g., CPU and SOC) in state s^t, we divide the remaining capacities of each node by a maximum $C_{max} = max(C_{k_n})$, $E_{max} = max(E_{k_n})$, $\forall k \in K, n \in N$. Similarly, we compress the bandwidth-related inputs and the rest of the VNF's requirements. Thus, the normalized input state is:

$$\left(\frac{C^t_{1_1}}{C_{max}}, \ldots, \frac{C^t_{1_{|n|}}}{C_{max}}, \ldots, \frac{C^t_{|k|_{|n|}}}{C_{max}}, \frac{E^t_{1_1}}{E_{max}}, \ldots, \frac{E^t_{1_{|n|}}}{E_{max}}, \ldots, \frac{E^t_{|k|_{|n|}}}{E_{max}}, \right.$$
$$\left. \frac{W^t_{1_1}}{W_{max}}, \ldots, \frac{W^t_{1_{|n|}}}{W_{max}}, \ldots, \frac{W^t_{|k|_{|n|}}}{W_{max}}, \frac{D_i}{C_{max}}, \frac{W_{r_i}}{W_{max}}, \frac{t_{r_i}}{100}, \frac{d_{r_i}}{100}, \frac{PF_{r_i}}{10} \right).$$

After creating the input state, the RL agent selects an action according to its strategy. In line 7, we verify if the agent is in exploration mode. If that is the case, it selects a random action with probability ϵ from the action space (line 8). Otherwise, the agent forwards the input state through the main neural network to obtain the Q-values for all possible actions and choose the highest one (line 10). At this point, the algorithm triggers the *global data writer* to notify the deployment of the current VNF in the selected node (line 11). Then the *global scheduler* waits for the acknowledgment of the VNF's deployment from the master node controlling the selected node (line 12). In line 13, we calculate the reward for the action taken using Equation (6.14). Thus, we evaluate how good the algorithm's decision was. By obtaining the next state s^{t+1}, we complete the current state transition (s^t, a^t, r^t, s^{t+1}) and store it in the replay buffer (line 14). In line 15, the algorithm verifies if there are enough experiences in the replay buffer to perform a training step.

When sufficient state transitions exist, Algorithm 6.1 executes the training phase. This stage starts by sampling a random mini-batch of transitions from the replay buffer (line 16). For each element in the mini-batch (line 17), the algorithm checks if the transition corresponds to the episode's last step (line 18). If so, the target state-action value for that step is similar to the step's

obtained reward since there is no next state from which to gather the reward (line 19). For the other transitions (line 20), the next state-action value is calculated by using the target neural network and applying the discount factor γ for future rewards (line 21). The next step in the training process is to calculate the mean squared error loss between the next state-action value and the obtained Q-value using the main network (line 22). This step also includes updating the main neural network parameters by applying a gradient descent algorithm [155] to minimize the loss. Finally, for every determined number of steps, the algorithm updates the target DQN network's weights with the ones in the main network to include previously learned experiences (line 23).

6.3 Evaluation and Results

This section evaluates the proposed solution by comparing it with different approaches, thus demonstrating its feasibility for deploying events in a multi-cluster environment.

6.3.1 Evaluation Environment

To evaluate the performance of the proposed solution, we built a testbed formed by three clusters of four SBC nodes. One of the clusters was deployed in the University Carlos III of Madrid (UC3M), while the remaining clusters were placed at the Universitat Politècnica de Catalunya (UPC). Figure 6.4 depicts the described testbed. We employed Kubernetes 20.04 as the management framework for virtualized services. They run as Docker containers within pods and are placed into the devices. Thus, the deployed events utilize their available capacity according to predefined requirements. The intelligent controller was implemented using Java 1.8.0. In addition, we used the Deep Java Library 0.16.0 to implement our DQN algorithm. As mentioned, the local scheduler of each cluster ran the SOCCS algorithm proposed in Chapter 5.

Testing equipment: The SBC nodes used were Raspberry Pi 4 Model B [14] with 8 GB of RAM and an ARM64 processor with 4 cores, thus representing resource-constrained devices. The intelligent controller (i.e., the one that runs the DQN algorithm) was run on an Ubuntu system (Intel i9-7900X CPU @ 3.30 GHz with 20 cores and 64 GB of RAM) without limitations in the available resources. We used the UM24C module [146] to measure the power consumption of each node. It connects to Raspberry Pi devices via Bluetooth. The energy sources for the Pi devices are batteries with a capacity of 10000 mAh.

Service requests: In our evaluation scenarios, the services to be scheduled arrive one at a time following a Poisson distribution. We explored arrival rates ranging from 5 to 15 events per time unit. The main parameters used for creating the services were selected randomly from the values shown in Table 6.2 following a uniform distribution. The evaluation parameters were defined considering typical workloads derived from the literature.

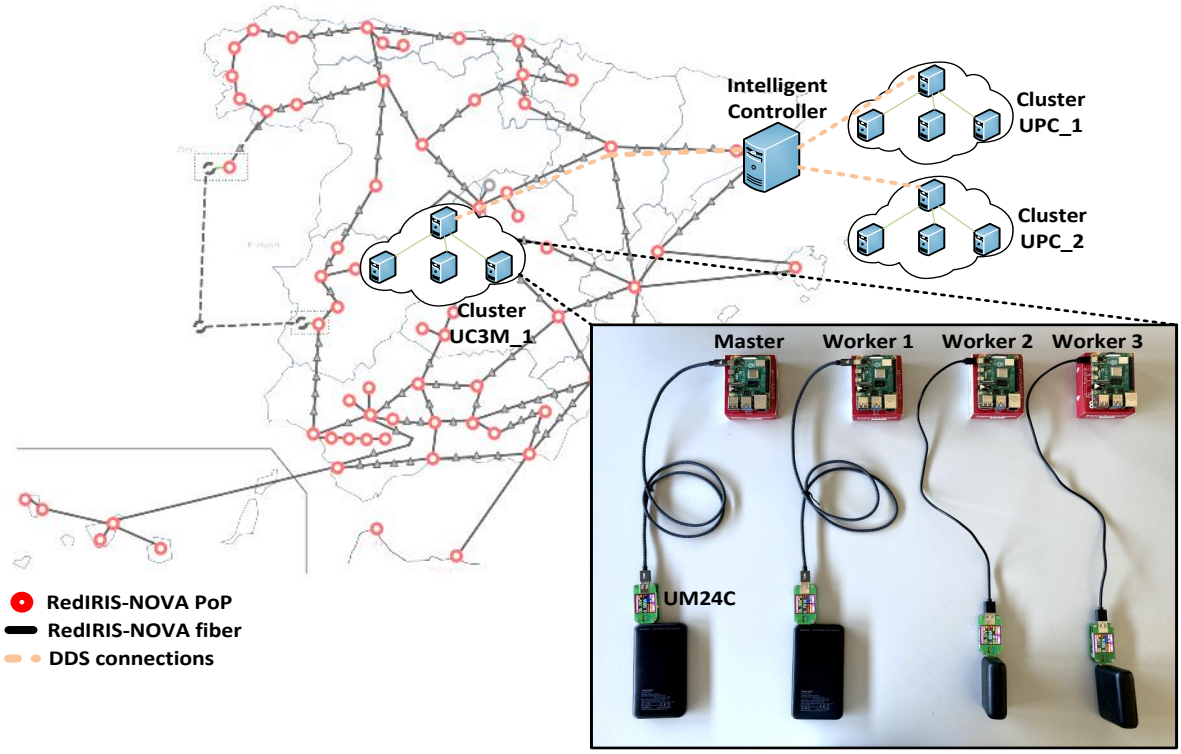


FIGURE 6.4. Deployed testbed formed by three distributed SBC clusters and the DQN-based intelligent controller.

Table 6.2: System model notation.

Parameter	**Values**
Number of VNFs in service	5–10
Processing capacity per node (MIPS)	500–3000
CPU capacity per node (milli-CPU)	4000
Memory capacity per node (Ki)	7998464
Required processing rate per event (MIPS)	100–500
Required CPU per event (milli-CPU)	150–250
Required memory per event (Ki)	200–500

Hyperparameters: The hyperparameters of the implemented DQNIC solution were tuned for efficiency and stability. The parameters of the DQN model were initialized with the learning rate $\alpha = 0.01$ and the reward discount factor $\gamma = 0.9$. The reward function's positive weights ζ, ξ, ϕ were set to 0.2, 0.5, and 0.3, respectively. The parameters related to the Epsilon decay schedule (i.e., initial, decay, and final Epsilon values) were 0.5, 0.002, and 0.01. We used a replay buffer with a storage capacity of 200 experiences and a batch size of 64 samples. The DQN networks, namely the online and target networks, were synchronized every 50 epochs. These networks were composed of three layers. The input layer with $3|K||N|+5$ neurons, where $|K|$ and $|N|$ were the

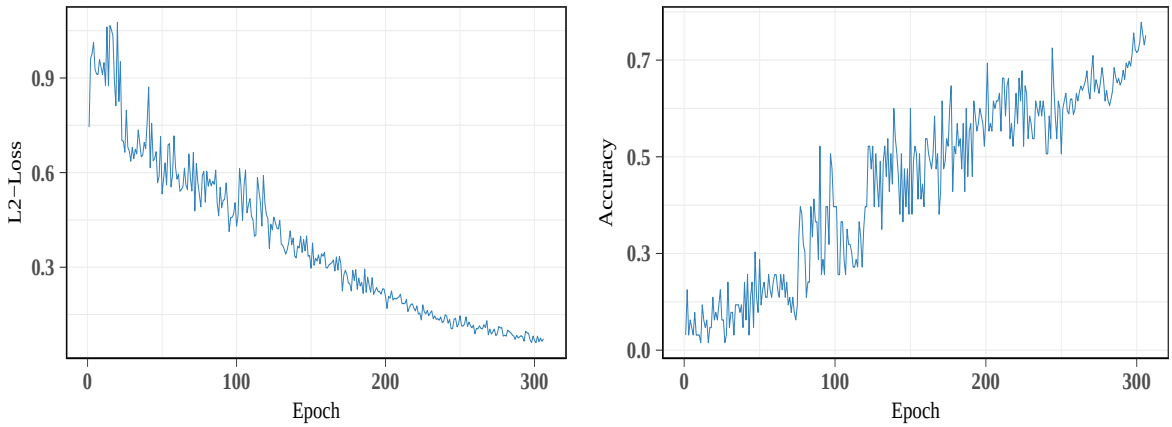
number of clusters and nodes, respectively. We used one hidden layer with a number of neurons equal to the mean of the input and output layers [156] and ReLU as the activation function. Finally, the output layer corresponded to the number of actions ($|K||N|+1$). The weights of the online network were updated after finishing every epoch.

Compared approaches: We compared our proposed solution with two algorithms: least loaded scheduler (LLS) and global SOC and capacity-based scheduler (GSOCCS).

- LLS – This mechanism allocates the events to the node with the highest available capacity. Thus, the node with the least CPU usage is chosen, resulting in a balanced use of computational resources.
- GSOCCS – This algorithm represents a global version of the solution presented in Chapter 5. It utilizes the information sent by the master of each cluster to calculate the nodes' scores before selecting the best one to deploy a determined request. This score allows the algorithm to choose the node with the maximum SOC and minimum CPU usage.

6.3.2 Training Phase Results

Before utilizing the proposed algorithm to deploy event requests in a multi-cluster environment, we first had to train the agent using Algorithm 6.1. To illustrate the training process, the training model's loss and accuracy are shown in Fig. 6.5.



(a) Model's loss.

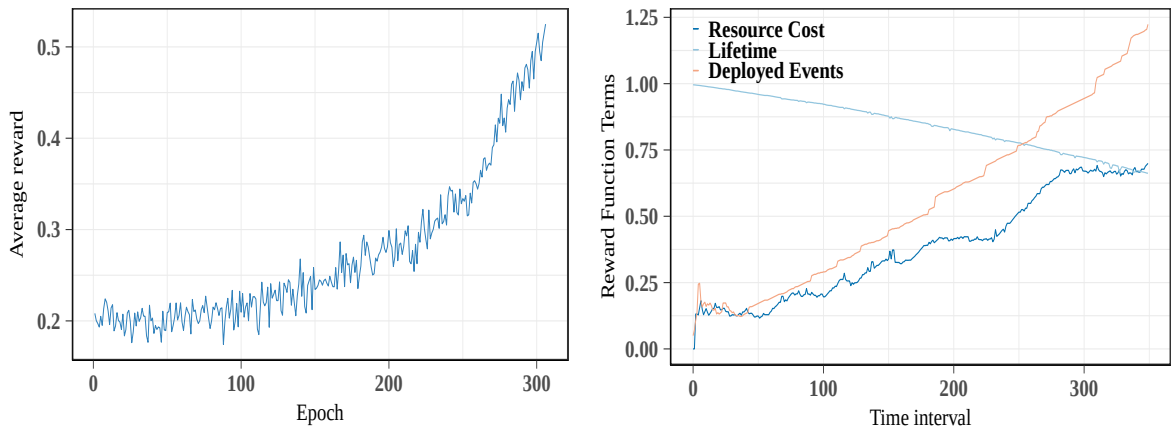
(b) Model's accuracy.

FIGURE 6.5. Training metrics of the model.

The former represents the mean square error between the label and prediction values obtained by the target and online networks, respectively. It can be observed that the loss between both networks gradually decreases and converges to 0 while increasing the number of epochs, see Fig. 6.5(a). On the other hand, Fig. 6.5(b) illustrates the behavior of the model's accuracy,

which slowly increases with the number of epochs. This metric indicates the number of correct predictions to the total number of predictions.

Another crucial indicator of the training model is the reward obtained after taking every action because it represents how well the environment performed with the agent's decision. Figure 6.6 depicts the behavior of this metric while training the model.



(a) Model's average reward.

(b) Reward function terms.

FIGURE 6.6. Reward function behavior during the training phase.

More specifically, Fig. 6.6(a) shows the average reward on each epoch, which gradually increases with the number of training periods. This performance proves the quality of the trained model since it ensures an increasing number of deployed events despite the depletion of available resources in the clusters' nodes.

The description above is better understood in Fig. 6.6(b), which illustrates the values of the reward function terms during the experiment. The primary term of the reward function is represented by the orange line and indicates the proportion of the deployed events against the requested ones. Its increasing behavior indicates the high acceptance ratio of the system while using the trained model. The light-blue line represents the system's lifetime, which decreases due to battery depletion. Including this term in the reward function guarantees that the actions taken consider the remaining battery's node. Additionally, the dark-blue line symbolizes the cost of the used resources. This value slightly increases since the model aims to reutilize nodes that have already deployed events while having available resources, thus minimizing the cost of using new ones.

During the training phase of our algorithm, we must include the exploration of the environment by applying random actions with a determined probability. Toward this aim, we implemented an Epsilon-greedy policy to work with our agent. Figure 6.7 depicts the calculated Epsilon values during each environment step. The Epsilon hyperparameters used are indicated

in Section 6.3.1. The Epsilon-greedy policy's behavior guaranteed random actions with higher probability at the beginning of the model training. However, it was most likely to calculate the Q_{max} to make a decision in the rest of the experiment.

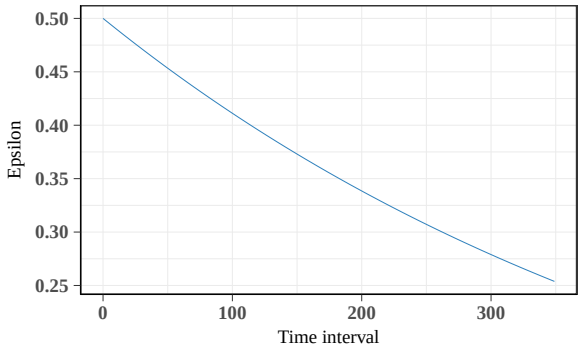


FIGURE 6.7. Values obtained through the Epsilon-greedy policy.

6.3.3 Comparison among Scheduling Approaches

This section evaluates the proposed algorithm by comparing it with the approaches described in Section 6.3.1. These algorithms were analyzed through the following metrics: scheduled and rejected events, acceptance ratio, resource cost, and battery consumption. We ran several experiments for each generation rate to ensure the reliability of the results. They show a confidence interval of 95% for all the analyzed methods.

6.3.3.1 Average Number of Scheduled and Rejected Events

Figure 6.8 portrays the results regarding requested, scheduled, and rejected services. It can be noted that the analyzed algorithms achieved similar results for the explored generation rates. All the algorithms deployed the requested services without rejection for the lowest generation rate (i.e., 5 events per time unit). GSOCCS and LLS rejected fewer services than the proposed solution for 10 events per time unit (i.e., one and two services, respectively). For the remaining generation rate, DQNIC increased by around one service the rejected events with regard to the compared approaches. Overall, the results reveal that the difference among the three algorithms was not significantly high.

Figure 6.9 provides a deeper insight into the constituent network functions. As stated in Section 6.1.1, the generated services were constituted by a set of VNFs. By analyzing the rejected VNFs, we noticed this metric increased with the event generation rate for all the analyzed algorithms. Expressly, no VNF was rejected for the lowest generation rate. Meanwhile, this metric gradually increased for the other rates. Similar to the rejected services, our proposal deployed fewer VNFs than the compared approaches but with a slight difference (i.e., around eight VNFs). This figure also includes the number of failed VNFs (red bar) as a metric to indicate

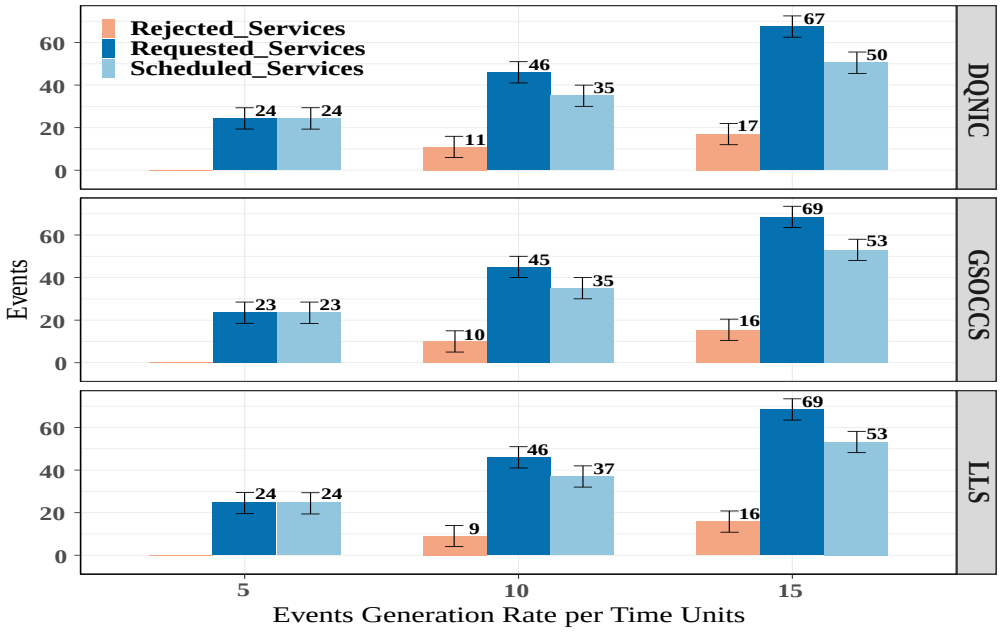


FIGURE 6.8. Number of requested, scheduled, and rejected services for all the scheduling algorithms.

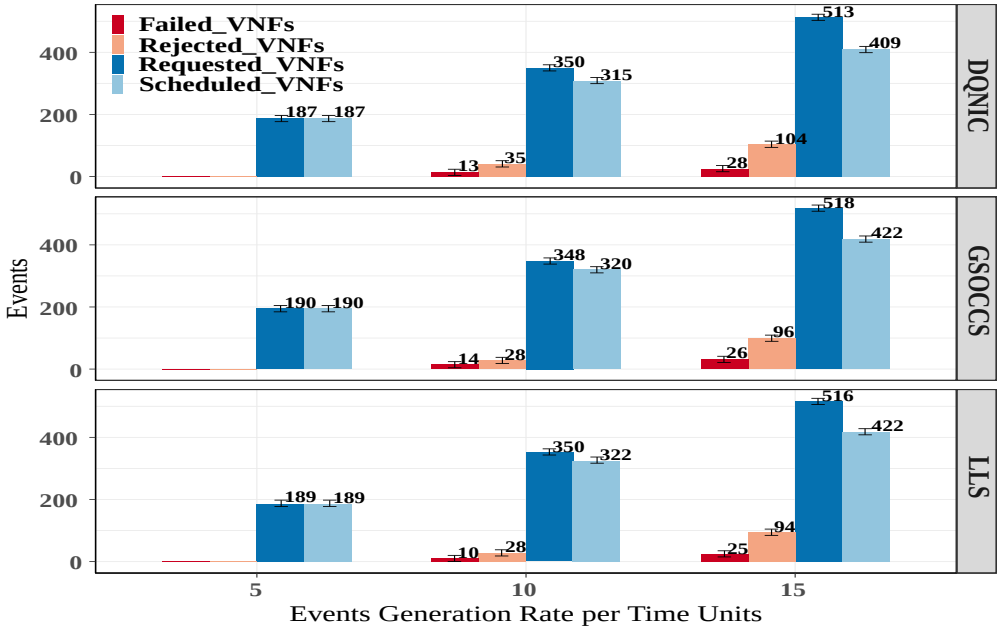


FIGURE 6.9. Number of requested, scheduled, rejected, and failed VNFs for all the scheduling algorithms.

the number of wrong decisions each algorithm makes. In more detail, it represents the non-deployed VNFs due to insufficient resources in the selected node, thus leading to the rejection

of the service and its constituent VNFs. By considering this metric, we indirectly evaluate the decision quality of the proposed algorithm as the number of failed VNFs is low for the evaluated generation rates with regard to the number of requested VNFs.

6.3.3.2 Average Acceptance Ratio

Similar to Chapter 5, we define the acceptance ratio as a quality metric of the algorithms that denotes the proportion between the number of scheduled and requested events. Figure 6.10 depicts the average value of this metric for each generation rate. The results correspond with the ones obtained in the above sections, where the proposed algorithm (i.e., DQNIC) had a lower acceptance ratio (i.e., 77% and 74% for 10 and 15 events per time unit, respectively) than the compared algorithms due to rejecting a small number of services. More specifically, the proposed solution decreased the acceptance ratio by around 3% and 5% with respect to GSOCCS and LLS, respectively, for 10 and 15 events per time unit. The three analyzed approaches have an acceptance ratio of 100% when deploying services with a generation rate of 5 events per time unit.

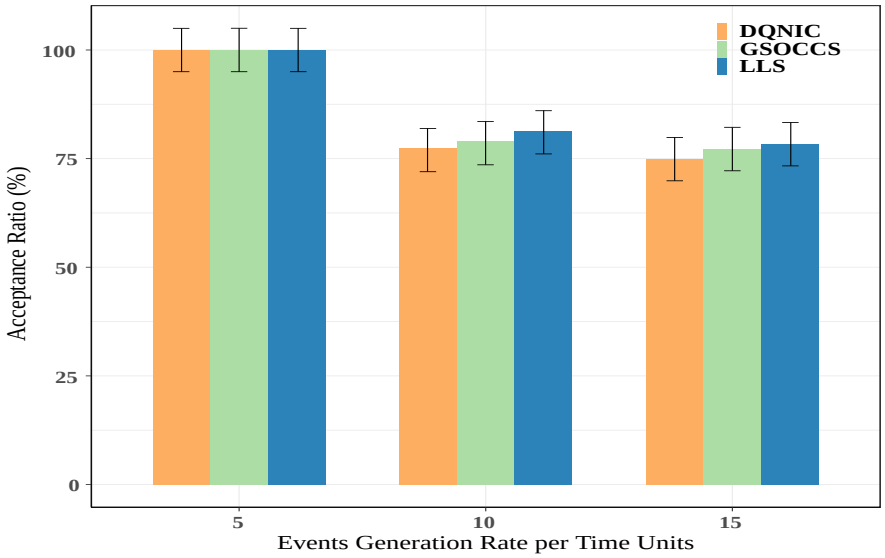


FIGURE 6.10. Event acceptance ratio for each scheduling algorithm.

6.3.3.3 Average Resource Cost

The resource cost is a crucial criterion to be considered in cloud and edge environments since it directly impacts the CAPEX and OPEX. We defined it as the sum of the proportion between the resources used and their maximum values, multiplied by a unitary cost. Figure 6.11 shows the average result of this metric for the analyzed allocation algorithms.

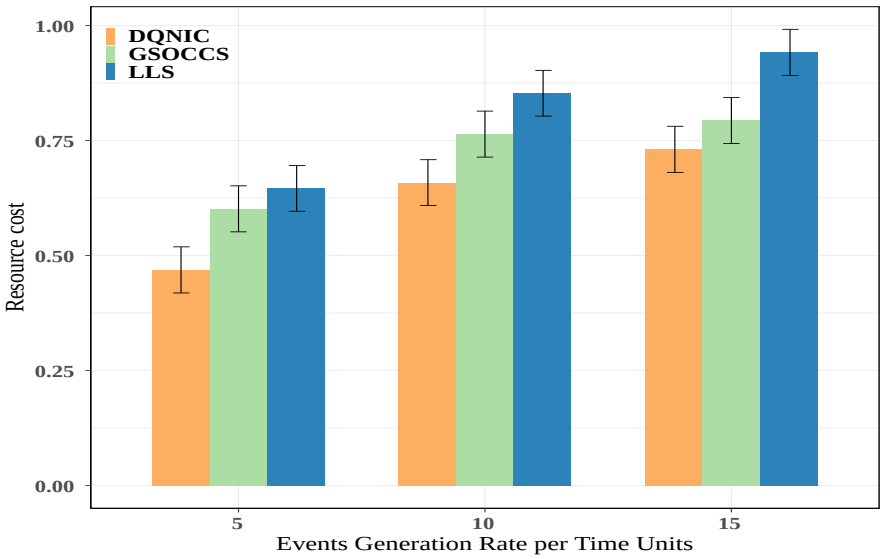


FIGURE 6.11. Resource cost for each scheduling algorithm.

When analyzing the 5 events per time unit generation rate, our proposal decreased the resources used with respect to GSOCCS and LLS by around 23% and 28%, respectively. Similarly, DQNIC outperformed the compared approaches with a reduction of 14% and 23% for a generation rate of 10 events per time unit. In the case of the highest analyzed generation rate, the proposed algorithm reduced the resources used by 8% and 22% in regard to GSOCCS and LLS, respectively. These results show that our solution makes cost-effective use of the node resources compared to the algorithms studied since it aims to reutilize as much as possible the same nodes to deploy new event requests.

6.3.3.4 Average Battery Consumption

The battery consumption was calculated by considering the difference between each experiment's initial and final values of SOC. Figure 6.12 illustrates the average battery consumption for each node in the three used SBC clusters when utilizing the studied allocation algorithms. To improve the readability of the figure, we grouped the nodes by considering their cluster's function. Thus, the dark colors represent the master of each cluster, while the light ones describe the worker nodes.

A general observation indicates that the battery consumption is higher while increasing the generation rates. This behavior was expected since the greater the event arrival rate, the higher the number of requested events. Thus, the clusters' operation time increases. Comparing the three schedulers, LLS had the highest battery consumption for all the generation rates since its balanced use of resources did not consider the nodes' SOC status, as did GSOCCS. These algorithms achieved less worker consumption imbalance than our proposal because they balanced

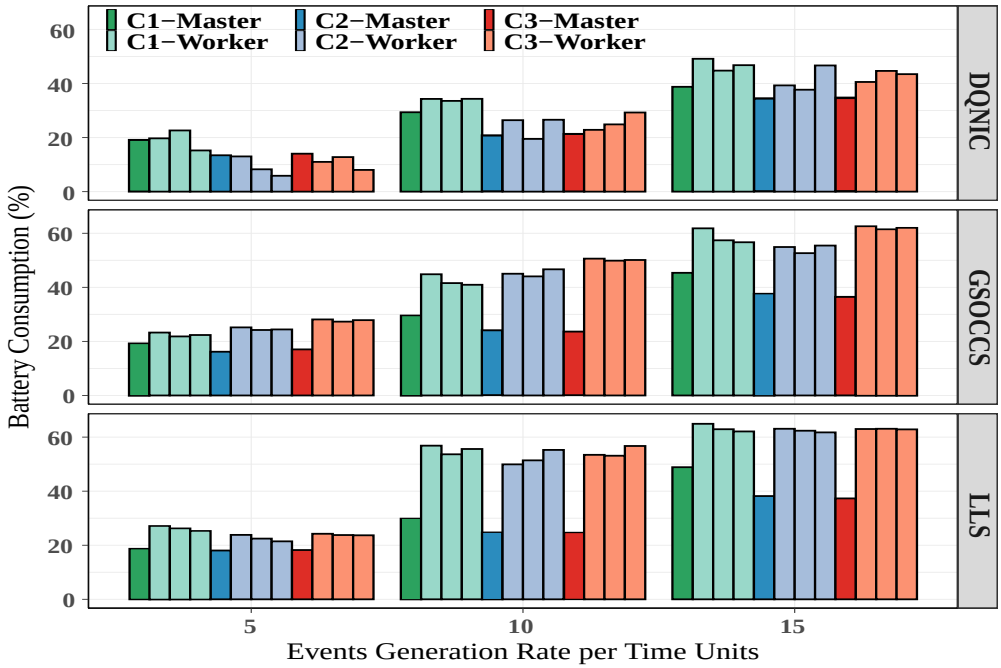


FIGURE 6.12. Battery consumption for each cluster's nodes while running different algorithms.

the event requests among all the clusters' workers. Nevertheless, DQNIC produced the lowest battery consumption in the overall system since its aim was to reduce this metric by reutilizing nodes to deploy events instead of using a different one.

A deep analysis of this figure reveals a similar battery consumption in the controller nodes of each cluster for the studied approaches. These nodes generally consumed less battery than the worker ones since they were only in charge of deploying the events according to the intelligent controller's decisions. Considering the highest generation rate, DQNIC saved up to 20% and 26% of the worker's average consumption in cluster 1 (light-green bars) compared with GSOCCS and LLS, respectively. Similarly, our approach reduced the worker's average consumption in cluster 2 (light-blue bars) by around 24% and 34% compared with the same algorithms. Regarding the cluster 3 worker's average consumption (orange bars), the proposed allocation mechanism diminished these values up to 30% and 32% with respect to GSOCCS and LLS, respectively.

6.4 Conclusion

This chapter proposed DQNIC as a global allocation mechanism capable of deploying service requests in a multi-domain edge environment by considering clusters' node status and service demands. Our approach implemented the DDS communication mechanism proposed in Chapter 3 to exchange information with the clusters' controller nodes. Additionally, a DRL algorithm was

integrated into our proposal to select the most suitable node where an event request must be deployed. The algorithm considered the nodes' status (i.e., CPU, memory utilization, and battery consumption) and the event's demands to make its decisions.

To evaluate the feasibility of our solution, we built a testbed formed by twelve SBC nodes grouped into three clusters and geographically distributed. In the first stage, we performed the training process of the proposed DRL algorithm to select the best set of hyperparameters that result in the expected behavior. Our proposed approach was compared with two baseline algorithms in the second phase. The results showed a slight reduction in the number of deployed events (i.e., services and VNFs) when running our solution compared with the other algorithms. Therefore, DQNIC had a lower acceptance ratio than the other algorithms, although the differences were insignificant.

In contrast, there were distinctions when analyzing the resource cost and battery consumption. Our proposed allocation mechanism reduced the resource cost by 23% and 28% compared with GSOCCS and LLS, respectively, for the lowest generation rate. Similarly, it decreased this metric for the highest generation rate with values up to 8% and 22% when comparing it with the same approaches. In terms of battery consumption, the differences were even more noticeable. Concretely, DQNIC reduced this metric in the clusters' worker nodes with values between 20% and 35% when compared with GSOCCS and LLS for the highest generation rate, thus increasing the lifetime of the overall system's nodes and, therefore, their resilience.

Based on these results, we can confirm that, in general, our algorithm outperformed the baseline approaches since it caused a significant reduction in the resource cost and the battery consumption in the clusters' nodes. Nevertheless, it was at the cost of a slight reduction in the acceptance ratio. However, the advantages outweigh the drawbacks when running DQNIC to deploy events in a multi-cluster resource-constrained environment.

CHAPTER 7

FINAL REMARKS

The fault tolerance and network resilience in SDN/NFV systems are crucial to provide high-quality services in 5G and beyond networks because the control elements of these environments are responsible for deploying the incoming requests. The existing literature mainly focuses on distributing these elements by introducing mechanisms to replicate determined information. However, the proposed approaches neglect the communication and collaboration among the control elements of these systems when allocating service requests. Moreover, the allocation process's effects are not considered as a factor that can impact the resilience of SDN/NFV systems, especially those that are resource-constrained. Therefore, this thesis focused on solving the above issues by providing different strategies that can be combined to address these problems on a global scale.

The primary objectives of this work to overcome the identified research challenges were as follow: 1) develop a communication mechanism capable of federating the control elements in SDN/NFV systems and enabling failure awareness in these devices, 2) introduce a dynamic energy-efficient allocation mechanism to place service requests in energy-constrained devices, and 3) propose an intelligent controller as a global allocation entity that integrates the proposed communication mechanism to exchange network information and allocation actions with other control elements. The study's contributions to these aims are detailed below.

7.1 Research Contributions

Resilience and fault-tolerant capabilities are key features in network systems since they are closely related to the fulfillment of QoS and reliability requirements. In this regard, most papers found during the literature review focused on distributing or replicating network information

to guarantee fault tolerance in the control elements. Thus, Chapter 3 contributed with a new communication mechanism for the SDN controllers, which enables network information exchange among controllers while allowing a faster response to network and controller failures. The proposed mechanism allows the auto-discovery of network elements and provides a configurable mode to announce the kind of information to be sent. In contrast to related works, the evaluation process of the DDS-based mechanism was performed in a real testbed comprising two geographically distant SDN domains. This contribution represented the first step to solving the second research objective to enable a communication channel between control elements in SDN/NFV environments.

Network resilience has been explored in the literature by distributing control elements. In this regard, we undertook a different approach to achieve network resilience with a use case that demonstrated the importance of applying reliable control mechanisms to guarantee fault tolerance. More specifically, Chapter 4 proposed an SDN-based solution to improve the availability of a deployed network service. By applying implemented load-balancing and auto-scaling strategies, the solution provides resilience at the service and infrastructure levels since the use case was susceptible to introducing network loops. Additionally, the proposed solution integrated the implemented DDS-based communication mechanism to provide redundancy to our approach and, at the same time, proved its interoperability by using it in different controllers' frameworks.

Chapter 5 analyzed network resilience from the perspective of VNF allocation and how this process could impact fault tolerance when considering energy-constrained systems. For this, we proposed a novel scheduling mechanism to allocate service requests in a battery-dependent SBC cluster. The proposed scheduler integrates a regression model to determine battery consumption through the used computing resources of the cluster's elements. This contribution enables intelligent decisions in the scheduler since it considers expected battery consumption and used resources during the allocation process. In this manner, the cluster's resilience was improved while optimizing the management of the available resources.

Additionally, the scheduler implementation expanded the utilization of the Kubernetes framework in energy-constrained scenarios since, in contrast to Kubernetes's scheduler, our proposal integrates energy measurements of the participating devices in the placement decisions. Furthermore, we included a study of the impact of considering the controller node in the scheduling process in a resource-constrained SBC cluster. Its outcome revealed a better utilization of the cluster's available resources by taking into account the computing and controller nodes. This criterion was observed during our scheduling mechanism's implementation phase.

Furthermore, Chapter 6 consolidated all these contributions and presented a global allocation mechanism to deploy virtual functions across multiple domains of SBC clusters. The proposed intelligent controller employs a DRL algorithm that considers resource utilization, battery consumption, and service requirements to select the most suitable nodes for where its constituent

virtual functions can be placed. The adoption of ML techniques improved the scalability of this solution with regard to the previous one presented when increasing the number of candidate nodes. Namely, the best node is obtained through the output of the used neural networks instead of iterating over all the candidates to find the best option. Moreover, the intelligent controller relies on the proposed DDS mechanism to exchange nodes' status, service requests, and allocation decisions. It demonstrated that the implemented communication mechanism guarantees message transmission among control elements of SDN and NFV environments.

Finally, the strategies above (i.e., DDS mechanism, SDN-based solution for transparent VNF cluster, energy-friendly scheduler, and intelligent controller) were evaluated in real testbeds composed of leading technologies, and all the implemented solutions were published in a GitHub repository, which can be accessed by request (see Appendix B).

7.2 Future Work

This section suggests several open research questions by considering the scope and limitations of the proposed approaches. In this regard, we encourage further research in several directions, identified below.

Traffic forecasting strategies in SDN-based solutions: The continuous development of ML techniques and Big Data allows the characterization or profiling of clients' demands, thus enabling the possibility of forecasting the network traffic. Existing load-balancing techniques are based on network information (e.g., current link and controller's load) provided by monitoring modules. However, taking action after link congestion or overloaded controller events is useless in 5G and beyond networks due to its stringent requirements in terms of latency and reliability.

Therefore, adopting traffic-forecasting mechanisms could enable load-balancing decisions in advance to adapt proactively to dynamic traffic variations. This can also be combined with scaling actions of network functions in virtualized environments. Furthermore, traffic forecasting facilitates SDN controller's operation since it can configure routing rules or trigger switch migration processes in advance. These ideas represent only a portion of what traffic forecasting could provide to increase fault tolerance and resilience in the network.

Event migration mechanisms in containerized environments: In resource-constrained environments (e.g., when using a determined number of SBC nodes), event-migration strategies could improve the fault tolerance of the cluster by reassigning demanding VNFs from critical nodes to available ones before going down due to battery depletion. Where and when are two crucial questions to answer during a migration process. In this regard, several factors should be considered such as resource consumption (e.g., CPU, memory, storage, and battery), node latency, and event requirements. Starting from an initial proposal can help to analyze more complex scenarios in further stages since the migration process could include nodes belonging to other clusters.

When analyzing existing migration strategies to deploy virtual functions, hot and cold migrations are often explored, especially when using virtual machines in VIMs such as OpenStack. However, there is a lack of proposals considering container deployments despite their growth in enabling technologies such as NFV and EC due to their versatility and slight virtualization. This scenario opens a broad research area through migration processes to improve fault tolerance and resilience in containerized environments. It is supported because the most prominent framework to manage and orchestrate containers (i.e., Kubernetes) does not implement any mechanism to reassign events from one node to another.

Reliable and latency-aware VNF placement and chaining mechanisms: Reliability and latency represent stringent requirements for 5G and beyond networks. The solutions proposed in this study consider service reliability when determining the best node in which to place a particular VNF. These approaches also take into account the specified running time of the service request during the node selection to enable the execution of its constituent VNFs within the system's lifetime. Such consideration is crucial in energy-constrained environments. However, deploying service requests with undefined running times is more complex because other strategies (e.g., event migration and VNF rechaining) must be considered to guarantee reliable service. The solutions proposed herein do not contemplate these mechanisms so as not to further increase their time consumption since additional variables and constraints must be defined.

Furthermore, an important parameter to consider in further research is latency because the end-to-end delay of service requests is crucial for future networks. In this sense, migration mechanisms must consider the latency among nodes to select the best target node without violating the service's end-to-end delay. This process should be additionally combined with a rerouting strategy to guarantee service reliability during the migration process. In this regard, backup data paths for VNF chaining can be considered during its initial placement by analyzing near deployments with similar characteristics. In this manner, the rerouting strategies can save time by using alternative data paths known beforehand. Only when the migration process has been completed could new incoming requests be routed through the original VNF already placed in a new node.

Federated learning using edge devices: Due to the improvement in computational resources of remote devices (e.g., mobile phones, wearable devices, and autonomous vehicles) and the development of Big Data, storing data locally and pushing network computation to the edge have increased. This scenario has led to a growing interest in exploiting the advantages of federated learning. This new paradigm belongs to distributed training methods and involves training models over remote devices. It enables collaboration among users in different locations to learn ML models without violating user privacy. More specifically, they do not send their privacy-sensitive personal data to a central server to obtain a well-trained model. Despite the benefits that federated learning offers, some research aspects remain to be explored such as the communication architecture and the fault tolerance among heterogeneous devices.

Communication architecture represents one of the core challenges in federated learning since the remote devices must communicate with a central server to update the global model. Therefore, communication is a key bottleneck during the training of a federated learning model because it involves many exchanged messages. This scenario increases the communication cost of federated learning. Thus, developing highly efficient communication methods is required. In this regard, sending small messages or reducing the communication intervals could improve communication efficiency in federated learning. However, utilizing these mechanisms introduces extra processing to the computational resources. Thus, finding the trade-off between the communication cost and the computational resources used is crucial.

Finally, the variety of participant devices in federated learning tasks is another characteristic of this paradigm. Computational and communication capabilities depend on the selected devices' hardware, network connectivity, and power. Under these circumstances, it is likely some devices cannot finish their tasks due to connectivity or energy problems. Therefore, federated learning mechanisms must tolerate heterogeneous hardware and be reliable regardless of any failure or disconnection in the participant devices.

APPENDIX A: PUBLICATIONS

- **A. Llorens-Carrodeguas**, I. Leyva-Pupo, C. Cervelló-Pastor, L. Piñeiro, S. Siddiqui, "An SDN-Based Solution for Horizontal Auto-Scaling and Load Balancing of Transparent VNF Clusters," *Sensors*, vol. 21, no. 24, Dec 2021.

- **A. Llorens-Carrodeguas**, S. G. Sagkriotis, C. Cervelló-Pastor, D. P. Pezaros, "An Energy-Friendly Scheduler for Edge Computing Systems," *Sensors*, vol. 21, no. 21, Oct 2021.

- **A. Llorens-Carrodeguas**, C. Cervelló-Pastor and I. Leyva-Pupo, "A Data Distribution Service in a Hierarchical SDN Architecture: Implementation and Evaluation," In *28th International Conference on Computer Communication and Networks (ICCCN)*, Valencia, Spain, 2019.

- I. Leyva-Pupo, C. Cervelló-Pastor and **A. Llorens-Carrodeguas**, "Optimal Placement of User Plane Functions in 5G Networks," In *17th International Conference on Wired/Wireless Internet Communications, IFIP WWIC 2019*, Wired/Wireless Internet Communications, Lecture Notes in Computer Science, vol 11618. Springer, Cham, 2019.

- **A. Llorens-Carrodeguas**, C. Cervelló-Pastor, I. Leyva-Pupo, J. M. López-Soler and J. Navarro-Ortiz, "The Role of Data Distribution Service in Failure-aware SDN Controllers," In *XXXIII Simposium Nacional de la Unión Científica Internacional de Radio (URSI)*, Granada, Spain, 2018.

- I. Leyva-Pupo, C. Cervelló-Pastor and **A. Llorens-Carrodeguas**, "A framework for placement and optimization of network functions in 5G," In *XXXIII Simposium Nacional de la Unión Científica Internacional de Radio (URSI)*, Granada, Spain, 2018.

- **A. Llorens-Carrodeguas**, C. Cervelló-Pastor and I. Leyva-Pupo, "Software Defined Networks and Data Distribution Service as Key Features for the 5G Control Plane," In *Distributed Computing and Artificial Intelligence, Special Sessions, 15th International Conference. DCAI 2018*, Advances in Intelligent Systems and Computing, vol 801. Springer, Cham, 2018.

- I. Leyva-Pupo, C. Cervelló-Pastor and **A. Llorens-Carrodeguas**, "The Resources Placement Problem in a 5G Hierarchical SDN Control Plane," In *Distributed Computing and Artificial Intelligence, Special Sessions, 15th International Conference. DCAI 2018*, Advances in Intelligent Systems and Computing, vol 801. Springer, Cham, 2018.

- **A. Llorens-Carrodeguas**, C. Cervelló-Pastor, I. Leyva-Pupo, J. M. Lopez-Soler, J. Navarro-Ortiz and J. A. Exposito-Arenas, "An architecture for the 5G control plane based on SDN and data distribution service," In *2018 Fifth International Conference on Software Defined Systems (SDS)*, Barcelona, Spain, 2018.

APPENDIX B

APPENDIX B: CODE REPOSITORY

The source code of the implemented strategies throughout this thesis can be accessed by request through an email to: alejandrollorens1991@gmail.com.

- **DDS-based communication mechanism** proposed in Chapter 3 can be found in: `https://github.com/alexllor1991/DDS-based-communication-mechanism`
- **SDN-based solution for transparent VNF cluster** proposed in Chapter 4 can be found in: `https://github.com/alexllor1991/SDN-based-solution-for-transparent-VNFs`
- **SOC and Capacity-based Scheduler (SOCCS)** proposed in Chapter 5 can be found in: `https://github.com/alexllor1991/SoC-Capacity-based-Scheduler-with-DDS`
- **Intelligent Controller** proposed in Chapter 6 can be found in: `https://github.com/alexllor1991/Intelligent-Controller`

References

[1] A. Mahmood, L. Beltramelli, S. F. Abedin, S. Zeb, N. I. Mowla, S. A. Hassan, E. Sisinni, and M. Gidlund, "Industrial IoT in 5G-and-beyond networks: Vision, architecture, and design trends," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 6, pp. 4122–4137, 2021.

[2] 3GPP, "Release 16 – The 5G System." Available online, URL: `https://www.3gpp.org/specifications-technologies/releases/release-16` (accessed on 02 November 2022).

[3] B. Blanco, J. O. Fajardo, I. Giannoulakis, E. Kafetzakis, S. Peng, J. Pérez-Romero, I. Trajkovska, P. S. Khodashenas, L. Goratti, M. Paolino, *et al.*, "Technology pillars in the architecture of future 5G mobile networks: NFV, MEC and SDN," *Computer Standards & Interfaces*, vol. 54, pp. 216–228, 2017.

[4] J. P. Sterbenz, D. Hutchison, E. K. Çetinkaya, A. Jabbar, J. P. Rohrer, M. Schöller, and P. Smith, "Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines," *Computer networks*, vol. 54, no. 8, pp. 1245–1265, 2010.

[5] A. S. da Silva, P. Smith, A. Mauthe, and A. Schaeffer-Filho, "Resilience support in software-defined networking: A survey," *Computer Networks*, vol. 92, pp. 189–207, 2015.

[6] Q. Duan, S. Wang, and N. Ansari, "Convergence of networking and cloud/edge computing: Status, challenges, and opportunities," *IEEE Network*, vol. 34, no. 6, pp. 148–155, 2020.

[7] S. Dutta, T. Taleb, and A. Ksentini, "QoE-aware elasticity support in cloud-native 5G systems," in *2016 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2016.

[8] J. Ma, W. Rankothge, C. Makaya, M. Morales, F. Le, and J. Lobo, "A comprehensive study on load balancers for VNF chains horizontal scaling," *arXiv preprint arXiv:1810.03238*, 2018.

[9] O. Adamuz-Hinojosa, J. Ordonez-Lucena, P. Ameigeiras, J. J. Ramos-Munoz, D. Lopez, and J. Folgueira, "Automated network service scaling in NFV: Concepts, mechanisms and scaling workflow," *IEEE Communications Magazine*, vol. 56, no. 7, pp. 162–169, 2018.

[10] S. Lange, N. Van Tu, S.-Y. Jeong, D.-Y. Lee, H.-G. Kim, J. Hong, J.-H. Yoo, and J. W.-K. Hong, "A network intelligence architecture for efficient vnf lifecycle management," *IEEE Transactions on Network and Service Management*, 2020.

[11] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and its role in the Internet of Things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, (New York, NY, USA), p. 13–16, Association for Computing Machinery, 2012.

[12] J. L. Álvarez, J. D. Mozo, and E. Durán, "Analysis of single board architectures integrating sensors technologies," *Sensors*, vol. 21, no. 18, p. 6303, 2021.

[13] E. Upton and G. Halfacree, *Raspberry Pi user guide*.
John Wiley & Sons, 2016.

[14] "Raspberry Pi 4." Available online, URL: `https://www.raspberrypi.org/products/raspberry-pi-4-model-b/` (accessed on 02 November 2022).

[15] S. J. Johnston, P. J. Basford, C. S. Perkins, H. Herry, F. P. Tso, D. Pezaros, R. D. Mullins, E. Yoneki, S. J. Cox, and J. Singer, "Commodity single board computer clusters and their applications," *Future Generation Computer Systems*, vol. 89, pp. 201–212, 2018.

[16] N. Slamnik-Kriještorac, H. Kremo, M. Ruffini, and J. M. Marquez-Barja, "Sharing distributed and heterogeneous resources toward end-to-end 5G networks: A comprehensive survey and a taxonomy," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 1592–1628, 2020.

[17] M. Abu-Lebdeh, D. Naboulsi, R. Glitho, and C. W. Tchouati, "On the placement of VNF managers in large-scale and distributed NFV systems," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 875–889, 2017.

[18] "Kubernetes." Available online, URL: `https://www.kubernetes.io/` (accessed on 02 November 2022).

[19] I. Real-Time Innovations, "RTI Connext DDS core libraries user's manual 5.2.3." Available online, URL: `http://community.rti.com/static/documentation/connext-dds/5.2.3/doc/manuals/connext_dds/html_files/RTI_ConnextDDS_CoreLibraries_UsersManual/index.htm#UsersManual/title.htm%3FTocPath%3D_____1` (accessed on 02 November 2022).

[20] Z. Zhang, Y. Xiao, Z. Ma, M. Xiao, Z. Ding, X. Lei, G. K. Karagiannidis, and P. Fan, "6G wireless networks: Vision, requirements, architecture, and key technologies," *IEEE Vehicular Technology Magazine*, vol. 14, no. 3, pp. 28–41, 2019.

[21] X. You, C.-X. Wang, J. Huang, X. Gao, Z. Zhang, M. Wang, Y. Huang, C. Zhang, Y. Jiang, J. Wang, *et al.*, "Towards 6G wireless communication networks: Vision, enabling technologies, and new paradigm shifts," *Science China Information Sciences*, vol. 64, no. 1, pp. 1–74, 2021.

[22] M. Hamdan, E. Hassan, A. Abdelaziz, A. Elhigazi, B. Mohammed, S. Khan, A. V. Vasilakos, and M. N. Marsono, "A comprehensive survey of load balancing techniques in software-defined network," *Journal of Network and Computer Applications*, vol. 174, p. 102856, 2021.

[23] W. Kellerer, A. Basta, P. Babarczi, A. Blenk, M. He, M. Klugel, and A. M. Alba, "How to measure network flexibility? a proposal for evaluating softwarized networks," *IEEE Communications Magazine*, vol. 56, no. 10, pp. 186–192, 2018.

[24] E. Kaljic, A. Maric, P. Njemcevic, and M. Hadzialic, "A survey on data plane flexibility and programmability in software-defined networking," *IEEE Access*, vol. 7, pp. 47804–47840, 2019.

[25] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Presented as part of the 2nd {USENIX} Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, vol. 12, pp. 1–6, 2012.

[26] Z. Cai, A. L. Cox, and T. Ng, "Maestro: a system for scalable OpenFlow control." Available online, URL: `http://hdl.handle.net/1911/96391` (accessed on 02 November 2022).

[27] D. Erickson, "The Beacon OpenFlow controller," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 13–18, ACM, 2013.

[28] Ryu, "RYU SDN framework." Available online, URL: `https://book.ryu-sdn.org/en/Ryubook.pdf` (accessed on 02 November 2022).

[29] P. Floodlight, "Floodlight OpenFlow controller." Available online, URL: `https://github.com/floodlight/floodlight` (accessed on 02 November 2022).

[30] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks.," in *OSDI*, vol. 10, pp. 1–6, 2010.

[31] A. Tootoonchian and Y. Ganjali, "HyperFlow: A distributed control plane for OpenFlow," in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pp. 3–3, 2010.

[32] P. Berde *et al.*, "ONOS: towards an open, distributed SDN OS," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 1–6, ACM, 2014.

[33] Y. Fu *et al.*, "A hybrid hierarchical control plane for flow-based large-scale software-defined networks," *IEEE Transactions on Network and Service Management*, vol. 12, no. 2, pp. 117–131, 2015.

[34] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain SDN controllers," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pp. 1–4, IEEE, 2014.

[35] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed SDN controllers in a multi-domain environment," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pp. 1–2, IEEE, 2014.

[36] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems.," in *USENIX annual technical conference*, vol. 8, Boston, MA, USA, 2010.

[37] J. Stribling *et al.*, "Flexible, wide-area storage for distributed systems with WheelFS.," in *NSDI*, vol. 9, pp. 43–58, 2009.

[38] F. Benamrane, M. Ben mamoun, and R. Benaini, "An east-west interface for distributed SDN control plane: implementation and evaluation," *Computers & Electrical Engineering*, vol. 57, pp. 162–175, 2017.

[39] P. Lin *et al.*, "A west-east bridge based SDN inter-domain testbed," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 190–197, 2015.

[40] "Titan distributed graph database." Available online, URL: `https://github.com/thinkaurelius/titan` (accessed on 02 November 2022).

[41] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[42] P. OpenDaylight, "OpenDaylight controller." Available online, URL: `https://www.opendaylight.org/` (accessed on 02 November 2022).

[43] "Project proposals: ODL-SDNi App - OpenDaylight project." Available online, URL: `https://wiki.opendaylight.org/display/ODL/ODL-SDNi+App` (accessed on 02 November 2022).

[44] "ODL-SDNi developer guide — OpenDaylight documentation Oxygen documentation." Available online, URL: `https://docs.opendaylight.org/en/stable-oxygen/`

developer-guide/odl-sdni-developer-guide.html (accessed on 02 November 2022).

[45] Object Management Group (OMG), "Data distribution service (DDS)." Available online, URL: https://www.omg.org/spec/DDS/ (accessed on 02 November 2022).

[46] L. Bertaux, A. Hakiri, S. Medjiah, P. Berthou, and S. Abdellatif, "A DDS/SDN based communication system for efficient support of dynamic distributed real-time applications," in *Distributed Simulation and Real Time Applications (DS-RT), 2014 IEEE/ACM 18th International Symposium on*, pp. 77–84, IEEE, 2014.

[47] A. Hakiri, P. Berthou, A. Gokhale, and S. Abdellatif, "Publish/subscribe-enabled software defined networking for efficient and scalable IoT communications," *IEEE communications magazine*, vol. 53, no. 9, pp. 48–54, 2015.

[48] A. Hakiri, P. Berthou, P. Patil, and A. Gokhale, "Towards a publish/subscribe-based open policy framework for proactive overlay software defined networking," *ISIS*, pp. 15–115, 2015.

[49] A. Hakiri and A. Gokhale, "Data-centric publish/subscribe routing middleware for realizing proactive overlay software-defined networking," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pp. 246–257, ACM, 2016.

[50] H.-Y. Liu, C.-Y. Chiang, H.-S. Cheng, and M.-L. Chiang, "OpenFlow-based server cluster with dynamic load balancing," in *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 99–104, IEEE, 2018.

[51] M.-L. Chiang, H.-S. Cheng, H.-Y. Liu, and C.-Y. Chiang, "SDN-based server clusters with dynamic load balancing and performance improvement," *Cluster Computing*, vol. 24, pp. 537–558, 2021.

[52] W. Chen, Z. Shang, X. Tian, and H. Li, "Dynamic server cluster load balancing in virtualization environment with OpenFlow," *International Journal of Distributed Sensor Networks*, vol. 11, no. 7, p. 531538, 2015.

[53] P. Manzanares-Lopez, J. P. Muñoz-Gea, and J. Malgosa-Sanahuja, "An MPTCP-compatible load balancing solution for pools of servers in OpenFlow SDN networks," in *2019 Sixth International Conference on Software Defined Systems (SDS)*, pp. 39–46, IEEE, 2019.

[54] Ntop, "PF_Ring: High-speed packet capture, filtering and analysis.." Available online, URL: https://www.ntop.org/products/packet-capture/pf_ring/ (accessed on 02 November 2022).

[55] J. V. G. de Oliveira, P. C. P. Bellotti, R. M. de Oliveira, A. B. Vieira, and L. J. Chaves, "Virtualizing packet-processing network functions over heterogeneous OpenFlow switches," *IEEE Transactions on Network and Service Management*, 2021.

[56] A. A. Abdelltif, E. Ahmed, A. T. Fong, A. Gani, and M. Imran, "SDN-based load balancing service for cloud servers," *IEEE Communications Magazine*, vol. 56, no. 8, pp. 106–111, 2018.

[57] A. Azzouni, R. Boutaba, and G. Pujolle, "NeuRoute: Predictive dynamic routing for Software-Defined Networks," *arXiv preprint arXiv:1709.06002*, 2017.

[58] A. Azzouni and G. Pujolle, "NeuTM: A neural network-based framework for traffic matrix prediction in SDN," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–5, IEEE, 2018.

[59] S. Sendra *et al.*, "Including artificial intelligence in a routing protocol using Software Defined Networks," in *Communications Workshops (ICC Workshops), 2017 IEEE International Conference on*, pp. 670–674, IEEE, 2017.

[60] G. Stampa *et al.*, "A Deep-Reinforcement Learning approach for Software-Defined Networking routing optimization," *arXiv preprint arXiv:1709.07080*, 2017.

[61] S.-C. Lin *et al.*, "QoS-aware adaptive routing in multi-layer hierarchical Software Defined Networks: a reinforcement learning approach," in *Services Computing (SCC), 2016 IEEE International Conference on*, pp. 25–33, IEEE, 2016.

[62] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications surveys & tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[63] ETSI, "Network functions virtualisation (NFV): Architectural framework." Available online, URL: `https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf` (accessed on 02 November 2022).

[64] ETSI-ISG-NFV, "Network function virtualization; management and orchestration," *White Paper*, vol. 1, 2014.

[65] W. Hajji, *Dynamic service chain composition in virtualised environment*. PhD thesis, Loughborough University, 2018.

[66] "OpenStack." Available online, URL: `https://www.openstack.org/` (accessed on 02 November 2022).

[67] "Amazon Web Service." Available online, URL: `https://aws.amazon.com/` (accessed on 02 November 2022).

[68] ONAP, "Open Network Automation Platform." Available online, URL: https://www.onap.org/ (accessed on 02 November 2022).

[69] OSM, "Open Source MANO." Available online, URL: https://osm.etsi.org/ (accessed on 02 November 2022).

[70] O. Baton, "An extensible and customizable NFV MANO-compliant framework." Available online, URL: https://openbaton.github.io/ (accessed on 02 November 2022).

[71] Cloudify, "Open source, multi-cloud orchestration platform." Available online, URL: https://cloudify.co/ (accessed on 02 November 2022).

[72] G. M. Yilma, Z. F. Yousaf, V. Sciancalepore, and X. Costa-Perez, "Benchmarking open source NFV MANO systems: OSM and ONAP," *Computer Communications*, vol. 161, pp. 86–98, 2020.

[73] P. Bertin, T. Mamouni, and S. Gosselin, "Next-generation pop with functional convergence redistributions," in *Fiber-Wireless Convergence in Next-Generation Communication Networks*, pp. 319–336, Springer, 2017.

[74] T. V. K. Buyakar, A. K. Rangisetti, A. A. Franklin, and B. R. Tamma, "Auto scaling of data plane VNFs in 5G networks," in *2017 13th International Conference on Network and Service Management (CNSM)*, pp. 1–4, IEEE, 2017.

[75] I. Alawe, Y. Hadjadj-Aoul, A. Ksentini, P. Bertin, and D. Darche, "On the scalability of 5G core network: the AMF case," in *2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pp. 1–6, IEEE, 2018.

[76] C. H. T. Arteaga, F. B. Anacona, K. T. T. Ortega, and O. M. C. Rendon, "A scaling mechanism for an evolved packet core based on network functions virtualization," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 779–792, 2019.

[77] A. Ghorab, A. Kusedghi, M. Nourian, and A. Akbari, "Joint VNF load balancing and service auto-scaling in NFV with multimedia case study," in *2020 25th International Computer Conference, Computer Society of Iran (CSICC)*, pp. 1–7, IEEE, 2020.

[78] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless fine-grained NFS for flexible per-flow customization," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pp. 3–17, 2016.

[79] Y. Li, Z. Han, S. Gu, G. Zhuang, and F. Li, "Dyncast: Use dynamic anycast to facilitate service semantics embedded in IP address," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–8, IEEE, 2021.

[80] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[81] P. J. Basford, S. J. Johnston, C. S. Perkins, T. Garnock-Jones, F. P. Tso, D. Pezaros, R. D. Mullins, E. Yoneki, J. Singer, and S. J. Cox, "Performance analysis of single board computer clusters," *Future Generation Computer Systems*, vol. 102, pp. 278–291, 2020.

[82] S. Sagkriotis, C. Anagnostopoulos, and D. P. Pezaros, "Energy usage profiling for virtualized single board computer clusters," in *2019 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1–6, IEEE, 2019.

[83] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A container-based edge cloud PaaS architecture based on Raspberry Pi clusters," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pp. 117–124, IEEE, 2016.

[84] X. Hu, S. Li, H. Peng, and F. Sun, "Robustness analysis of State-of-Charge estimation methods for two types of Li-ion batteries," *Journal of power sources*, vol. 217, pp. 209–219, 2012.

[85] K. S. Ng, C.-S. Moo, Y.-P. Chen, and Y.-C. Hsieh, "Enhanced coulomb counting method for estimating state-of-charge and state-of-health of lithium-ion batteries," *Applied energy*, vol. 86, no. 9, pp. 1506–1511, 2009.

[86] V. Pop, H. Bergveld, P. Notten, J. O. het Veld, and P. P. Regtien, "Accuracy analysis of the State-of-Charge and remaining run-time determination for lithium-ion batteries," *Measurement*, vol. 42, no. 8, pp. 1131–1138, 2009.

[87] Z. Xi, M. Dahmardeh, B. Xia, Y. Fu, and C. Mi, "Learning of battery model bias for effective state of charge estimation of lithium-ion batteries," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 9, pp. 8613–8628, 2019.

[88] J. G. Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[89] Y. Wu, Y. He, and L. Shi, "Energy-saving measurement in LoRaWAN-based wireless sensor networks by using compressed sensing," *IEEE Access*, vol. 8, pp. 49477–49486, 2020.

[90] J. C. Lim and C. Bleakley, "Adaptive WSN scheduling for lifetime extension in environmental monitoring applications," *International Journal of Distributed Sensor Networks*, vol. 8, no. 1, p. 286981, 2011.

[91] S. K. Abd, S. A. R. Al-Haddad, F. Hashim, A. B. Abdullah, and S. Yussof, "An effective approach for managing power consumption in cloud computing infrastructure," *journal of computational science*, vol. 21, pp. 349–360, 2017.

[92] A. Marahatta, Y. Wang, F. Zhang, A. K. Sangaiah, S. K. S. Tyagi, and Z. Liu, "Energy-aware fault-tolerant dynamic task scheduling scheme for virtualized cloud data centers," *Mobile Networks and Applications*, vol. 24, no. 3, pp. 1063–1077, 2019.

[93] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[94] P. Gazori, D. Rahbari, and M. Nickray, "Saving time and cost on the scheduling of fog-based IoT applications using deep reinforcement learning approach," *Future Generation Computer Systems*, vol. 110, pp. 1098–1115, 2020.

[95] D. Ding, X. Fan, Y. Zhao, K. Kang, Q. Yin, and J. Zeng, "Q-learning based dynamic task scheduling for energy-efficient cloud computing," *Future Generation Computer Systems*, vol. 108, pp. 361–371, 2020.

[96] A. Varasteh, B. Madiwalar, A. Van Bemten, W. Kellerer, and C. Mas-Machuca, "Holu: Power-aware and delay-constrained VNF placement and chaining," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1524–1539, 2021.

[97] I. Litvinchev and E. L. Ozuna, "Lagrangian heuristic for the facility location problem," *IFAC Proceedings Volumes*, vol. 46, no. 24, pp. 107–113, 2013.

[98] O. Soualah, M. Mechtri, C. Ghribi, and D. Zeghlache, "Energy efficient algorithm for vnf placement and chaining," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 579–588, IEEE, 2017.

[99] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[100] A. Jayanetti, S. Halgamuge, and R. Buyya, "Deep reinforcement learning for energy and time optimized scheduling of precedence-constrained tasks in edge–cloud computing environments," *Future Generation Computer Systems*, vol. 137, pp. 14–30, 2022.

[101] Pegasus, "Pegasus: Makes the work flow." Available online, URL: `https://pegasus.isi.edu/` (accessed on 02 November 2022).

[102] O. A. Wahab, N. Kara, C. Edstrom, and Y. Lemieux, "MAPLE: A machine learning approach for efficient placement and adjustment of virtual network functions," *Journal of Network and Computer Applications*, vol. 142, pp. 37–50, 2019.

[103] V. Eramo, M. Ammar, and F. G. Lavacca, "Migration energy aware reconfigurations of virtual network function instances in nfv architectures," *IEEE Access*, vol. 5, pp. 4927–4938, 2017.

[104] V. Eramo, E. Miucci, M. Ammar, and F. G. Lavacca, "An approach for service function chain routing and virtual function network instance migration in network function virtualization architectures," *IEEE/ACM Transactions on Networking*, vol. 25, no. 4, pp. 2008–2025, 2017.

[105] G. D. Forney, "The Viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[106] J. Pei, P. Hong, M. Pan, J. Liu, and J. Zhou, "Optimal VNF placement via deep reinforcement learning in SDN/NFV-enabled networks," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 2, pp. 263–278, 2019.

[107] D. Qi, S. Shen, and G. Wang, "Towards an efficient VNF placement in network function virtualization," *Computer Communications*, vol. 138, pp. 81–89, 2019.

[108] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE Transactions on Network and Service Management*, vol. 13, no. 4, pp. 725–739, 2016.

[109] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, pp. 1346–1354, IEEE, 2015.

[110] Y. Mu, L. Wang, and J. Zhao, "Energy-efficient and interference-aware VNF placement with deep reinforcement learning," in *2021 IFIP Networking Conference (IFIP Networking)*, pp. 1–9, IEEE, 2021.

[111] G. L. Santos, T. Lynn, J. Kelner, and P. T. Endo, "Availability-aware and energy-aware dynamic SFC placement using reinforcement learning," *The Journal of Supercomputing*, vol. 77, no. 11, pp. 12711–12740, 2021.

[112] R. Solozabal, J. Ceberio, A. Sanchoyerto, L. Zabala, B. Blanco, and F. Liberal, "Virtual network function placement optimization with deep reinforcement learning," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 2, pp. 292–303, 2019.

[113] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[114] C. Schulte, M. Lagerkvist, and G. Tack, "Gecode." Available online, URL: `https://www.gecode.org` (accessed on 02 November 2022).

[115] R. Bruschi, A. Carrega, and F. Davoli, "A game for energy-aware allocation of virtualized network functions," *Journal of Electrical and Computer Engineering*, vol. 2016, 2016.

[116] M. M. Tajiki, S. Salsano, L. Chiaraviglio, M. Shojafar, and B. Akbari, "Joint energy efficient and QoS-aware path allocation and VNF placement for service function chaining," *IEEE Transactions on Network and Service Management*, vol. 16, no. 1, pp. 374–388, 2018.

[117] W. Khemili, J. E. Hajlaoui, and M. N. Omri, "Energy aware fuzzy approach for vnf placement and consolidation in cloud data centers," *Journal of Network and Systems Management*, vol. 30, no. 3, pp. 1–29, 2022.

[118] F. Fkih and M. N. Omri, "Irafca: an o (n) information retrieval algorithm based on formal concept analysis," *Knowledge and Information Systems*, vol. 48, no. 2, pp. 465–491, 2016.

[119] X. Xia, L. Gui, and Z.-H. Zhan, "A multi-swarm particle swarm optimization algorithm based on dynamical topology and purposeful detecting," *Applied Soft Computing*, vol. 67, pp. 126–140, 2018.

[120] N. H. Thanh, N. T. Kien, N. Van Hoa, T. T. Huong, F. Wamser, and T. Hossfeld, "Energy-aware service function chain embedding in edge–cloud environments for iot applications," *IEEE Internet of Things Journal*, vol. 8, no. 17, pp. 13465–13486, 2021.

[121] C. Assi, S. Ayoubi, N. El Khoury, and L. Qu, "Energy-aware mapping and scheduling of network flows with deadlines on vnfs," *IEEE Transactions on Green Communications and Networking*, vol. 3, no. 1, pp. 192–204, 2018.

[122] 3GPP, "TS 23.501- System Architecture for the 5G System; Stage 2." Available online, URL: `http://www.3gpp.org/ftp/Specs/archive/23_series/23.501/23501-f00.zip` (accessed on 02 November 2022).

[123] 3GPP, "TR 23.791 - Study of Enablers for Network Automation for 5G; Release 16." Available online, URL: `https://www.3gpp.org/ftp/Specs/archive/23_series/23.791/23791-g20.zip` (accessed on 20 November 2022).

[124] "RedIRIS." Available online, URL: `https://www.rediris.es/lared/` (accessed on 02 November 2022).

[125] O. Salman *et al.*, "SDN controllers: a comparative study," in *Electrotechnical Conference (MELECON), 2016 18th Mediterranean*, pp. 1–6, IEEE, 2016.

[126] OpenDaylight, "OpenFlow plugin operation." Available online, URL: `https://docs.opendaylight.org/projects/openflowplugin/en/latest/users/operation.html` (accessed on 02 November 2022).

[127] "Mininet: An instant virtual network on your laptop (or other PC)." Available online, URL: `http://mininet.org/` (accessed on 02 November 2022).

[128] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

[129] M. Bencivenni, D. Bortolotti, A. Carbone, A. Cavalli, A. Chierici, S. Dal Pra, D. De Girolamo, L. Dell'Agnello, M. Donatelli, A. Fella, *et al.*, "Performance of 10 gigabit ethernet using commodity hardware," *IEEE Transactions on Nuclear Science*, vol. 57, no. 2, pp. 630–641, 2010.

[130] D. Bortolotti, A. Carbone, D. Galli, I. Lax, U. Marconi, G. Peco, S. Perazzini, V. M. Vagnoni, and M. Zangoli, "Comparison of UDP transmission performance between IP-over-infiniband and 10-Gigabit ethernet," *IEEE Transactions on Nuclear Science*, vol. 58, no. 4, pp. 1606–1612, 2011.

[131] M. J. Christensen and T. Richter, "Achieving reliable UDP transmission at 10 Gb/s using BSD socket for data acquisition systems," *Journal of Instrumentation*, vol. 15, no. 09, p. T09005, 2020.

[132] HAProxy, "The reliable, high performance TCP/HTTP load balancer." Available online, URL: `https://www.haproxy.org/` (accessed on 02 November 2022).

[133] OpenStack, "Introducing Octavia." Available online, URL: `https://docs.openstack.org/octavia/queens/reference/introduction.html` (accessed on 1 November 2022).

[134] OpenStack, "OpenStack Nova service." Available online, URL: `https://docs.openstack.org/nova/latest/` (accessed on 02 November 2022).

[135] B. Isyaku, M. S. Mohd Zahid, M. Bte Kamat, K. Abu Bakar, and F. A. Ghaleb, "Software defined networking flow table management of OpenFlow switches performance and security challenges: A survey," *Future Internet*, vol. 12, no. 9, p. 147, 2020.

[136] N. Ha and N. Kim, "Efficient flow table management scheme in SDN-based cloud computing networks," *Journal of Information Processing Systems*, vol. 14, no. 1, pp. 228–238, 2018.

[137] S. Kaur, K. Kumar, J. Singh, and N. S. Ghumman, "Round-robin based load balancing in Software Defined Networking," in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 2136–2139, 2015.

[138] Iperf, "The ultimate speed test tool for TCP, UDP and SCTP." Available online, URL: `https://iperf.fr/` (accessed on 02 November 2022).

[139] B. Turkovic, F. A. Kuipers, and S. Uhlig, "Fifty shades of congestion control: A performance and interactions evaluation," *arXiv preprint arXiv:1903.03852*, 2019.

[140] A. Statkus, Š. Paulikas, and A. Krukonis, "TCP acknowledgment optimization in low power and embedded devices," *Electronics*, vol. 10, no. 6, p. 639, 2021.

[141] C. Tipantuña, X. Hesselbach, V. Sánchez-Aguero, F. Valera, I. Vidal, and B. Nogales, "An NFV-based energy scheduling algorithm for a 5G enabled fleet of programmable unmanned aerial vehicles," *Wireless Communications and Mobile Computing*, vol. 2019, 2019.

[142] B. Nogales, I. Vidal, V. Sanchez-Aguero, F. Valera, L. Gonzalez, and A. Azcorra, "OSM PoC 10 automated deployment of an IP telephony service on UAVs using OSM." Available online, URL: `https://osm.etsi.org/wikipub/index.php/OSM_PoC_10_Automated_Deployment_of_an_IP_Telephony_Service_on_UAVs_using_OSM` (accessed on 02 November 2022), 2020.

[143] F. Huet, "A review of impedance measurements for determination of the state-of-charge or state-of-health of secondary batteries," *Journal of power sources*, vol. 70, no. 1, pp. 59–69, 1998.

[144] L. Fahrmeir, T. Kneib, S. Lang, and B. Marx, "Regression models," in *Regression*, pp. 21–72, Springer, 2013.

[145] Z. Deng, X. Hu, X. Lin, Y. Che, L. Xu, and W. Guo, "Data-driven state of charge estimation for lithium-ion battery packs based on Gaussian process regression," *Energy*, vol. 205, p. 118000, 2020.

[146] "UM24C." Available online, URL: `https://www.mediafire.com/folder/0jt6xx2cyn7jt` (accessed on 02 November 2022).

[147] L. Wang and E. Gelenbe, "Adaptive dispatching of tasks in the cloud," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 33–45, 2015.

[148] M. Sarvabhatla, S. Konda, C. S. Vorugunti, and M. N. Babu, "A dynamic and energy efficient greedy scheduling algorithm for cloud data centers," in *2017 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pp. 47–52, IEEE, 2017.

[149] AI@EDGE, "The AI@EDGE H2020 Project." Available online, URL: https://aiatedge.eu/ (accessed on 02 November 2022).

[150] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "{NetBricks}: Taking the v out of {NFV}," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 203–216, 2016.

[151] D. Qi, S. Shen, and G. Wang, "Virtualized network function consolidation based on multiple status characteristics," *IEEE Access*, vol. 7, pp. 59665–59679, 2019.

[152] Q. Zhang, F. Liu, and C. Zeng, "Online adaptive interference-aware VNF deployment and migration for 5G network slice," *IEEE/ACM Transactions on Networking*, vol. 29, no. 5, pp. 2115–2128, 2021.

[153] D. Zeng, L. Gu, S. Pan, J. Cai, and S. Guo, "Resource management at the network edge: A deep reinforcement learning approach," *IEEE Network*, vol. 33, no. 3, pp. 26–33, 2019.

[154] Z. Ding, Y. Huang, H. Yuan, and H. Dong, "Introduction to reinforcement learning," in *Deep reinforcement learning*, pp. 47–123, Springer, 2020.

[155] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[156] J. Heaton, *Introduction to neural networks with Java*.
Heaton Research, Inc., 2008.