Repository Istituzionale dei Prodotti della Ricerca del Politecnico di Bari

Enabling Technologies and Hierarchical Control Plane Management of Software-Defined Networks

This is a PhD Thesis

Original Citation:
Enabling Technologies and Hierarchical Control Plane Management of Software-Defined Networks / Shah, Awais Aziz. - ELETTRONICO. - (2022). [10.60576/poliba/iris/shah-awais-aziz_phd2022]

Availability:
This version is available at http://hdl.handle.net/11589/232838 since: 2021-12-29

Published version
DOI:10.60576/poliba/iris/shah-awais-aziz_phd2022

Publisher: Politecnico di Bari

Terms of use:

(Article begins on next page)

26 June 2026

Department of Electrical and Information Engineering
ELECTRICAL AND INFORMATION ENGINEERING
Ph.D. Program
S.S.D. ING-INF/03 – TELECOMMUNICATIONS

Dissertation

Enabling Technologies and Hierarchical Control Plane Management of Software-Defined Networks

by

Awais Aziz Shah

Supervisors:
Prof. Luigi Alfredo Grieco
Prof. Gennaro Boggia

Coordinator of the Ph.D. Program:
Prof. Mario Carpentieri

Course n°34, 01/11/2018-31/10/2021

Politecnico di Bari

LIBERATORIA PER L'ARCHIVIAZIONE DELLA TESI DI DOTTORATO

Al Magnifico Rettore
del Politecnico di Bari

Il/la sottoscritto/a AWAIS AZIZ SHAH nato/a PESHAWAR (PAKISTAN) il 11/09/1985

residente a BARI in via SIGISMONDO CASTROMEDIANO 75 e-mail AWAIS.SHAH@POLIBA.IT

iscritto al 3° anno di Corso di Dottorato di Ricerca in INGEGNERIA ELETTRICA E DELL'INFORMAZIONE ciclo XXXIV

ed essendo stato ammesso a sostenere l'esame finale con la prevista discussione della tesi dal titolo:

ENABLING TECHNOLOGIES AND HIERARCHICAL CONTROL PLANE MANAGEMENT OF SOFTWARE-DEFINED NETWORKS

DICHIARA

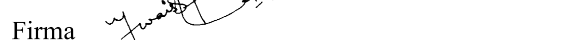
1) di essere consapevole che, ai sensi del D.P.R. n. 445 del 28.12.2000, le dichiarazioni mendaci, la falsità negli atti e l'uso di atti falsi sono puniti ai sensi del codice penale e delle Leggi speciali in materia, e che nel caso ricorressero dette ipotesi, decade fin dall'inizio e senza necessità di nessuna formalità dai benefici conseguenti al provvedimento emanato sulla base di tali dichiarazioni;
2) di essere iscritto al Corso di Dottorato di ricerca INGEGNERIA ELETTRICA E DELL'INFORMAZIONE ciclo XXXIV, corso attivato ai sensi del "*Regolamento dei Corsi di Dottorato di ricerca del Politecnico di Bari*", emanato con D.R. n.286 del 01.07.2013;
3) di essere pienamente a conoscenza delle disposizioni contenute nel predetto Regolamento in merito alla procedura di deposito, pubblicazione e autoarchiviazione della tesi di dottorato nell'Archivio Istituzionale ad accesso aperto alla letteratura scientifica;
4) di essere consapevole che attraverso l'autoarchiviazione delle tesi nell'Archivio Istituzionale ad accesso aperto alla letteratura scientifica del Politecnico di Bari (IRIS-POLIBA), l'Ateneo archivierà e renderà consultabile in rete (nel rispetto della Policy di Ateneo di cui al D.R. 642 del 13.11.2015) il testo completo della tesi di dottorato, fatta salva la possibilità di sottoscrizione di apposite licenze per le relative condizioni di utilizzo (di cui al sito http://www.creativecommons.it/Licenze), e fatte salve, altresì, le eventuali esigenze di "embargo", legate a strette considerazioni sulla tutelabilità e sfruttamento industriale/commerciale dei contenuti della tesi, da rappresentarsi mediante compilazione e sottoscrizione del modulo in calce (Richiesta di embargo);
5) che la tesi da depositare in IRIS-POLIBA, in formato digitale (PDF/A) sarà del tutto identica a quelle **consegnate**/inviate/da inviarsi ai componenti della commissione per l'esame finale e a qualsiasi altra copia depositata presso gli Uffici del Politecnico di Bari in forma cartacea o digitale, ovvero a quella da discutere in sede di esame finale, a quella da depositare, a cura dell'Ateneo, presso le Biblioteche Nazionali Centrali di Roma e Firenze e presso tutti gli Uffici competenti per legge al momento del deposito stesso, e che di conseguenza va esclusa qualsiasi responsabilità del Politecnico di Bari per quanto riguarda eventuali errori, imprecisioni o omissioni nei contenuti della tesi;
6) che il contenuto e l'organizzazione della tesi è opera originale realizzata dal sottoscritto e non compromette in alcun modo i diritti di terzi, ivi compresi quelli relativi alla sicurezza dei dati personali; che pertanto il Politecnico di Bari ed i suoi funzionari sono in ogni caso esenti da responsabilità di qualsivoglia natura: civile, amministrativa e penale e saranno dal sottoscritto tenuti indenni da qualsiasi richiesta o rivendicazione da parte di terzi;
7) che il contenuto della tesi non infrange in alcun modo il diritto d'Autore né gli obblighi connessi alla salvaguardia di diritti morali od economici di altri autori o di altri aventi diritto, sia per testi, immagini, foto, tabelle, o altre parti di cui la tesi è composta.

Luogo e data Bari, 30-12-2021 Firma

Il/La sottoscritto, con l'autoarchiviazione della propria tesi di dottorato nell'Archivio Istituzionale ad accesso aperto del Politecnico di Bari (POLIBA-IRIS), pur mantenendo su di essa tutti i diritti d'autore, morali ed economici, ai sensi della normativa vigente (Legge 633/1941 e ss.mm.ii.),

CONCEDE

- al Politecnico di Bari il permesso di trasferire l'opera su qualsiasi supporto e di convertirla in qualsiasi formato al fine di una corretta conservazione nel tempo. Il Politecnico di Bari garantisce che non verrà effettuata alcuna modifica al contenuto e alla struttura dell'opera.
- al Politecnico di Bari la possibilità di riprodurre l'opera in più di una copia per fini di sicurezza, back-up e conservazione.

Luogo e data Bari, 30-12-2021 Firma 

Politecnico
di Bari

Department of Electrical and Information Engineering
ELECTRICAL AND INFORMATION ENGINEERING
Ph.D. Program
SSD: ING-INF/03–TELECOMMUNICATIONS

Final Dissertation

Enabling Technologies and Hierarchical Control Plane Management of Software-Defined Networks

---

by

Awais Aziz Shah

Referees:

Prof. Sabrina Sicari

Prof. Floriano De Rango

Supervisors:

Prof. Luigi Alfredo Grieco

Prof. Gennaro Boggia

Coordinator of Ph.D. Program:

Prof. Mario Carpentieri

Course n°34, 01/11/2018-31/10/2021

To my mother, whose sacrifices enabled me to perform to the best of my ability. She has been praying for me the entire time, encouraging me to accomplish the best during this arduous process, making me more energetic and passionate to do this work in order to complete my PhD degree.

To my lovely wife and kids for their unwavering support in the most difficult times. Without their caring support and encouragement, I would not have been able to attain these goals. My wife has been extremely helpful throughout the entire journey to my destination. She gave up her time, comforts, and multiple loving relationships to allow me to advance in my life. Because of my busy schedule and tight deadlines, I was unable to provide you guys with the time, attention, and comfort you needed.

To my grandparents and uncles, who raised me since I was a child. They've been the foundation of everything I've accomplished.

To the memory of my grandfather, Muyassir Shah, who always believed in my ability to be successful in the academic arena. You are gone but your belief in me has made this journey possible.

Acknowledgements

Here, I take this opportunity to acknowledge some important people, who have continuously been a source of courage and inspiration to accomplish this work and towards achieving my short and long term milestones throughout this doctoral period.

First of all, my supervisors Prof. Ing. Luigi Alfredo Grieco and Prof. Ing. Gennaro Boggia who always helped me out and guided me towards the right path. I am thankful to them for providing me with the opportunity to work closely with the Telecom companies in Italy. I am also very thankful to Dr. Giuseppe Piro for his guidance towards my research, and pointing out my mistakes which have been a source of continuous and quick learning for me during these days. Here, I acknowledge that it could not have been possible without their selfless support and availability round the clock. I admire the way they obliged me offering their valuable time and suggestions whenever needed.

Last, but not the least, I would like to thank my colleagues in the lab including Paolo, Sergio, Arcangela, Vittoria, and Pietro who were always there for me whenever I required any kind of help. It was really a wonderful experience working here at the Telematics Laboratory in such a cooperative environment where all the colleagues around me were amazing and their behavior made the things easier to look forward with every passing day at POLIBA.

Thank you all

Awais Aziz Shah

Declaration

I, Awais Aziz Shah, declare that this thesis titled, “Enabling Technologies and Hierarchical Control Plane Management of Software-Defined Networks” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date: 30-12-2021

Abstract

This thesis details the research activities on the investigations and experimentation performed for the development of a Software-Defined Networks and Network Function Virtualization based large-scale distributed hierarchical architecture for Software-Defined Networks.

In the first step, this research started off with analyzing the state of the art enabling technologies by performing a qualitative cross-comparison among the available technologies (including container engines, orchestrators, and many other supporting tools) using set of Key Performance Indicators (KPIs). Then, in the second step, to understand the interplay of these technologies in virtualized service infrastructures, by considering the main outcomes of the qualitative analysis, the set of technologies including Docker as the container engine, Docker Swarm and Kubernetes as orchestrators with load balancing and service discovery capabilities are deployed on bare-metal and OpenStack cloud platforms. Experimental tests are conducted to identify the most suitable set of technologies in the high-load and industrial use-case of a smart farm using KPIs e.g., CPU utilization, memory footprint, network load, connection delay, and request completion time. In the third step, an innovative, distributive, and hierarchical framework based on the SDN paradigm is developed, which comprises two levels of SDN controllers to configure and monitor large-scale optical switches and networking functionalities. On top of that, Virtual Network Functions (VNFs) are optimally deployed and managed by a centralized orchestrator. The final step proceeds towards the formation of a novel methodology for the dynamic and reactive management of forwarding rules in a (potentially large-scale) SDN-based network, based on the knowledge of network topology, the power consumption of optical switches, the expected volume of traffic, and the variability of the actual traffic load.

Contents

List of Figures ix

List of Tables xiii

Introduction: Dissertation Overview xv

Personal Scientific Contributions xxi

1 Qualitative Cross-Comparison of Emerging Technologies 1

1.1 Overview 1

1.2 Containerization Technologies 3

1.3 Container Engines 3

1.3.1 LXC(Linux Containers) 3

1.3.2 Docker 3

1.3.3 LXD 4

1.3.4 Rkt 4

1.3.5 Kata Containers 4

1.4 Container orchestrators 6

1.4.1 Kubernetes 6

1.4.2 Docker Compose 6

1.4.3 Docker Swarm 6

1.4.4 OpenStack 7

1.4.5 Nomad 7

1.4.6 Apache Mesos 7

1.4.7 Bistro 8

1.4.8 Amazon's Elastic Container Service (ECS) 8

1.4.9 Cloud Foundry 8
1.4.10 OpenShift 8
1.5 Supporting Tools for Containerization 9
1.6 Load balancing tools 9
1.6.1 Envoy 9
1.6.2 HAProxy and Bamboo 9
1.6.3 Kube-Proxy 9
1.6.4 MetalLB 10
1.6.5 NGINX 10
1.6.6 Traefik 10
1.6.7 Vamp-router 10
1.6.8 Vulcand 10
1.6.9 Service discovery tools 10
1.6.10 ZooKeeper 11
1.6.11 Etcd 11
1.6.12 Consul 11
1.6.13 Mesos-DNS 11
1.6.14 SkyDNS 11
1.6.15 WeaveDNS 11
1.6.16 SmartStack 12
1.6.17 Eureka 12
1.7 User interfaces 12
1.7.1 Rancher 12
1.7.2 Clocker 12
1.7.3 Tutum 13
1.7.4 Portainer 13
1.7.5 Kitematic 13
1.7.6 DockStation 13
1.7.7 Panamax 13
1.7.8 Docker UI 13
1.7.9 Docker Compose UI 14
1.7.10 Shipyard 14
1.8 Protocols 14

1.8.1 OpenFlow 14
1.8.2 NETCONF 14
1.8.3 RESTCONF 15
1.9 Cross comparison of SDS enabling technologies 15
1.9.1 Summary 20

2 Quantitative Cross-Comparison of Emerging Technologies for Virtualized Service Infrastructures 23

2.1 Motivation and overview 24
2.2 Background on container networking 26
2.3 Integration of cutting-edge technologies enabling the container networking paradigm 29
2.4 Cross comparison and performance assessment 35
2.4.1 Cross-comparison in a high-load environment 37
2.4.1.1 CPU utilization 38
2.4.1.2 Memory footprint 39
2.4.1.3 Network load 42
2.4.1.4 Connection delay and request completion time 44
2.4.1.5 Final considerations emerging from the analysis of the high-load environment 48
2.4.2 Performance assessment in a smart farm scenario 48
2.4.2.1 CPU utilization 50
2.4.2.2 Memory Footprint 50
2.4.2.3 Network load 50
2.4.2.4 Connection delay and request completion time 53
2.4.2.5 Final considerations emerging from the analysis of the smart farm scenario 56
2.5 Summary 57

3 Design and Development of Optical Network Orchestration Framework 59

3.1 Motivation and Overview 60
3.2 The Proposed Framework 61
3.2.1 Targeted use cases 62

3.2.2 High-level architecture 65
3.2.2.1 Telecom Nodes 65
3.2.2.2 Specialized SDN Controller (level-1 Controller) 65
3.2.2.3 Multi-vendor/Multi-domain SDN Controller (level-2 Controller) 65
3.2.2.4 VNF Orchestrator 65
3.2.2.5 Framework Orchestrator 65
3.3 The implemented testbed 65
3.3.1 Components of the simulation framework 66
3.3.2 Communication protocols and interaction 67
3.3.3 The developed driver 68
3.3.3.1 UML 68
3.3.4 Sequence Diagrams 70
3.3.5 Orchestrator API 73
3.3.6 Achieved implementation and the developed simulation framework 74
3.3.7 Automated deployment 76
3.3.8 Demo of the simulated optical nodes 77
3.4 Summary 79

4 Designing of Dynamic Forwarding Strategy with Energy and Bandwidth Constraints 81
4.1 Routing Strategies discovered in the state of the art 81
4.2 The reference architecture and main assumptions 83
4.3 The conceived approach 85
4.3.1 Initial network configuration based on the traffic matrix (Task 1). 85
4.3.2 Redefinition of forwarding rules based on congestion episodes (Task 2). 86
4.4 Performance Evaluation 88
4.5 Summary 91

5 Conclusions and Future Research Directions 93

References 97

List of Figures

1.1 Big picture of Software Defined Systems based on container networking. 5

2.1 Testbed 1: integration of Docker and Docker Swarm on bare-metal. . . . 31
2.2 Testbed2: integration of Docker and Kubernetes on bare-metal. 33
2.3 Testbed 3: integration of Docker and Docker Swarm on OpenStack cloud. 34
2.4 Testbed 4: integration of Docker and Kubernetes on OpenStack cloud. . 36
2.5 CPU utilization measured when $\lambda = 5$ requests/minutes. 38
2.6 CPU utilization measured when $\lambda = 10$ requests/minutes. 39
2.7 Cumulative distribution function of CPU utilization measurements. . . . 40
2.8 Memory footprint measured when $\lambda = 5$ requests/minutes. 41
2.9 Memory footprint measured when $\lambda = 10$ requests/minutes. 41
2.10 Cumulative distribution function of memory footprint measurements. . . 42
2.11 Network load measured when $\lambda = 5$ requests/minutes. 43
2.12 Network load measured when $\lambda = 10$ requests/minutes. 43
2.13 Cumulative distribution function of network load measurements. 44
2.14 Connection delays measured when $\lambda = 5$ requests/minutes. 45
2.15 Connection delays measured when $\lambda = 10$ requests/minutes. 46
2.16 Request completion time measured when $\lambda = 5$ requests/minutes. 47
2.17 Request completion time measured when $\lambda = 10$ requests/minutes. . . . 47
2.18 The virtualized service infrastructure evaluated in the smart farm use case. 49
2.19 CPU usage measured during the emulation of the smart farm use case. . 51
2.20 Cumulative distribution function of CPU usage measured during the emulation of the smart farm use case. 51

2.21 Memory footprint measured during the emulation of the smart farm use case. 52
2.22 Cumulative distribution function of the memory footprint measured during the emulation of the smart farm use case. 53
2.23 Network load measured during the emulation of the smart farm use case. 54
2.24 Cumulative distribution function of the network load measured during the emulation of the smart farm use case. 54
2.25 Connection delay measured during the emulation of the smart farm use case. 55
2.26 Cumulative distribution function of the conneciton delay measured during the emulation of the smart farm use case. 55
2.27 Request completion time measured during the emulation of the smart farm use case. 55
2.28 Cumulative distribution function of the request completion measured during the emulation of the smart farm use case. 56

3.1 The conceived high-level framework. 61
3.2 UML Class diagram of the developed ONOS Driver. 69
3.3 UML Class diagram of the INTENTO Driver's Model Objects. 71
3.4 Sequence diagram of a controller connection. 72
3.5 Sequence diagram of the links discovery process. 73
3.6 UML Class diagram of the OrchestratorAPI App. 74
3.7 Sequence diagram of the execution after an API request. 75
3.8 The achieved implementation. 76
3.9 The developed simulation framework. 77
3.10 The automated deployment script. 78
3.11 A virtual telecom node representing multiple interfaces on the optical node simulator. 78

4.1 Reference Transport-SDN (T-SDN) network architecture. 84
4.2 Example showing the ability of the proposed approach to achieve energy and quality of service constraints 89
4.3 Power consumption. 89
4.4 Percentage of links turned off. 90

4.5 Throughput degradation registered by active data flows 91

List of Tables

1.1 Cross comparison among containerization engines. 16
1.2 Cross comparison among containers orchestrators. 18
1.3 Joint usage of load balancing tools and container engines. 19
1.4 Joint usage of service discovery tools and container engines. 19
1.5 Joint usage of user interfaces and container engines. 20

2.1 Summary of the investigated state of the art. 28

Introduction: Dissertation Overview

Over the last decade, there has been a drastic increase in the applications of cloud computing and communication-related services due to the rapid increase in the number of smart mobile devices. After sensing the high demand for these services and applications, Telco operators are rapidly expanding their existing network infrastructures. To achieve this goal, virtualization technologies serve as an efficient solution to better utilize the existing network resources. The aim is to provide acceptable quality of service to the end-users within the available resources.

Recently, with advances in the telecommunication networks, Software-Defined Networks (SDN) emerged as a game changing technology by revolutionizing the network architecture to virtualize the network resources and introduce network intelligence. It basically separates the data plane from the control plane by introducing a centralized entity known as 'SDN controller' that allows the network Administrators to globally regulate the network states via network policies in either a centralized (one controller) or distributed manner (many controllers). Traditionally, the network architectures were tightly coupled with the data plane making it a challenging task to dynamically manage the underlying infrastructure and virtualize the network. SDN introduces agility within the operator's networks that leads to a significant reduction in the network's capital and operating expenditures. The evolution of SDN drives to programmatically control the network devices and have a global view of the underlying network. By leveraging the programmability and the global network view, SDN can simplify network operations and also increase network resource utilization.

Recently, Network Function Virtualization (NFV) emerged as a paradigm shift to facilitate the SDN deployments to virtually deploy network functions on generic hard-

ware within the network. The joint-integration of SDN, NFV, and cloud computing facilitates unprecedented levels of network control, dynamicity, and flexibility and thus leads towards the definition of Software-Defined Systems (SDS). Teleco operators are willing to opt for this opportunity to deploy SDN, NFV, and cloud computing in their optical transport networks.

SDS aims to deploy the resources, services, and applications in a virtualized manner to ensure flexibility, agility, isolation, and performance within the network. For decades, virtualization was achieved using Virtual Machines i.e., a hypervisor-based approach that emulates the underlying operating environment but recently, a novel light-weight virtualization approach known as 'Containers' has made a paradigm shift in the virtualization world. Containers are used as state-of-the-art technology for the deployment of services and applications in the modern communication networks, however, the selection of technologies for the deployment of virtualized service infrastructure based on container networking is a tough task since a plethora of technologies for container networking (i.e., container engine and orchestrator) are emerging every day backed by several standardization organizations and the telco operators are unsure of the performance and joint integration of these technologies to be deployed within their networks.

To fully utilize the potentials of SDN/NFV paradigm, Telco operators aim for a Telco-cloud orchestration platform based on the SDN/NFV principles for the development, deployment, testing, and simulation of network services/functions, applications, and algorithms in a real-time complex T-SDN environment which can be replicated in their Optical Transport networks. However, achieving this aim is not straightforward and poses numerous challenges in terms of design and selection of technologies for the simulation/orchestration framework to ensure unprecedented levels of network dynamicity, orchestration, management, and flexibility in the network. In addition, with the enormous growth in the years to come, a challenging goal frequently noticed in the current literature is in fact to reduce the power consumption of the operating network, while satisfying the requested levels of quality of service (e.g., bandwidth consumption).

The potential of the cloud-computing supported the design of advanced services and applications, leveraging virtual components distributed at the large scale. Indeed, the joint integration of SDN, NFV, and cloud-computing principles recently paved the way towards the definition of SDS. The baseline principles of SDS embrace a number of novel enabling technologies (like container engines, orchestrators, and many other

supporting tools) that significantly simplify the integration and the management of virtual components, while promising high level of flexibility, isolation, and performance. These technologies play an important part in achieving the light-weight virtualization. To this end, we started off with a qualitative cross-comparison of the aforementioned technologies already available in the industry. A variety of technologies are studied with a special focus on the joint-integration of these technologies along with presenting the pros and cons of each of them, which provides various insights for the adoption of these tools. It is important to understand the interplay of these technologies in virtualized service infrastructures, therefore, by considering the main outcomes of the qualitative analysis, the set of technologies including Docker as the container engine, Docker Swarm and Kubernetes as orchestrators with load balancing and service discovery capabilities are deployed on bare-metal and OpenStack cloud platforms. Experimental tests are conducted to identify the most suitable set of technologies in high-load and industrial use-case of a smart farm using Key Performance Indicators (KPIs) that include CPU utilization, memory footprint, network load, connection delay, and request completion time. The results indicate that the combination of Docker and Kubernetes on the bare-metal deployment platform represents a suitable solution for effectively exploiting container networking capabilities in real deployments.

Moreover, an innovative hierarchical orchestration framework based on the SDN/NFV principles has been developed in the context of INTENTO project (recently funded by the Apuglia reigon Italy). The objective is to create an innovative simulation/orchestration framework by selecting the best technologies and use it to test applications, services, and advanced optimization algorithms in a real environment. A large-scale, distributed, and hierarchical Transport T-SDN architecture has been designed, where optical switches and networking functionalities are monitored and dynamically configured through a two-level hierarchical structure of SDN controllers. On top of that, Virtual Network Functions (VNFs) are optimally deployed and managed by a centralized orchestrator, based on the network condition, user requests, and application requirements. This architecture is used as a foundation for a complex simulation environment that harmoniously integrates within the OpenStack cloud: optical node simulators composed by simulation agent and a suitable hardware emulation layer; proprietary SDN network controller designed to enable the innovative optical nodes characteristics; Open Network Operating System as the second level

controller, enabling the integration of third-party or standardized models (multivendor environment), based on standardized interfaces and communication protocols.

Additionally, this work investigates the state-of-the-art routing strategies available in the T-SDN context. The literature review highlights a requirement for a real-time dynamic routing strategy that should mutually provide energy efficiency and quality of service within the network. To accomplish this purpose, a novel routing strategy has been proposed for the dynamic management of forwarding rules in T-SDN deployments with energy and bandwidth constraints. The proposed strategy jointly considers the network topology, the power consumption of optical switches, the expected volume of traffic, and the variability of the actual traffic load. In particular, the proposed strategy starts by activating the minimum required nodes and transport links between the source and destination pair predefined within a given traffic matrix, based on the network topology and the estimated power consumption of the optical switches. Then, the bandwidth utilization of the activated transport links is periodically monitored by a centralized controller to recognize the actual traffic load. The proposed strategy is dynamically reactive, thanks to the deployment of SDN controller which periodically monitors the network elements and in the case if a congestion is detected on optical nodes, new transport links and optical switches are activated to ensure the smooth running of the traffic inside the network while ensuring energy efficiency and bandwidth requirements by the user. Experimental tests demonstrate the better trade-off between the power consumption and quality of service.

Moreover, other correlated research activities, strictly in lined with those aforementioned, have been carried out during the PhD work. A list of produced scientific contributions is presented immediately following this introduction. To conclude, a brief description on the structuring of this dissertation is outlined below:

- Chapter 1: Qualitative Cross-Comparison of Emerging Containerization Technologies. It starts with analyzing various container networking technologies available in the industry to achieve the SDS vision and provides a cross-comparison based on a set of KPIs between the available technologies along with their pros and cons. It finally outlines the various possible integrations between the set of available technologies.

- Chapter 2: Quantitative Cross-Comparison of Container Networking Technologies for Virtualized Service Infrastructures. In this chapter, to fully analyze the potentials of container networking technologies, performance analysis has been performed by practically deploying the best set of technologies within virtualized service infrastructures. Experimental tests are firstperformed in the use case of high load environment and further, in a industrial smart farm environment. Results obtained based are discussed and prominent technologies are highlighted in this domain.

- Chapter 3: Design and Development of Optical Network Orchestration Framework. It provides the design and development of an innovative orchestration framework based on the SDN/NFV paradigm for the development, deployment, testing, and simulation of network functions, services, and application. It demonstrates the elements involved in the design of multi-level architecture and further provides a demo of the simulated optical nodes.

- Chapter 4: Designing of Dynamic Forwarding Strategy with Energy and Bandwidth Constraints. This chapter focuses on the design and development of a novel methadology for the dynamic management of forwarding rules within the T-SDN deployments. First, the description of the proposed strategy is given, followed by practical implementation in a simulation environment. Finally, experimental results are compared with the existing strategy and further discussed.

- Chapter 5: Conclusions and Future Research Directions: The chapter starts with the concluding remarks on this Ph.D. thesis outlining the key findings of the overall work.

0. INTRODUCTION: DISSERTATION OVERVIEW

Personal Scientific Contributions

Scientific contributions leading to publications during this PhD work are enlisted in what follows. They have been accepted and published in international journals and conferences.

International Journals

1. Awais Aziz Shah, Giuseppe Piro, Luigi Alfredo Grieco, and Gennaro Boggia,“ A quantitative cross-comparison of container networking technologies for virtualized service infrastructures in local computing environments,” Transactions on Emerging Telecommunications Technologies, vol. 32, Issue 4, pp. 1 - 23, 2021. doi: 10.1002/ett.4234.

International Conferences

1. Awais Aziz Shah, Giuseppe Piro, Luigi Alfredo Grieco, and Gennaro Boggia, A Qualitative Cross-Comparison of Emerging Technologies for Software-Defined Systems, Proc. of 2019 Sixth International Conference on Software Defined Systems (SDS). IEEE, June,2019.

2. Awais Aziz Shah, Giuseppe Piro, Luigi Alfredo Grieco, and Gennaro Boggia, A Review of Forwarding Strategies in Transport Software-Defined Networks, Proc. of 22nd International Conference on Transparent Optical Networks (ICTON). IEEE, July, 2020.

3. Awais Aziz Shah, Marco Mussini, Francesco Nicassio, Giorgio Parladori, Francesco Triggiani, Giovanni Grieco, Giuseppe Iaffaldano, and Giuseppe Piro,“ A Real-time Simulation Framework for Complex and Large-scale Optical Transport Networks based on the SDN Paradigm”, Proc. of IEEE/ACM

International Symposium on Distributed Simulation and Real Time Applications (DS-RT), Sep., 2020.

4. Antonio Petrosino, Giancarlo Sciddurlo, Giovanni Grieco, Awais Aziz Shah, Giuseppe Piro, Luigi Alfredo Grieco, and Gennaro Boggia, Dynamic Management of Forwarding Rules in a T-SDN Architecture with Energy and Bandwidth Constraints, Proc. of IEEE International Conference on Ad Hoc Networks and Wireless (AdHoc-Now), Springer, October, 2020.

1

Qualitative Cross-Comparison of Emerging Technologies

This chapter starts with the brief introduction of SDS and throws light on the state of the art enabling technologies (i.e., container networking) available for the deployment of SDS-based infrastructures. A qualitative cross-comparison of set of emerging technologies including container engines, orchestrators, load-balancers, service discovery tools, communication protocols, and graphical shells is presented here. The comparison is based on a set of KPIs and the results are discussed in detail to facilitate the selection of technologies for SDS, which provides the grounds for the deployment of the T-SDN-based hierarchical control plane and framework conceived in this work.

1.1 Overview

During the last decade, SDN and NFV introduced a revolutionary way to deploy programmable network architectures, based on a strict separation between data plane and control plane and a native ability to dynamically define, install, and configure virtualized network facilities [1],[2]. At the same time, the potential of the cloud-computing supported the design of advanced services and applications, leveraging virtual components distributed at the large scale. Indeed, the joint integration of SDN, NFV, and cloud-computing principles recently paved the way towards the definition of SDS [3].

In the SDSs vision, resources, services, and applications are treated as a combination of virtual boxes, interacting with each other through the underlying communica-

tion infrastructure. Their management demands high levels of flexibility, isolation, and performance [4, 5, 6]. Since virtualization techniques based on traditional Virtual Machines (VMs) cannot meet these requirements [7], other enabling technologies are gaining momentum in the current state of the art. They include, for example, lightweight containers, container orchestrators, and many other supporting tools [8, 9, 10, 11]. Therefore, as a matter of fact, SDS are significantly grounding their roots into the container networking paradigm [12, 13, 14].

At the time of this writing, the major tech giants (like Google, IBM, Amazon, and so on) are drastically adopting these technologies to build and manage their large scale infrastructures [15]. On the other hand, however, many other organizations, especially small and medium enterprises, are reluctant to launch their services through the SDS concept. The reason is that they generally experience hard difficulties in the selection of containerization softwares and tools that satisfy their business needs and visions [16]. To make things worse, a variety of containerization tools and management instruments are continuously emerging, developers are constantly introducing new features in their latest updates. Accordingly, their integration within an operating framework still appears as an important barrier to face.

Starting from these premises, the work presented herein intends to shed some light on the key enabling technologies of SDS deployments based on container networking. Specifically, the conducted study provides a three-folded contribution. First, it explores containerization technologies, including both container engines (i.e., LXC, Docker, LXD, Rkt, and Kata container) and orchestrators (i.e., kubernetes, docker compose, docker swarm, OpenStack, Nomad, Apache Mesos, Bistro, ECS, Cloud Foundry, and OpenShift). Second, it analyzes the main supporting tools that offer advanced and additional features to SDSs, such as load balancing, service discovery, user interfaces, and communication protocols. Third, it defines a set of qualitative Key Performance Indicators (KPIs) through which carrying out a preliminary comparison of the reviewed key enabling technologies. The resulting analysis clearly shows pros and cons arising from an integration of containers, container orchestrators and supporting tools and provides high-level guidelines and constructive comments to foster the widespread usage of SDSs in the near future.

1.2 Containerization Technologies

As anticipated in the previous Section, the main rationale behind upcoming SDSs is based on container networking. As depicted in Figure 1.1, resources, services, and applications are implemented as containers and distributed across network elements and clouds. Differently from traditional VMs, containers installed over the same physical machine share the same operating system. But, at the same time, they may still experience a good level of isolation. Also, local operations, like loading and turn-off procedures, could register limited latencies. On the other hand, a logically centralized orchestrator automates the management of containers lifecycle. This Section reviews containerization technologies available in the current state of the art.

1.3 Container Engines

Popular container engines are: LXC, LXD, Docker, Rkt, and Kata Containers.

1.3.1 LXC(Linux Containers)

It is an open-source software supported by Canonical Limited and in active development since 2008. Its first stable Version 1.0 was released on February 20, 2014. Containers in LXC rely on the Linux kernel containment features[8]. This functionality allows the user to run multiple images on a single host. The isolation between the containers is provided through kernel namespaces. Resource management, instead, is done through cgroup [16]. It supports multiple applications in a container and provides partial portability across Ubuntu distributions only [16]. These features make LXC more performant than standard VMs [17].

1.3.2 Docker

It is one of the leading containerization tools in the industry. The initial version of Docker was released on March 13, 2013. It uses the features of Linux kernel namespaces and cgroups to completely isolate applications and the underlying operating system. It also allows to run a single process in a container [8, 16, 18]. Docker extends the Linux container technology by creating portable, easy to use, and flexible images [18,

19, 20]. It creates self-elastic clusters with a size managed according to the workload and provides enhanced elasticity because of the fast loading speed of container [11].

1.3.3 LXD

Built on top of LXC, it provides new and better user experience. It is a free and open-source software, built under the Apache 2.0 license[1]. The initial Version 0.1 of LXD was released on February 13, 2015. Differently from LXC, it offers new features like a new single command line tool to enhance user experience for containers management. Moreover, multiple applications can run in a single container [21]. LXC containers can be easily used through REST API and a CLI clients.

1.3.4 Rkt

Its first version was released on November 27, 2014 by CoreOS (acquired by RedHat). It allows to run multiple isolated images sharing a common kernel namespace. Rkt container runtime is interoperable, secure, and open-source. It also provides security in various aspects. For example, it validates the authenticity of the image after downloading by cross checking publisher's image signature [16]. The work presented in [16] demonstrates that Rkt emerges as a suitable choice in computational and data intensive high performance application environment.

1.3.5 Kata Containers

It is an open source project available under the Apache 2.0 license. The first Version of Kata containers was released on May 22, 2018. It is a novel implementation of lightweight VMs, that seamlessly integrates within the container ecosystem. Therefore, benefits of both container and VMs are met, including high performance, workload isolation, and security. Kata Containers support the Open Containers Initiative (OCI)[2] [22].

[1] https://linuxcontainers.org/lxd/

[2] OCI is a project under governance of Linux Foundation. It was started in June 2015 by Docker, CoreOS, and other leaders of containerization industry to standardize the container formats and run-time.

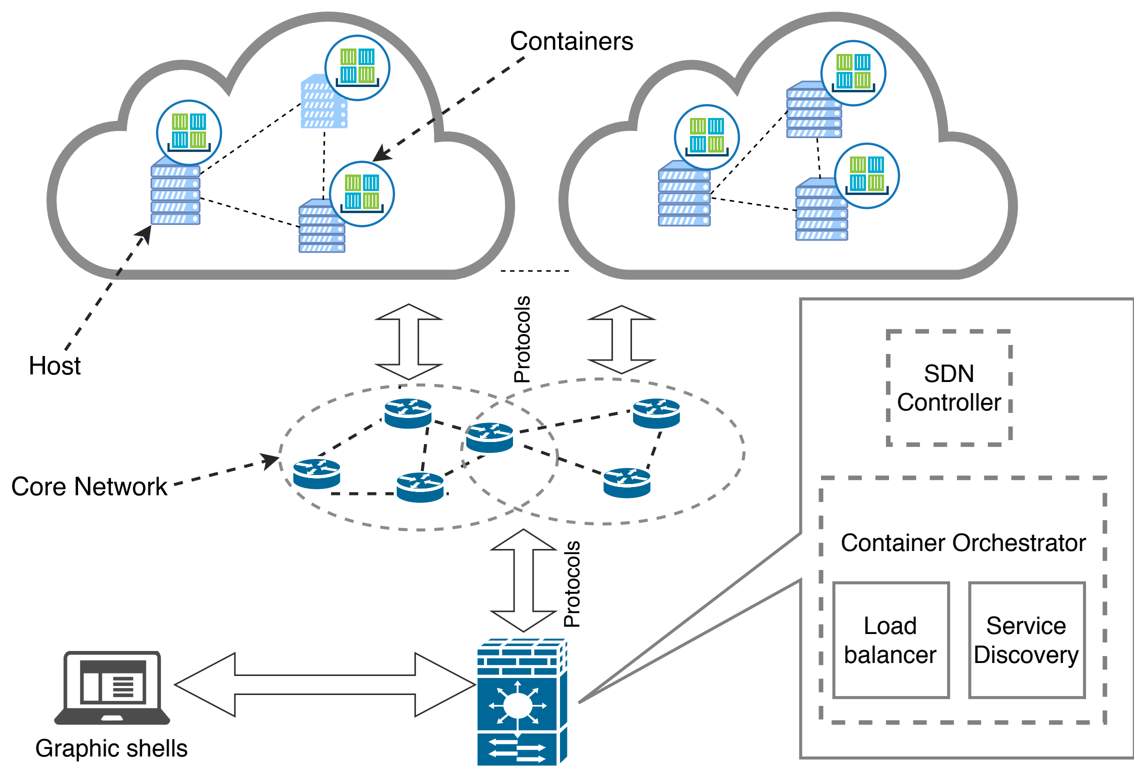


Figure 1.1: Big picture of Software Defined Systems based on container networking.

1.4 Container orchestrators

Popular container orchestrators are: Kubernetes, Docker Compose, Docker Swarm, OpenStack, Nomad, Apache Mesos, Bistro, ECS, Cloud Foundry, and OpenShift.

1.4.1 Kubernetes

It is an open-source container orchestrator for automating application deployment, scaling, and management across clusters of hosts (physical or VMs) [22]. It is freely available under the Apache License 2.0. Its first available Version was released on June 7, 2014. It was developed by Google under their experience of building container management solutions [22]. It is now maintained by the Cloud Native Computing Foundation. It works with a range of technologies and provides an orchestration layer for managing Docker containers on different physical entities [10]. Kubernetes uses a master-slave architecture. The slave hosts, namely nodes, are intended for running the containers assigned by the master. It also adds an abstraction layer on top of containers, namely Pod. Specifically, it represents a group of up to five containers that share storage, networking resources, IP address. Pods form an atomic unit of scheduling and are created or destroyed automatically. They can communicate with each other without any Network Address Translation (NAT). This ensures a very easy management of a multi-host cluster [23].

1.4.2 Docker Compose

It is a solution developed by Docker for creating and running applications, including multiple Docker Containers [19]. The first production ready Version 1.0 of Docker Compose was released on October 16, 2014. It helps to configure applications and containers by using a YAML file with a single command [18].

1.4.3 Docker Swarm

It is the Docker's local clustering solution developed by Google. While the standard Docker API launchs the containers, Swarm takes care of selecting the appropriate hosts for running containers [23],[18]. It can group together several hosts, allowing the user to manage them as a unified cluster by using the Swarm CLI utility[19]. It combines multiple Docker engines into a single virtual engine. For managing and configuring

services in the containers, a specific discovery service can be used with Swarm. The advantage of Swarm is that it natively incorporates Docker API calls. This consequently makes easy to move large workloads and applications to different clusters [23].

1.4.4 OpenStack

It is an open-source operating system developed to manage and control large pool of computations in the cloud. It was developed in Python by NASA and Rackspace to handle massive infrastructure in the cloud[24][25]. Its initial version was released on October 21, 2010. It has already been used by several companies worldwide for managing Petabytes of distributed architectures, scaling to over 60 million VMs [25]. OpenStack provides many other services like multi-tenant security, monitoring, storage, and more [25]. Google Sponsors the OpenStack foundation [26]. It provides support for managing Bare-metal, VMs, and container based hosts [27][28].

1.4.5 Nomad

Hashi group's Nomad was developed in 2015. It is an open-source scheduler that uses a declarative job file for scheduling containerized applications. It follows an agent-based architecture using single binary that is responsible for rolling upgrades and draining nodes for rebalancing [29].

1.4.6 Apache Mesos

It is an open-source and low level clustering solution, that integrates with a high level framework to provide the complete orchestration [23]. Mesos combines the resources of the cluster (like CPU, RAM etc.) in a way that looks like single giant host to the developer [29]. It was originally developed by students of UC Berkeley RAD Lab in 2009. Apache announced its Version 1.0 on July 27, 2016. It comes with a distributed system kernel that provides applications with API's for resource management and scheduling in large-scale clustered environments. It allows developers to create their applications. A job scheduler tool can be used with it for scheduling and running tasks [26]. Since Mesos acts like a kernel of the distributed OS therefore, a framework i.e., Marathon is used with it for Container Orchestration.

1.4.7 Bistro

Facebook's Bistro is a closed-source scheduler that runs data-intensive batch jobs on distributed systems. The present public release of this technology is partial, including just the server components. It is designed to handle workload efficiently and to respond rapidly to the changing configurations. This is because Facebook stores a high amount of data in different formats and frequently runs batch jobs for transformation and transfer of data [30].

1.4.8 Amazon's Elastic Container Service (ECS)

It is a closed-source solution used in Amazon Web Services (AWS) for running, scaling, and securing container applications[1]. ECS can be used with any third-party hosted Docker image repository or accessible private registry, such as Docker Hub.

1.4.9 Cloud Foundry

It is an open-source, multi-cloud application platform as a service. The software was originally developed as a container-based architecture by VMware in 2011. Then, it was transferred to Pivotal Software, which is a joint venture by EMC, VMware, and General Electric. Cloud Foundry supports Docker images by connecting to the Docker registry, giving those enterprises that are already running Docker take advantage of all the platform capabilities that Cloud Foundry provides. It also supports the OCI initiative.

1.4.10 OpenShift

Developed by the RedHat under the Apache license 2.0 on May 4, 2011. It is a combination of open-source technologies for orchestration, including RedHat Enterprise Linux, OCI-standard containers, and Kubernetes for orchestration and management. OpenShift extends to give users their choice of frameworks, databases, and runtimes. OpenShift is a part of the CNCF Certified Kubernetes program, ensuring portability and interoperability for container workloads.

[1] https://aws.amazon.com/ecs/features/

1.5 Supporting Tools for Containerization

There are many tools supporting specific facets in container networking, including load balancing, service discovery, protocols, and user interfaces. They facilitate the orchestration and management of containers, depending on the business requirements.

1.6 Load balancing tools

Load balancing tools aims at spreading the inbound requests (i.e., the load) across the containers. In this way, they minimize the response time and increase the throughput [29]. The list of some popular load balancing tools for containerization are reported below.

1.6.1 Envoy

It was built at Lyft in C++. It is a distributed proxy designed for single applications and services. It may also serve as a communication bus and data plane for microservice architectures [29].

1.6.2 HAProxy and Bamboo

It is a very popular, fast, and reliable solution for load balancing, which ensure a high availability for TCP and HTTP-based applications. For years, it has become the standard load balancing tool. Thus, it is now shipped with most mainstream linux distributions. It can be integrated with almost all the existing technologies [29]. On the other hand, Bamboo is a tool that automatically configures HAProxy for web services deployed on the top of Apache Mesos and Marathon [29].

1.6.3 Kube-Proxy

It runs on every node of the kubernetes cluster. It works for the cluster's internal load balancing and service discovery [29].

1.6.4 MetalLB

It is a tool for load balancing in bare-metal Kubernetes clusters. This tool exists because Kubernetes does not offer default load balancing mechanisms for bare metal clusters [29].

1.6.5 NGINX

It is a load balancing tool for HTTP-based traffic. It offers further mechanisms for configuration and monitoring[1] [29].

1.6.6 Traefik

It is an open source tool that is gaining much popularity nowadays like HAProxy [29]. It is supported by every major cluster technology. The popular features of Traefik are the auto discovery and tracing of clusters (including Kubernetes, Mesos, Docker Swarm, Marathon, and Rancher).

1.6.7 Vamp-router

It is a service routing, load balancing, and filtering application. It updates the configurations through REST API or Zookeeper [2] [29].

1.6.8 Vulcand

It is a HTTP API management and micro services tool that uses Etcd as a configuration backend. With Vulcand, changes to configuration take effect immediately without restarting the service.

1.6.9 Service discovery tools

In container networking, it is not possible to manually assign applications to high number of containers. Instead, such a task is managed by a specific software, generally referred to as scheduler, that controls the lifecycle of containers. In this context, service discovery is used to determine where the container ended up with being scheduled.

[1] https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/
[2] https://github.com/magneticio/vamp-router

1.6.10 ZooKeeper

It is a software by Apache Software Foundation, originally developed at Yahoo. It is a key-value store for maintaining configuration information in distributed systems. ZooKeeper organizes the data in a file system (to this end, it basically uses tables, called znodes) [29]. Even if it emerges as a mature tool, its installation procedure appears complex [29]. This tool is sponsored by giants like Google, Microsoft, AWS, and Facebook.

1.6.11 Etcd

It is another key-value store developed by the CoreOS team in the GO language. The Etcd security feature allows TLS/SSL authentication between client and cluster Etcd nodes [29].

1.6.12 Consul

It is a key-value store developed by the HashiGroup in the Go language. Consul offers a multi-data center support for service registration, discovery, and health monitoring [29].

1.6.13 Mesos-DNS

It is a DNS-based customized solution for service discovery, working in Apache Mesos. Mesos-DNS is written in the Go Language. Moreover, it polls the active running processes on the Mesos architecture and exposes the running tasks through DNS and HTTP APIs [29].

1.6.14 SkyDNS

It is a DNS-based service discovery tool used with Etcd. It stores the service records in Etcd and updates their DNS records accordingly [29].

1.6.15 WeaveDNS

It is another DNS-based service discovery solution that allows containers to find other containers through their IP addresses.

1.6.16 SmartStack

The Airbnb smartstack writes the service registration in Zookeeper and dynamically configures the HAProxy for lookup [29].

1.6.17 Eureka

It was developed and deployed for the AWS (i.e., where Netflix runs) [29]. Specifically, Netflix's Eureka is a Rest-based service discovery tool used for load balancing and checks the failover of intermediary servers. It comes with its own load balancer.

1.7 User interfaces

Graphic User Interfaces (GUIs) and User Interfaces (UIs) provide an abstract access to the remote resources for supporting the monitoring and the management of cloud-based processes. Popular solutions include:

1.7.1 Rancher

It is an open-source software for delivering Kubernetes-as-a-Service for multi-cloud computing.

1.7.2 Clocker

It is a self-hosted and open-source container-management platform, built on the top of Apache Brooklyn. Clocker is used to easily deploy production grade Docker swarms or Kubernetes clusters to a range of clouds (including AWS, Azure, Google Cloud, IBM Softlayer, and IBM BlueBox)[1]. Moreover, it supports a wide range of cloud providers through the use of the jclouds toolkit [23]. It can be configured with cloud though deployment tokens and access keys. After this, it can automatically detect hosts and install a network with the support of service discovery tools like Weave or Project Calico [23].

[1] http://www.clocker.io

1.7.3 Tutum

It was used to build, deploy, and manage containerized applications across any cloud infrastructure. Tutum decouples the orchestration layer from the underlying infrastructure on which the application runs. Tutum works on the top of any infrastructure provider and users can choose the provider that best satisfies their requirements. Started in 2013, later acquired and integrated by Docker in 2015.

1.7.4 Portainer

It is a lightweight management UI, which allows to easily manage different Docker environments (like Docker hosts or Swarm clusters). Portainer is able to provide support for Linux, Windows, and OSX [1].

1.7.5 Kitematic

It is a powerful UI for managing containers[2]. Kitematic gives a one-click installation ability for running Docker on Mac.

1.7.6 DockStation

It is a developer-centric tool for managing Docker projects. Instead of lots of CLI commands you can monitor, configure, and manage services/containers by using a GUI. DockStation supports MacOS, Linux, Ubuntu, and Windows.

1.7.7 Panamax

It is a browser rendered GUI for pulling together image compositions. It supports Docker.

1.7.8 Docker UI

It provides a simple interface into the currently running docker VM and allows the users to browse/check the state of installed containers.

[1] https://github.com/portainer/portainer
[2] https://kitematic.com/

1.7.9 Docker Compose UI

It is a UI for Docker containers. Differently from other dashboards, it appears like a browser-based interface to manage deployed container compositions.

1.7.10 Shipyard

It is another UI for managing Docker Containers. Within few seconds it is ready for the login and gives the snappy looking dashboard. Shipyard has a simple installation procedure. After pulling down a script, simply run it, and Shipyard pulls down a number of images and spins them up[1].

1.8 Protocols

The communication between the different components in container networking is performed with the help of protocols. Main solutions are: OpenFlow, NETCONF, and RESTCONF.

1.8.1 OpenFlow

Born in 2008, it was considered as the first standard for SDN [31]. Therefore, it is a well-known communication protocol that gives access to the physical components (i.e., routers or switches) of the data plane [32], and adapt their functionalities to changing business requirements. Today, all the standard manufacturers of network devices provide support for OpenFlow in their devices.

1.8.2 NETCONF

The Network Configuration Protocol (NETCONF) provides a simple mechanism to manage, configure and upload configurations of a network device. It allows the device to expose a full and formal API, used by the applications to receive configuration data sets [33].

[1] https://dzone.com/articles/managing-docker-containers-with-shipyard

1.8.3 RESTCONF

NETCONF defines the configuration data stores and a set of CRUD operations (that include create, read, update and delete) useful to access to these data stores. RESTCONF uses HTTP methods to provide CRUD operations on a conceptual datastore containing YANG-defined data [34].

1.9 Cross comparison of SDS enabling technologies

In order to remark the pros and cons of SDSs enabling technologies and to shed some lights on their joint usage within a more complete framework, this Section defines a set of KPIs and proposes a qualitative cross comparison.

First of all, the qualitative KPIs defined for evaluating container engines include the development language, the developing company, the community support, the reference operating system, the licensing type, the OCI compliance, and the support for IPv6. The resulting cross comparison is reported in Table 1.1. It is evident that LXC, LXD, Docker, Rkt, and Kata containers are open-source technologies, promoted by strong community, and working with Linux. All the container engines, except LXD, are compliant with the OCI guidelines. Moreover, excepting Kata containers, all the other engines support IPv6. Among the others, Docker emerges as a powerful technology that works on multiple platforms (i.e. Linux, Windows, and MacOS).

The set of qualitative KPIs used to evaluate containers orchestrators include the development language, the developing company, the community support, the licensing type, the builtin scheduler, the native load balancing feature, the native service discovery, the reference operating system, the cloud support, the bare metal support, and the possibility to handle clustering. The resulting cross comparison is shown in Table 1.2. The conducted study demonstrates that Kubernetes and OpenStack are the open-source orchestration technologies that come up with built-in scheduler, load balancer, service discovery native functionalities. They support multiple operating systems, cloud, bare metal, and clustering features. They also have a strong community support, backed by Google and Cloud Native Computing Foundation (CNCF). On the other hand, instead, other technologies guarantee limited features (they are used in the industry only for specific purposes). Among all the investigated solutions, Kubernetes

Table 1.1: Cross comparison among containerization engines.

Engine	**Language**	**Developers**	**Community support**	**OS Support**	**Licencing type**	**OCI**	**IPv6**
LXC	C, Python, Lua	Canonical Ltd	Canonical Ltd and Ubuntu* (Forum & Github)	Linux	Opensource, GNU LGPL v.2.1	✓	✓
LXD	Go	Canonical Ltd	Canonical Ltd and Ubuntu** (Forum & Github)	Linux	Opensource, Apache 2.0	✗	✓
Docker	Go	Docker, Inc.	Docker, Inc. (Community, Forum, Blog & Github)	Linux, Win, Mac	Opensource, Apache 2.0	✓	✓
Rkt	Go	CoreOS (RED-HAT)	Red Hat Inc., Github	Linux	Opensource, Apache 2.0	✓	✓
Kata Containers	Go	The OpenStack Foundation	OpenStack foundation, Blog, Github,	Linux	Opensource, Apache 2.0	✓	✗

*LXC 1.0 will be supported until June 1st 2019 and LXC 2.0 until June 1st 2021.
**The current LTS of LXD if 3.0, which will be supported until June 2023

allows a huge amount of flexibility for containerization networking. It can be used with almost all the container engines to provide agility.

The joint usage of container engines and supporting tools is evaluated below.

Table 1.3, for instance, focuses on load balancing tools. It clearly emerges that NGINX can be natively used with all the reviewed container engines. HA-Proxy, instead, can be used with LXC, LXD, and Docker. The rest of load balancing tools, excluding Bamboo, could be used with all the container engines thanks to the additional plugins implemented in both Kubernetes orchestrator and Etcd tool. Among the containers, Docker can work with all the considered load balancing tools. The Bamboo tool, instead, provides a very scarce support for the container engines.

Table 1.4 illustrates the possible joint usage of service discovery tools and container engines. Also in this case, many container engines are natively or indirectly (i.e., with the usage of integration instruments provided by some container orchestrators) compatible with the reviewed service discovery tools. Nevertheless, a very scarce integration is observed for both SmartStack and Eureka.

Table 1.5 concludes the analysis by showing the usage of user interfaces with the considered container engines. In this case, only the Rancher graphical interface is supported by all the container engines. The rest of the tools, instead, have been specifically conceived for Docker.

Table 1.2: Cross comparison among containers orchestrators.

Name	Language	Developers	Community support	Licensing	S	LB	SD	Platform	Cloud	BM	Cluster
Kubernetes	Go	Google	Cloud Native Computing Foundation	OpenSource (Apache License 2.0)	✓	✓	✓	Linux, Mac, Windows	✓	✓	✓
Docker Compose	Python	Docker, Inc	Docker, Inc. (GitHub)	OpenSource (Apache License 2.0)	✓	✗	✗	Linux, Mac, Windows	✓	✓	✗
Docker Swarm	Python	Docker, Inc	Docker, Inc. (GitHub)	OpenSource (Apache License 2.0)	✓	✗	✗	Linux, Mac, Windows	✓	✗	✓
OpenStack	Python	NASA/ Rackspace	OpenStack Foundation/ Google	OpenSource (Apache License 2.0)	✓	✓	✓	Ubuntu	✓	✓	✓
Nomad	Go	Hashicorp	Hashicorp (GitHub)	Mozilla Public License 2.0	✓	✓	✗	Linux, Mac, Windows	✓	✓	✓
Apache Mesos	C++	UC Berkeley RAD Lab Students	Apache Software Foundation(Blog, GitHub)	OpenSource (Apache License 2.0)	✓	✗	✗	Linux, Mac, Windows	✓	✓	✓
Bistro	C++	Facebook Inc.	Facebook, Inc.(Bistro Group, GitHub)	Closed source (BSD License)	✓	✗	✗	Ubuntu	✓	✗	✓
ECS	JS etc	Amazon.com, Inc.	Amazon.com, Inc.	Paid	✓	✓	✓	Linux, Windows	+(only AWS)	✗	✓
Cloud Foundry	Go, Ruby, Java	VMWare	Cloud Foundry Foundation Community and GitHub	OpenSource (Apache License 2.0)	✓	*	*	Linux, Mac, Windows	✓	✓	*
OpenShift	Go, AngularJS	RedHat Software	Red Hat (OpenShift personal Blog and Community)	OpenSource (Apache License 2.0)	+	+	+	Linux, Mac, Windows	✓	✗	✓

S = Scheduler; LB = Load Balancer; SD = Service Discovery; * through BOSH or Kubernetes; + through Kubernetes

Table 1.3: Joint usage of load balancing tools and container engines.

Tool	LXC	LXD	Docker	Rkt	Kata Containers
Bamboo	✗	✗	✓	✗	✗
Envoy	*	*	✓	*	*
HAProxy	✓	✓	✓	*	*
Kube-Proxy	*	*	*	*	*
MetalLB	*	*	*	*	*
NGINX	✓	✓	✓	✓	✓
Traefik	*	*	✓	*	*
Vamp-router	*	*	✓	✓	*
Vulcand	**	**	✓	**	**

* through Kubernetes; ** through Etcd

Table 1.4: Joint usage of service discovery tools and container engines.

Tool	LXC	LXD	Docker	Rkt	Kata Containers
ZooKeeper	Yes	Yes	Yes	*	**
Etcd	Yes	**	Yes	Yes	**
Consul	**	**	Yes	**	**
Mesos-DNS	***	***	***	***	✗
SkyDNS	****	****	****	****	*****
WeaveDNS	**	**	Yes	**	**
SmartStack	✗	✗	Yes	✗	✗
Eureka	✗	✗	Yes	✗	✗

through *Zetcd and Etcd3; **Kubernetes; ***Apache Mesos; ****Etcd; *****combination of Etcd and Kubernetes

Table 1.5: Joint usage of user interfaces and container engines.

Tool	LXC	LXD	Docker	Rkt	Kata Containers
Rancher	✓	✓	✓	*	*
Clocker	✗	✗	✓	✗	✗
Tutum	✗	✗	**	✗	✗
Portainer	✗	✗	✓	✗	✗
Kitematic	✗	✗	✓	✗	✗
DockStation	✗	✗	✓	✗	✗
Panamax	✗	✗	✓	✗	✗
Docker UI	✗	✗	✓	✗	✗
Docker Compose-UI	✗	✗	✓	✗	✗
Shipyard	✗	✗	✓	✗	✗

*through Kubernetes; ** Docker acquired and integrated it

1.9.1 Summary

SDS allow to define, deploy, and use virtual resources, services, and applications across the network elements and clouds. To fulfill strict requirements (including, for instance, flexibility, isolation, and high performance), their enabling technologies include container engines, container orchestrators, and many other supporting tools. This chapter deeply reviewed the state of the art on enabling technologies for SDS in the context of container networking and provided a qualitative cross comparison, based on a specific set of Key Performance Indicators. The conducted study clearly demonstrated that Docker is emerging as the leading container engine. In fact, it offers a strong set of features and allows the usage of a large number of supporting tools. At the same time, it is remarked that Kubernetes appears as a very promising container orchestrator because it comes up with its own scheduler, load balancer, and service discovery features.

This study was aimed at providing a clear roadmap for the selection of technologies to the enterprises who are reluctant to adopt container networking for launching their

services and applications and deploy SDS based virtualized service infrastructures, respectively. Additionally, this chapter provides a basis for the selection of best available emerging technologies for practical deployment and quantitative cross-comparison in the next chapter.

2

Quantitative Cross-Comparison of Emerging Technologies for Virtualized Service Infrastructures

Container networking is emerging as a game-changer paradigm for the deployment of SDS-based virtualized service infrastructures in a faster and reliable way. Nevertheless, Small and Medium Enterprises are still skeptical to revise their business in this direction because of the absence of deep studies showing its effectiveness in real deployment environments, the selection of technologies is a huge challenge. To bridge this gap, this chapter presents a quantitative cross-comparison of cutting-edge technologies for container networking (including Docker as a container engine, Docker Swarm and Kubernetes as orchestrators, bare-metal and OpenStack cloud as deployment platform), properly deployed and integrated to realize a virtualized service infrastructure within a commercial workstation. Initial experimental tests are conducted to identify the most suitable combination of technologies for high-load environments and later, in the case of industrial smart farm, respectively. The obtained results have been discusses in detail. The set of emerging technologies that highlights best results herein are used for practical deployment in the hierarchical T-SDN based framework presented in Chapter 3.

2.1 Motivation and overview

As well known, virtualization gives the opportunity to optimize the usage of hardware resources and to conceive advanced and isolated services through a common (frequently distributed) platform [35, 36]. Since decades, virtualization was achieved by using Virtual Machines, that are emulator of computers having their own kernels, managed by a hypervisor system [37, 38]. More recently, instead, a novel virtualization technology, namely container, is gaining momentum. Different from Virtual Machines, containers act as high-level applications running on top of the same Operating System [39]–[27], thus offering quick startup time, rapid application loading, less memory space requirement, a swift recovery from failure, and portability (e.g., build once and run anywhere). Furthermore, thanks to the container networking paradigm, containers can interact with each other, while paving the road to a new way to conceive applications with less cost and reduced time-to-market and revising the industry business [41]–[43].

At the time of this writing, Tech Giants (like VMWare, Xen, Microsoft, Amazon, and Google) already offer subscription-based solutions for deploying virtualized service infrastructures in the cloud [44]. On the other hand, Small and Medium Enterprises (SMEs) would like to implement their own virtualized service infrastructures into local computing environments (apart the deep control and personalization of implemented functionalities, it would erase heavy subscription fees related to the usage commercial clouds) [45, 46]. Theoretically, this is possible thanks to the presence of a number of open-source technologies enabling container networking [47]–[72]. Nevertheless, SMEs are still skeptical about the usage because of the following two main reasons. First, the effective usage of container networking requires the selection and the joint integration of container engines (i.e, the technology that effectively implements the container), orchestrator (i.e, the technology that is responsible for managing, scheduling, and deploying individual containers for applications within the cluster, while offering load balancing and service discovery functionalities), and many other supporting tools that make possible their implementation and usability in specific platforms. Unfortunately, this task cannot be successfully and quickly achieved by SMEs with limited technical skills and revenue to spend on research and development activities [73]. Second, the scientific literature either investigated the behavior and the performance of containers against Virtual Machines or the technologies enabling container networking

separately (for more details, see the summary of the state of the art discussed in Section 2.2). Thus, there are no contributions that address a quantitative investigation of the joint integration of containerization technologies in local computing environments, along with a clear description of the pros and cons characterizing the popular solutions available today.

To bridge this gap, this work presents an experimental cross-comparison of cutting-edge technologies for container networking, properly integrated to realize a virtualized service infrastructure in local computing environments. Specifically, a centralized orchestrator is configured to offer service discovery and load balancing functionalities (i.e., management of clients' requests and their distribution to available containers). At the same time, some containers are deployed to expose resources and advanced services to remote clients. By considering the main outcomes of a qualitative analysis of containerization technologies presented (by the same authors of this work) in [74], the set of technologies investigated herein includes: (1) Docker as the container engine, (2) Docker Swarm and Kubernetes as orchestrators with load balancing and service discovery capabilities, (3) bare-metal and OpenStack cloud as deployment platforms and (4) Docker-compose, Docker-Machine, Kubeadm, and Flannel as supporting tools for scheduling and deployment functionalities. Due to the possible combinations between the selected orchestrator technologies (i.e., Docker Swarm and Kubernetes) and deployment platforms (i.e., bare-metal and OpenStack cloud), four different experimental testbeds have been implemented within a commercial workstation having computing capabilities that are comparable to those available in most of SMEs realities.

Initial experimental tests are conducted to identify the most suitable combination of technologies for high-load environments (i.e., when the whole system is in change of managing a high traffic load), where many clients contact the virtualized service infrastructure to download files of large size. Clients' requests are generated through the Poisson statistics from a laptop, connected to the aforementioned virtualized service infrastructures through a real-world network. Moreover, different KPIs, that include CPU utilization, memory footprint, network load, connection delay, and request completion time, are measured by assuming an average number of requests per unit of time equal to 5 and 10 requests/minute. Obtained results demonstrate that the integration of Docker and Kubernetes on the bare-metal deployment platform provides better performance in terms of percentage of CPU used by containers, distribution of

the network load over the time and among the deployed containers, connection delay, and request completion time, while registering a slight (but still acceptable) increment of the memory footprint.

To provide further insight, the behavior of the more performant technologies is also evaluated in a more complex scenario of a smart farm use case. Differently from the previous case, a variable number of drones flying in a smart farm is now emulated on two laptops, connected to the virtualized service infrastructure by means of two different wireless access points. Drones generate livestock pictures with a Poisson statistic and deliver them to the virtualized service infrastructure. The service orchestrator forwards these pictures to available containers, which will recognize the type and the number of animals within the pictures through a machine learning-based image processing elaboration. The outcome of this processing is finally delivered to a remote server for monitoring purposes. This new campaign of experimental tests remarks that the behavior of the virtualized service infrastructure is not drastically influenced by the presence of mobile users: in any case, the service orchestrator is able to properly forward users' requests to available containers, while guaranteeing a uniform balancing of computing tasks. The execution of heavy tasks inevitably brings to higher computing and memory requirements, while reducing the overall traffic load. Nevertheless, the tests fully confirm that the combination of Docker and Kubernetes on the bare-metal deployment platform represents a suitable solution for effectively exploiting container networking capabilities in real deployments with local (hence limited) computing capabilities.

The rest of the chapter is organized as follows: Section 2.2 presents background on container networking and reviews the state of the art. Section 2.3 deeply describes the technologies taken into the account in this study and illustrates the implemented experimental testbeds. The quantitative cross-comparisons, between the identified technologies for container networking are presented in Section 2.4. Finally, Section 2.5 concludes the work and draws future research activities.

2.2 Background on container networking

Virtualization generally refers to the implementation of an additional layer above the host Operating System, aiming at providing a separate environment for running appli-

cations with virtually allocated resources [47]. In the past, a popular hypervisor-based approach was adopted for virtualization, known as Virtual Machines. A Virtual Machine represents an emulated computer within a virtualization environment, having its own guest Operating System and kernel, and works above the host Operating System [44]. More recently, instead, containers emerged as a game-changer in the virtualization context. They can implement services and networking functionalities at the application layer of a given Operating System. Since containers emotively share the same Operating System, they generally ensure better usage of hardware resources through virtualization [47]. At the same time, containers embrace their own binaries, libraries, and runtime component, provide portability and agility (i.e., once built, they run anywhere), and introduce a new disruptive way to design and deploy future applications and services [44]. The comparison between containers and Virtual Machines were investigated in many works, including [38, 40, 44, 48]$^{-}$[68][25]$^{-}$[80]. The majority of these studies compare CPU and memory usage, disk input/output, execution time, and network load of both virtualization technologies and clearly highlight that containers perform better or equal to Virtual Machines. According to the container networking paradigm, containers can interact with each other, while offering the opportunity to deploy novel applications over distributed and virtualized environments [11]. To reach this goal, however, it is necessary to integrate technologies implementing container engine, orchestrator, load balancer, and service discovery tools within a specific deployment platform.

Regarding the deployment platform, two main solutions are adopted today: bare-metal and OpenStack cloud. Bare-metal refers to the conventional physical system, where available hardware resources (e.g., compute, storage, and other resources) are managed by the computer's main Operating System. Here, all the resources are acquired by a single user [81]. During the latest years, instead, OpenStack emerged as a leading open-source solution for the setup of both small and large scale cloud operating environments [27]. It provides virtualized resources and workspace to multiple independent users[25][79]. Surely, containers can be deployed within both bare-metal and OpenStack deployment platforms [26]$^{-}$ [84]. However, the work discussed in [69, 70] remark that containers deployed in OpenStack cloud load quicker as compared to bare-metal systems. However, there are no other contributions that investigate the performance of containers deployed on bare-metal and OpenStack cloud in other directions.

Table 2.1: Summary of the investigated state of the art.

Reference Papers	Container resource utilization	Comparison with Virtual Machine	Network performance	Orchestration	Implementation in the Cloud	Implementation on bare-metal	load balancer
[38]						✓	
[44]	✓		✓		✓		
[48]		✓	✓			✓	
[17, 50, 68]	✓	✓				✓	
[49, 51, 52, 58, 62]–[66]	✓	✓	✓			✓	
[53]		✓				✓	
[54, 67]	✓			✓		✓	
[55, 56, 85]	✓					✓	
[59]		✓					
[60, 86]	✓	✓					
[61]						✓	
[7]	✓	✓	✓		✓		
[69]					✓	✓	
[70]	✓	✓			✓	✓	
[71]				✓			
[72]	✓			✓	✓		
[87, 88]	✓		✓				
[89]	✓			✓			
[90]	✓				✓		
[91]	✓						
This work	✓		✓	✓	✓	✓	✓

In the container networking context, the deployment of scalable applications over multiple nodes is achieved through an orchestrator. Specifically, the orchestrator automates and controls many tasks, such as provisioning and deployment of containers, redundancy, and availability of containers, scaling, and removing containers to spread

application load evenly across host infrastructure. The role of orchestrator is very important in large and dynamic environments. Surveys conducted on this topic report that there is a need for research activities to evaluate the impact of technologies implementing orchestration functionalities in the container networking context [71, 72].

The horizontal distribution of traffic across multiple containers in a cluster is carried out by a load balancer. This task is necessary to prevent containers from getting overloaded and ensure service availability. At the time of this writing, [38] is the only contribution presenting some technological details related to the orchestrator used in their test. Also in this case, however, the contribution does not discuss a comparison among different technologies offering the same functionalities.

A qualitative cross-comparison of emerging technologies for container networking (including container engines, orchestrators, load balancers, and service discovery tools) have been performed by the same authors of this work in [74]. The results of the comparison highlight that Docker is a powerful emerging container engine that works on multiple platforms, and that Kubernetes and Docker Swarm are the open-source orchestration technologies that come up with built-in scheduler, load balancer, and service discovery functionalities.

Table 2.1 summarizes the topics covered by the current scientific literature, including the analysis of container resource utilization, comparison with Virtual Machine, network performance, orchestration, implementation in the cloud and bare-metal, and load balancer implementation. It emerges that at the time of this writing, to the best of the Author's knowledge, there are no contributions that implement the integration of state of the art enabling technologies for container networking. For this reason, there is a need for a quantitative cross-comparison of cutting-edge technologies for container engine, and orchestrators with load balancing and service discovery features deployed on multiple deployment platforms.

2.3 Integration of cutting-edge technologies enabling the container networking paradigm

This section presents a detailed description of the developed virtualized service infrastructure based on container networking. Starting from the main outcomes of the qualitative cross comparison of container networking technologies presented in [74],

Docker has been selected as the reference container engine, whereas Docker Swarm and Kubernetes are considered as possible orchestrators offering load balancing and service discovery capabilities. Note that the integration of these technologies requires the usage of additional supporting tools (i.e., Virtualbox, Docker-compose, Docker-Machine, Kubeadm, and Flannel), as discussed below. The whole virtualized service infrastructure, instead, has been realized through bare-metal and OpenStack cloud deployment platforms.

Without loss of generality, the developed scenario embraces virtual machines. One of them hosts the orchestrator and a container application. The rest of the virtual machines, instead, implements container applications only. The virtualized service infrastructure is able to receive multiple requests issued through the HTTP protocol. In particular, the former request is handled by the container placed within the virtual machine hosting the orchestrator. In case of multiple requests, instead, the orchestrator activates the load balancing functionality to distribute them across the rest of the available containers.

Given the possible combinations of orchestrator technologies and deployment platforms, four different experimental testbeds have been realized:

- **Testbed 1: integration of Docker and Docker Swarm on bare-metal**. In the bare-metal deployment platform, the Docker-Machine supporting tool is used to deploy the whole virtualized service infrastructure. First of all, four virtual machines are created through Virtualbox. According to the Docker Swarm technology, one of these virtual machines has been configured as the Swarm Manager, which represents the orchestrator running service discovery and load balancing functionalities. The setup of the Swarm Manager implies the creation of the Docker Bridge and the Docker API Proxy System. The former provides a communication bus for interconnected containers. The latter defines a unified interface between the real-world network and the virtualized service infrastructure. A join-token provided by the Swarm Manager is used by other virtual machines for establishing a connection with the Swarm Manager and the Docker API Proxy system. After joining the Swarm Manager, the other three virtual machines are configured as slave nodes, namely Swarm Workers. From this moment on, Swarm Manager and Swarm Workers can also interact with each other through

the Docker Bridge. At the same time, Swarm Manager and Swarm Workers can communicate with the external real-world network through the Docker API Proxy System, as depicted in Figure 2.1.

On the Swarm Manager, a YAML file is used to describe the structure of the virtual environment to be deployed and the services to be executed in each container. Then, the Docker-compose supporting tool is used to launch the container application on each virtual machine.

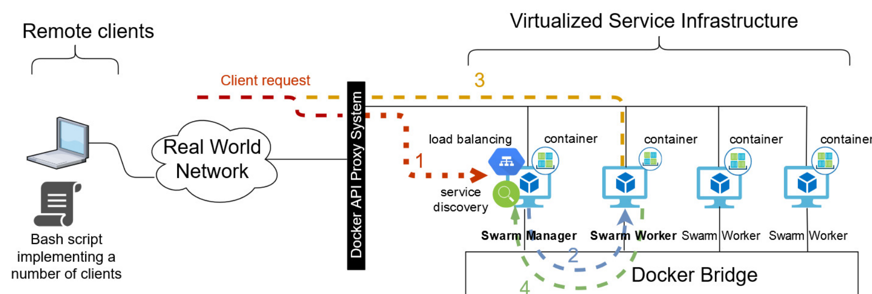


Figure 2.1: Testbed 1: integration of Docker and Docker Swarm on bare-metal.

Now, a client can issue its request, that will be delivered to the developed virtualized service infrastructure through the real-world network. Figure 2.1 explains the resulting communication pattern. First, the client request is received by the Swarm Manager via the Docker API Proxy System (step 1). Then, the Swarm Manager distributes the incoming requests among available containers. The distribution follows a round-robin approach, starting from the container available within the nodes hosting the Swarm Manager. The example reported in Figure 2.1 shows that the request is forwarded to the container installed on the first Swarm Worker through the Docker bridge (step 2). Then, the Swarm Worker answers to the given client by sending back the requested content through the Docker API Proxy System (step 3). Finally, the Swarm Worker updates its status with the Orchestrator on the Docker Bridge (step 4).

- **Testbed 2: integration of Docker and Kubernetes on bare-metal**. This testbed still uses the bare-metal deployment platform. The virtualized service infrastructure depicted in Figure 2.2 highlights the presence of four virtual machines created through Virtualbox. Docker and Kubernetes packages are installed into each virtual machine.

The Kubernetes cluster embraces two kinds of nodes: Kube Master and Kube Worker. The former one is the orchestrator, which implements service discovery and traffic load functionalities. The latter one, instead, refers to the generic node controlled by the orchestrator and exposing a service or a resource. On the other side, the Flannel supporting tool is properly configured to create an overlay network that interconnects the nodes belonging to the Kubernetes cluster. The Kube Manager is declared and a Kube Proxy System is created, which manages the communication with the client. The join-token returned by the Kube Manager is used by the other virtual machines for establishing a connection with the orchestrator. From now onwards, the Kube Workers are connected with the Kube Master, Kube API Proxy System for communication with the client, and the Flannel overlay network for exchanging internal information.

A YAML file is used to describe the structure of the virtual environment to be deployed and the services to be executed in each container. Then, the deployment of containers is created and exposed in the cluster through Kubernetes.

Now, the virtualized service infrastructure is ready to handle the client requests. Figure 2.2 describes the resulting communication pattern: First, the client request is received by the Kube Master via the Kube API Proxy System (step 1). Then, the Kube Master distributes the incoming requests among available containers. The distribution exploits the round-robin approach, which starts by delivering the first client request to the container installed on the same machine of the Kube Master. The example shown here demonstrates that the request is forwarded to the container deployed on the first Kube Worker through the Flannel overlay network (step 2). Then, the Kube Worker answers to the given client by sending back the requested content through the Docker API Proxy system (step 3) Finally, the Kube Worker exchange its state with the Kube Master on the Flannel overlay network (step 4).

- **Testbed 3: integration of Docker and Docker Swarm on OpenStack cloud**. This testbed is based on a different deployment platform, which is an OpenStack cloud. It firstly requires the configuration of key components of the OpenStack cloud, that are: 1) Nova, the primary computing engine behind OpenStack), 2) Swift, a storage system for objects and files, 3) Cinder, a block storage

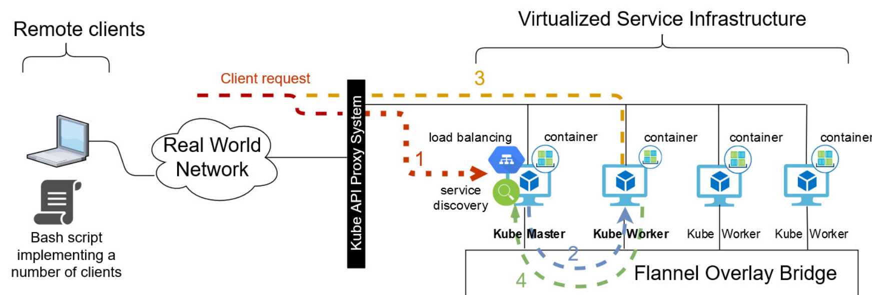


Figure 2.2: Testbed2: integration of Docker and Kubernetes on bare-metal.

component that allows access to specific locations on a disk drive, 4) Neutron, the entity providing the networking capability to the overall virtualized environment, 5) Keystone, the security manager, and 6) Glance, the component providing virtual machine images. Once installed on the physical machine, OpenStack cloud can be initially managed through its graphical user interface or command-line instructions. Different from the bare-metal deployment platform, OpenStack cloud does not require the Virtualbox tool. It is able to autonomously create virtual machines, which are simply referred to as instances.

By default, OpenStack cloud creates a virtual network infrastructure made up of private and public networks. All the instances are installed within the private network. They have their private IP addresses and can communicate with the external real-word network through a virtual router running the Network Address Translation (NAT) protocol.

The integration of Docker and Docker Swarm into the private network of the OpenStack cloud is achieved by means of the same approach already explained for the Testbed 1. One of the OpenStack cloud instances is designed as the Swarm Manager. The setup of the Swarm Manager implies, as expected, the creation of both Docker Bridge and Docker API Proxy System which handles the client communication. The join-token returned by Swarm Manager is used by other virtual machines for establishing a connection with the Swarm Manager and the Docker API Proxy system. The other three virtual machines are configured as slave nodes, namely Swarm Workers.

From this moment on, Swarm Manager and Swarm Workers can interact with each other through the Docker Bridge and with the client through the Docker

API Proxy System, within the private network created in the OpenStack cloud, as depicted in Figure 2.3. The Docker-compose supporting tool has been used to launch a container application in each virtual machine. Now, the client requests can access to the virtualized service infrastructure. The resulting communication process has been presented in Figure 2.3. The Swarm Manager receives the client request via Docker API Proxy System (step 1). Likewise, Testbed 1, the Swarm Manager distributes the incoming requests to the available containers according to the round-robin technique. In this example, the request is forwarded to the container installed on the first Swarm Worker through the Docker bridge (step 2). Then, the Swarm Worker answers to the given client by sending back the requested content of the container through the Docker API Proxy System (step 3). Finally, the Swarm Worker updates its status with the Swarm Manager through the Docker bridge (step 4).

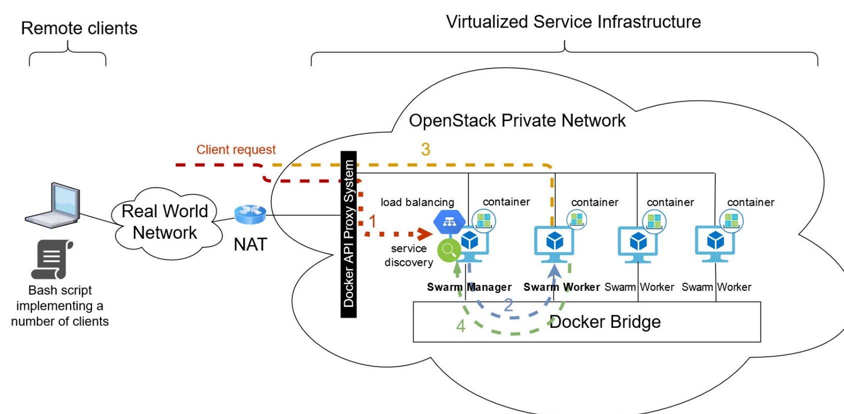


Figure 2.3: Testbed 3: integration of Docker and Docker Swarm on OpenStack cloud.

- **Testbed 4: integration of Docker and Kubernetes on OpenStack cloud**. In this final deployment, the OpenStack cloud is used to create the four instances within the private network. Then, Kube Manager and Kube Workers are configured as already discussed for the Testbed 2.

  Indeed, the Kube Manager is declared by means of the kubeadm tool. As mentioned before, a join-token be used by the other virtual machines for establishing a connection with the orchestrator, within the private network of the OpenStack cloud. Within the Kube Manager, a YAML file is used to describe the structure

of the virtual environment to be deployed and the services to be executed in each container. The Flannel supporting tool is properly configured to create an overlay network that interconnects the nodes belonging to the Kubernetes cluster. Also, in this case, a virtual router is implemented with NAT rules that connect the virtualized service infrastructure with the real-world network.

Then, Kubernetes creates and exposes the deployment of the containers in the cluster. From this moment on, the Kube Manager and Kube Workers are connected to the Kube API Proxy System for communication with the client and the Flannel overlay network for exchanging information.

Now the client can make a request, that will be delivered to virtualized service infrastructure. As seen in Figure. 2.4, the client's request is handled in the same pattern by the Orchestrator as already discussed in Testbed 2.

Now, the client requests can access to the virtualized service infrastructure. The resulting communication process has been presented in Figure 2.3. The Swarm Manager receives the client request via Docker API Proxy System (step 1). Likewise, Testbed 1, the Swarm Manager distributes the incoming requests to the available containers based on the round-robin technique. In this example, the request is forwarded to the container installed on the first Swarm Worker through the Docker bridge (step 2). Then, the Swarm Worker answers to the given client by sending back the requested content of the container through the Docker API Proxy System (step 3). Finally, the Swarm Worker updates its status with the Swarm Manager through the Docker bridge (step 4).

2.4 Cross comparison and performance assessment

This Section presents a cross-comparison and performance assessment of the reviewed and integrated cutting-edge technologies enabling the container networking paradigm. Specifically, two campaigns of experimental tests are discussed below. First, the behavior of four Testbeds described in Section 2.3 is investigated in a high-load environment, where many clients contact the virtualized service infrastructure to download files of large size. This study is useful to identify the most suitable combination of technologies ensuring better performance in computing environments typically available for Small

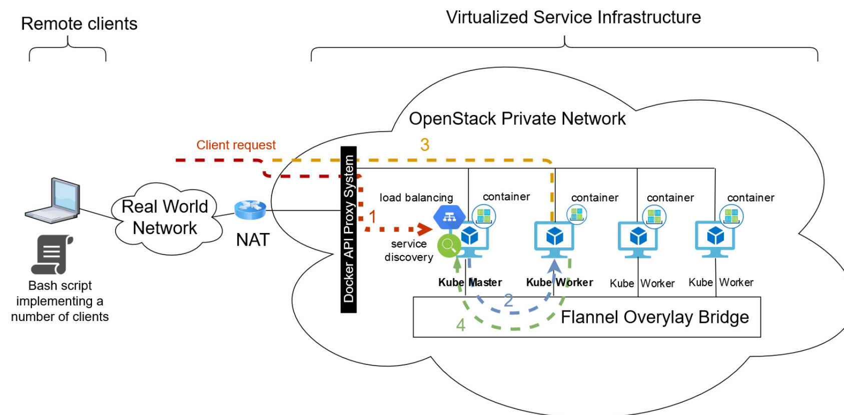


Figure 2.4: Testbed 4: integration of Docker and Kubernetes on OpenStack cloud.

and Medium Enterprises. Second, the performance of the identified virtualized service infrastructure is also evaluated in a smart farm use case, where containers are in charge of processing images provided by mobile drones for monitoring purposes. In this case, the analysis would highlight the promising capabilities offered by container networking in real deployments, exploiting local computing environments.

The computing architecture adopted in both tests is a commercial workstation with Intel® Xeon(R) E5-16200 CPU (made up of 8 cores working at 3.60 GHz each), 16 GHz RAM, and Ubuntu 18.04.2 LTS Operating System. Of course, the proposed implementation can be safely and easily extended to develop more complex scenarios, by using machines with higher computing capabilities.

Moreover, the conducted analysis considered two groups of KPIs, simply referred to as infrastructure KPIs and end-user KPIs. Infrastructure KPIs are introduced for describing the behavior of integrated technologies within the virtualization platform. They include:

- **CPU utilization**: it refers to the percentage of CPU utilized by the container. It has been monitored by running bash scripts on each virtual machine for gathering data through the Docker Stats command, which returns a live data stream of the containers.

- **Memory footprint**: it represents the amount of main memory consumed by the container, measured in MB. Similar to the previous case, it was collected by

executing bash scripts containing on each virtual machine using the same Docker Stats command.

- **Network load**: it reports the average amount of data sent by the container in a unit of time. Indeed, it is expressed in terms of Mbps. It was measured by running bash scripts inside the containers for monitoring data flow on the internal bridge of the container, using the brctl command.

On the other hand, the end-users KPIs are defined to evaluate the quality of service experienced by the clients willing to retrieve the video file exposed by the virtualized service infrastructure. They include:

- **Connection delay**: it is the amount of time required to establish the connection between the client and the virtualized service infrastructure at the transport level. It was measured in minutes by running a bash script running at the client-side, which exploits its TCP features.

- **Request completion time**: it refers to the amount of time required to download the whole video file. It was measured in minutes by running a bash script running at the client-side.

2.4.1 Cross-comparison in a high-load environment

In order to compare the behavior of the four Testbeds described in Section 2.3 in a high-load environment (i.e., when the whole system is in change of managing a high traffic load), the virtualized service infrastructure is configured to expose resources to remote clients. In particular, to verify the right capability of the load balancing functionality to distribute incoming requests among available containers, each container was configured to host the same data content, which represents a video file of 13 minutes (i.e., a trailer taken from YouTube), encoded at an average rate of 1.45 Mbps. A laptop, connected to the virtualized service infrastructure through the real-world network, is used to emulate many remote clients willing to retrieve the aforementioned video content. In this regard, a Python script is used to generate client requests, and the communication between clients and remote containers is established through the HTTP protocol. To test the impact of the traffic load on system performance, requests are generated according to the Poisson statistics, where the average number of requests per minute, λ, is set to

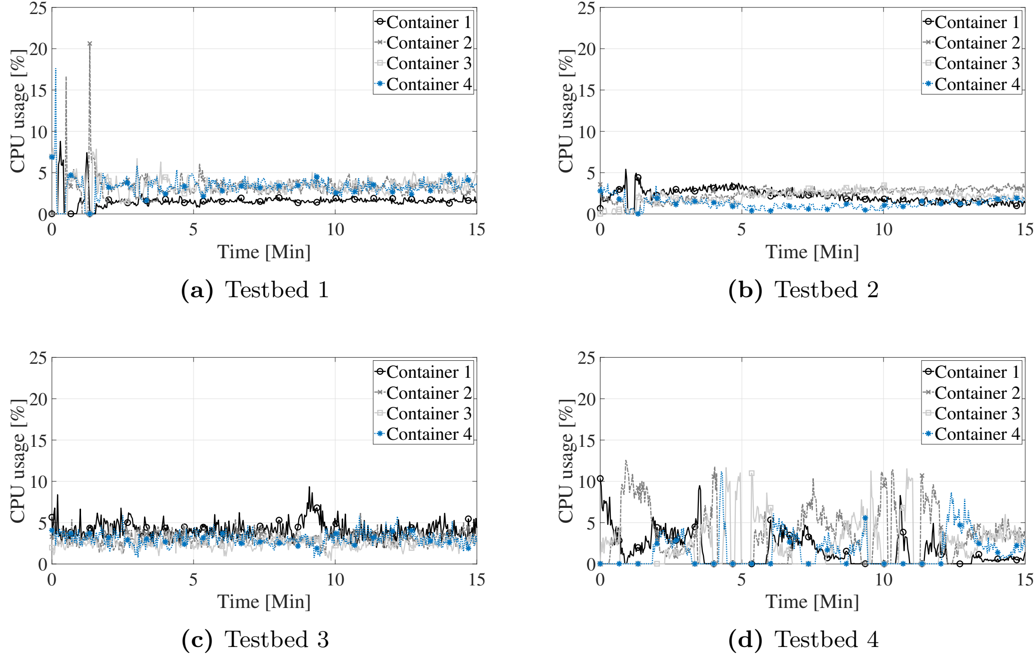


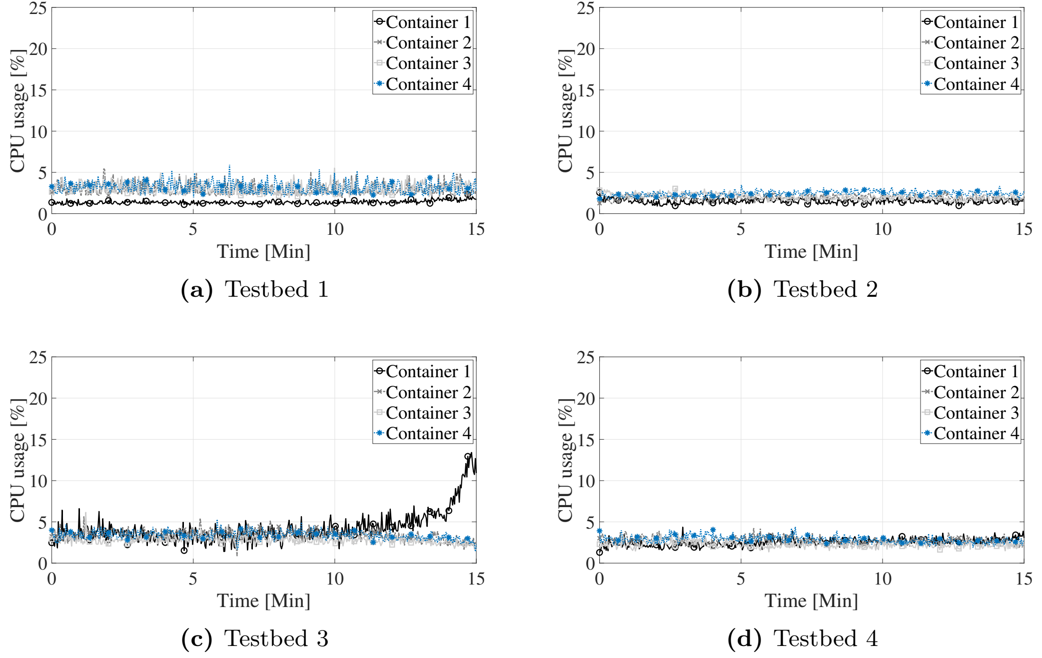
Figure 2.5: CPU utilization measured when $\lambda = 5$ requests/minutes.

5 and 10. Finally, while each test lasts 20 minutes, the results are extracted from an intermediate observation interval of 15 minutes.

2.4.1.1 CPU utilization

Figures 2.5 and 2.6 show the CPU usage of different containers deployed within the four investigated testbeds when the average number of client requests is set to 5 and 10 per minute, respectively. In all the cases, it emerges that the CPU usage registered by the available containers is almost similar during the time. This demonstrates the ability of all the selected technologies, and in particular of the load balancing functionalities implemented by the orchestrator, to offer a fair distribution of tasks within the whole virtualized service infrastructure, independently from the traffic load.

However, to better investigate the different behavior of developed testbeds, the cumulative distribution function of the CPU utilization values measured for all the containers belonging to a given testbed is reported in Figure 2.7. From the statistical perspective, it emerges that the integration of the Kubernetes orchestrator within



(a) Testbed 1 **(b)** Testbed 2

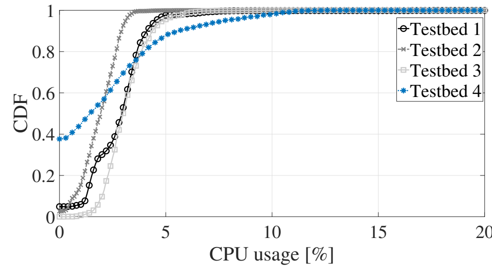
(c) Testbed 3 **(d)** Testbed 4

Figure 2.6: CPU utilization measured when $\lambda = 10$ requests/minutes.

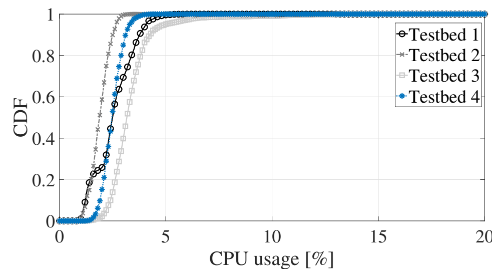
the bare-metal deployment platform registers the lowest CPU utilization. On the contrary, the virtualized service infrastructure exploiting the Docker Swarm orchestrator and using the OpenStack deployment platform achieves the worst performance. In the OpenStack cloud, several components (previously mentioned in Testbed3) are involved in providing the virtual computing, storage, and networking resources to the instances running into the considered deployment platform. This additional computational overhead justifies the higher CPU usage registered by the testbed leveraging the OpenStack cloud deployment platform.

2.4.1.2 Memory footprint

Figures 2.8 and 2.9 show the amount of memory occupied by the different containers deployed within the four investigated testbeds when the average number of client requests is set to 5 and 10 requests per minute, respectively. Apart from the different memory footprint experienced by each testbed, it is very important to highlight these two considerations. First, when the bare-metal deployment platform is used, the virtual machine hosting orchestrator and container (that is the first one in the presented imple-



(a) λ=5 requests/minute

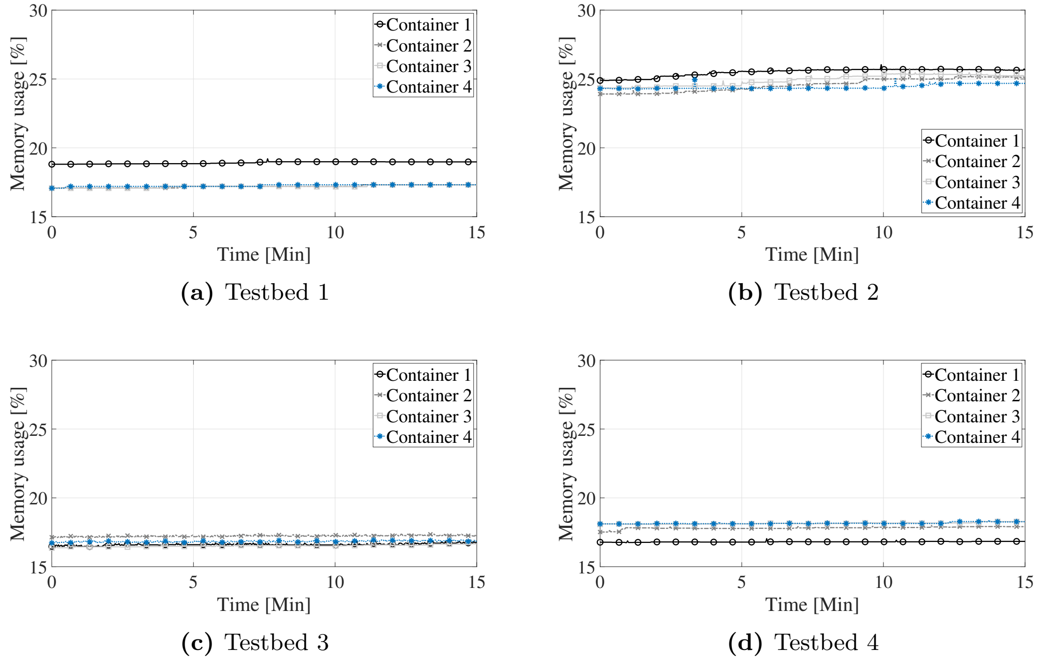


(b) λ=10 requests/minute

Figure 2.7: Cumulative distribution function of CPU utilization measurements.

mentations) experience a higher memory usage. Second, the memory footprint slightly grows during the first half of the experiment, when the number of parallel requests is increasing. This behavior is more evident when the average number of requests per minute is set to 10. Here, in fact, the increment of the traffic load brings to higher memory consumption.

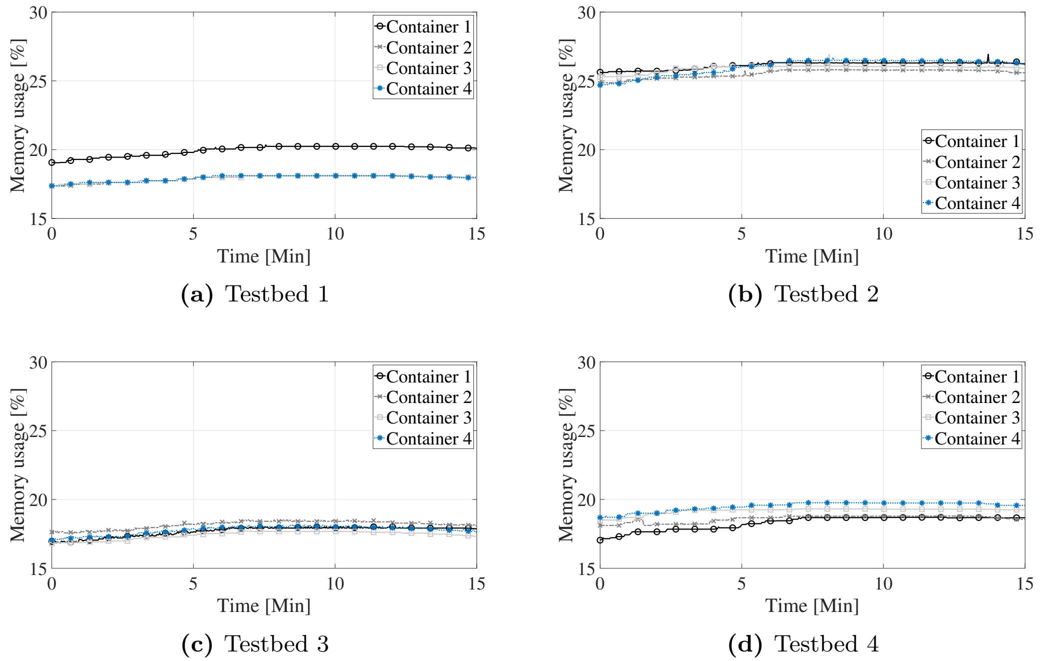
The cumulative distribution functions of the memory footprint values measured for each testbed are depicted in Figure 2.10. Results demonstrate that the lowest memory footprint is registered by the integration of Docker Swarm orchestrator into the OpenStack deployment platform. On the contrary, the adoption of Kubernetes within the bare-metal deployment platform consumes the highest amount of memory. These results reverse the considerations discussed for the CPU utilization: the testbed registering the highest CPU utilization ensures the lowest memory footprint, whereas the testbed registering the lowest CPU utilization achieves the highest memory footprint. Indeed, the containers deployed with Kubernetes on the bare-metal platform perform their tasks by utilizing more memory and less CPU than the technologies available on other testbeds.



(a) Testbed 1 (b) Testbed 2

(c) Testbed 3 (d) Testbed 4

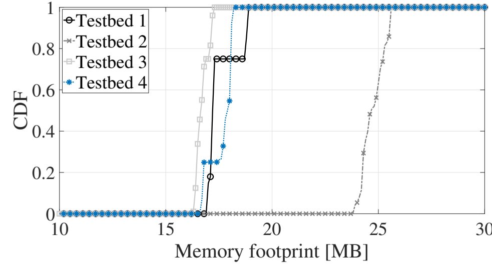
Figure 2.8: Memory footprint measured when $\lambda = 5$ requests/minutes.



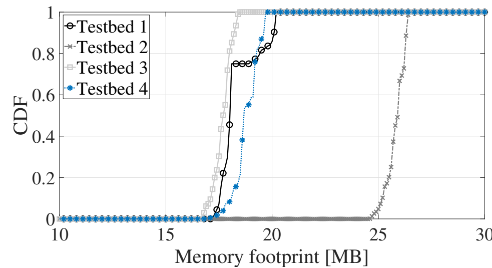
(a) Testbed 1 (b) Testbed 2

(c) Testbed 3 (d) Testbed 4

Figure 2.9: Memory footprint measured when $\lambda = 10$ requests/minutes.



(a) λ=5 requests/minute



(b) λ=10 requests/minute

Figure 2.10: Cumulative distribution function of memory footprint measurements.

2.4.1.3 Network load

The average amount of traffic generated by each container in a unit of time, when the average number of client requests is set to 5 and 10 requests per minute, is reported in Figures 2.11 and 2.12, respectively. In the case of $\lambda = 5$ requests per minute, the deployments with Docker Swarm produces consistently low throughput. On the contrary, the deployment of Kubernetes as the orchestrator achieves high throughput, thus ensuring that the requests are generally completed in a lower amount of time.

The cumulative distribution functions of network load measurements are reported in Figure 2.13. The results show that high network throughput is achieved by the integration of Kubernetes on the bare-metal deployment platform. On the opposite side, the deployment of Docker Swarm on the bare-metal platform produced lower throughput. This behavior is more clearer when the average number of requests per minute is set to 10. As anticipated before, OpenStack builds on several components that provide the virtualized network resources to the instances inside the OpenStack cloud. It can be the reason behind the average network throughput in the case of deployments on OpenStack.

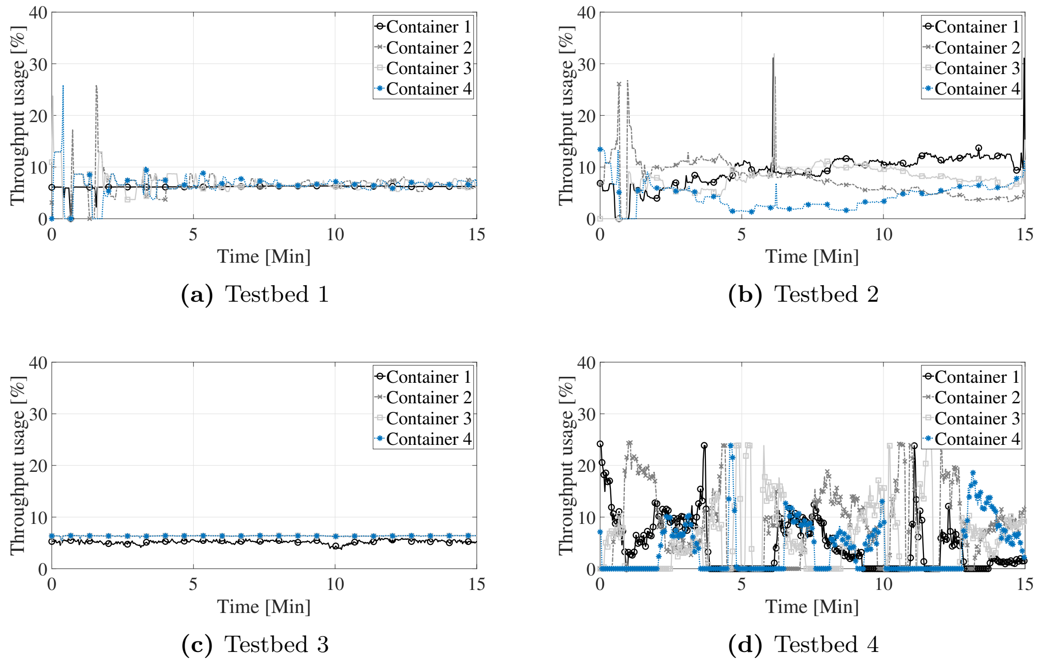


Figure 2.11: Network load measured when $\lambda = 5$ requests/minutes.

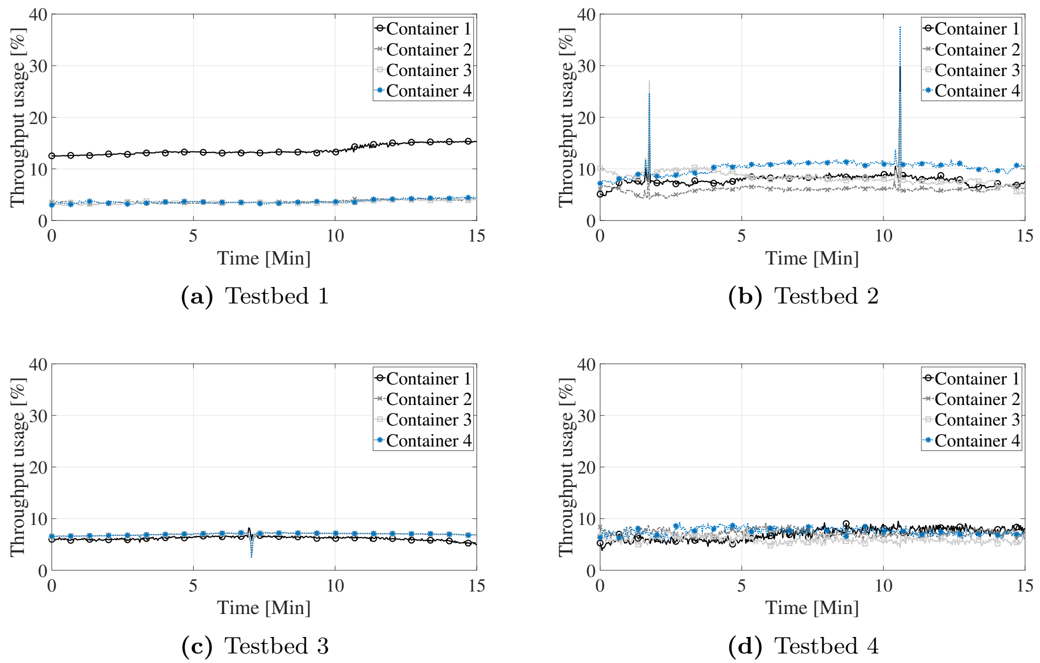
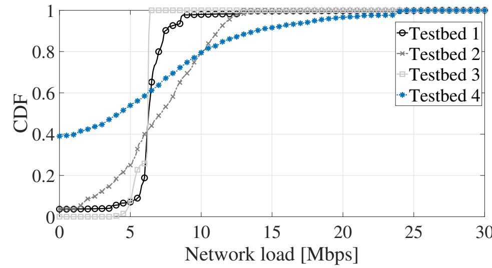
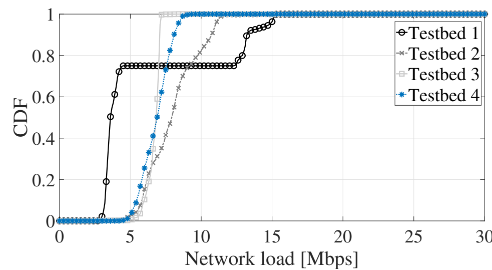


Figure 2.12: Network load measured when $\lambda = 10$ requests/minutes.



(a) λ=5 requests/minute



(b) λ=10 requests/minute

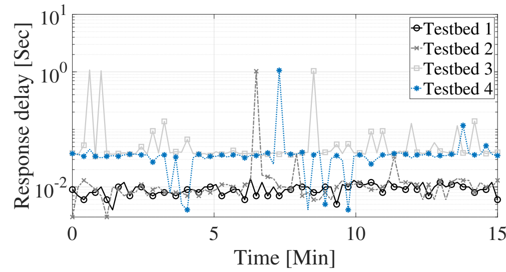
Figure 2.13: Cumulative distribution function of network load measurements.

2.4.1.4 Connection delay and request completion time

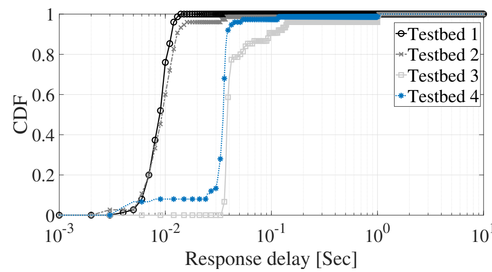
To conclude the cross-comparison, the KPIs introduced to evaluate the impact of developed testbeds on the quality of experience registered by remote clients are discussed below.

As the first set of results, Figures 2.14 and 2.15 show the connection delay measured when the average number of client requests is set to 5 and 10 requests per minute, respectively. As expected, different requests experience different connection delays, ranging from hundreds of milliseconds to a few seconds. Connection delays tend to increase with the traffic load, because of the increment of both traffic and tasks the virtualized service infrastructure handles. However, what clearly emerges from the reported curves is that the adoption of OpenStack as a deployment platform always provides higher connection delays. The delayed responses from OpenStack cloud is mainly due to the presence of NAT, which introduces an additional communication latency to the message exchange between clients and the virtualized service infrastructure.

More specifically, the integration of Docker Swarm within the OpenStack deployment platform registers higher connection delays. Whereas, clients connected to a



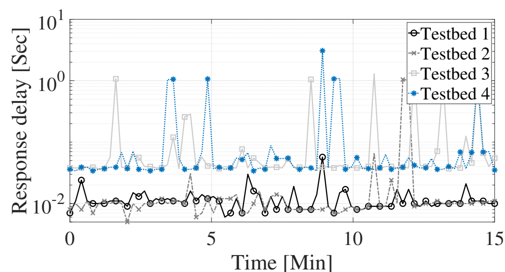
(a) Measured values during the time



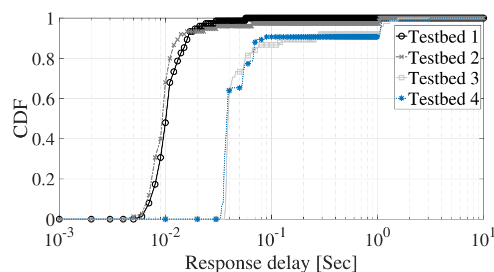
(b) Cumulative distribution function

Figure 2.14: Connection delays measured when $\lambda = 5$ requests/minutes.

virtualized service infrastructure embracing the Kubernetes orchestrator and adopting the bare-metal deployment platform generally registers lower connection delays.



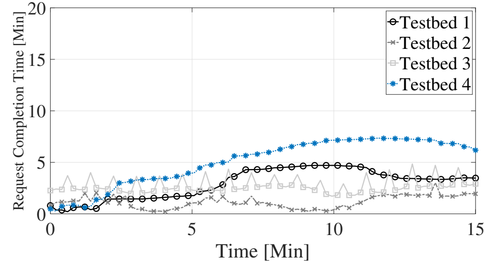
(a) Measured values during the time



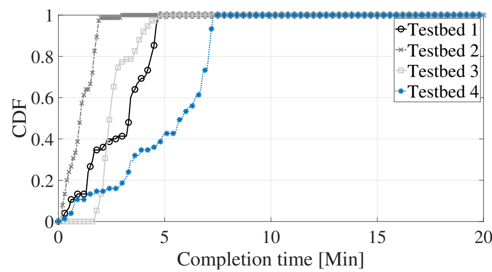
(b) Cumulative distribution function

Figure 2.15: Connection delays measured when $\lambda = 10$ requests/minutes.

The request completion time measured when the average number of client requests is set to 5 and 10 requests per minute are reported in Figures 2.16 and 2.17, respectively. In line with previous results, the request completion time tends to increase in the first half of the experimental tests, because of the increment of the number of concurrent requests. The higher the number of active requests, in fact, the higher the number of tasks executed by the components belonging to the virtualized service infrastructure. That, in turn, provokes the introduction of additional latencies. Here, the impact of the traffic load is tremendous: on the average, the request completion time registers an increment of about 10 minutes. In any case, however, the results demonstrate that the deployment with Kubernetes and bare-metal achieves better results. On the contrary, Docker Swarm within the OpenStack deployment platform registers the worst behavior.

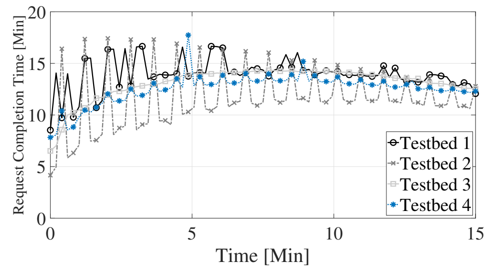


(a) Measured values during the time

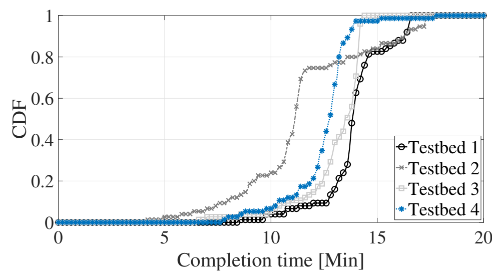


(b) Cumulative distribution function

Figure 2.16: Request completion time measured when $\lambda = 5$ requests/minutes.



(a) Measured values during the time



(b) Cumulative distribution function

Figure 2.17: Request completion time measured when $\lambda = 10$ requests/minutes.

2.4.1.5 Final considerations emerging from the analysis of the high-load environment

The results obtained from the first campaign of experimental tests firstly demonstrate that the integration of cutting-edge technologies for container networking is already feasible for hardware platforms whose computing capabilities are much more limited with respect to those available in the cloud. This would encourage SMEs to use these technologies in their business, services, and applications.

At the same time, the integration of Docker engine and Kubernetes orchestrator within the bare-metal deployment platform emerges as the most suitable solution because it ensures better performance in terms of CPU usage of containers, distribution of the network load during the time and among the deployed containers, connection delays, and request completion time, while registering a slight (but still acceptable) increment of the memory footprint. On the contrary, the integration of the Docker engine and Docker Swarm orchestrator within the OpenStack deployment platform, instead, generally achieves lower performance levels.

As a final consideration, it is important to note that computing tasks significantly influences the amount of energy consumed by involved servers. In fact, the higher is the weight of computing tasks, the higher is the amount of consumed energy. Thus, the evaluation of the CPU consumption gives an idea of the amount of energy consumed by containers integrated within the developed virtualized service infrastructure. In this sense, the conducted study demonstrates that all the reviewed service orchestrators can uniformly distribute users' requests among available containers, thus guaranteeing a very good balancing of the energy consumption among the key component of the system involved in computing tasks.

2.4.2 Performance assessment in a smart farm scenario

The goal of this Section is to investigate the performance of the selected combination of technologies in a more complex scenario, referring to a smart farm use case. The resulting architecture is depicted in Figure 2.18. It embraces many drones flying in a smart farm, able to take pictures with their on-board camera and deliver them to the remote virtualized service infrastructure for monitoring purposes. The drones are emulated through two laptops, connected to the virtualized service infrastructure by

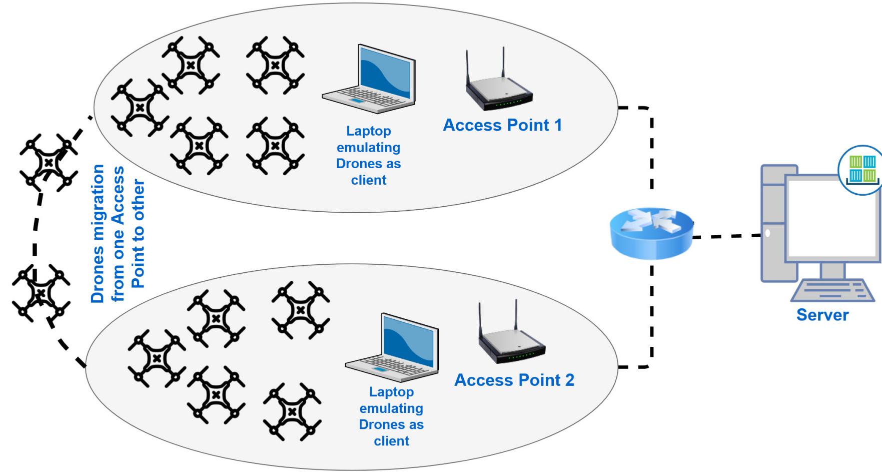


Figure 2.18: The virtualized service infrastructure evaluated in the smart farm use case.

means of two different wireless access points. Note that the number of drones emulated in each laptop changes over the time and the reference scripts are properly configured to emulate the movement of drones among the two available network attachment points. This way, the proposed campaign of experimental tests is also able to evaluate the ability of the whole system to properly work also in the presence of mobile users. During the tests, each drone randomly selects a livestock image from a local storage according to the Poisson statistic. Indeed, the number of messages delivered by each drone to the remote virtualized service infrastructure in a unit of time, λ, is set to $\lambda = 10$ requests/minute and $\lambda = 20$ requests/minute. The service orchestrator forwards the received pictures to available containers, which implements an object recognition application developed by using python and TensorFlow trained models. Here, the application identifies the type and the number of animals recognized in the image and share these outcomes with another server for monitoring purposes.

In line with the previous campaign of experimental tests, also in this case, the study evaluates the impact that the execution of heavy computing tasks, generated in different network load conditions, to the usage of computing, memory, and communication capabilities of the virtualized service infrastructure. This time, however, tests last 45 minutes.

2.4.2.1 CPU utilization

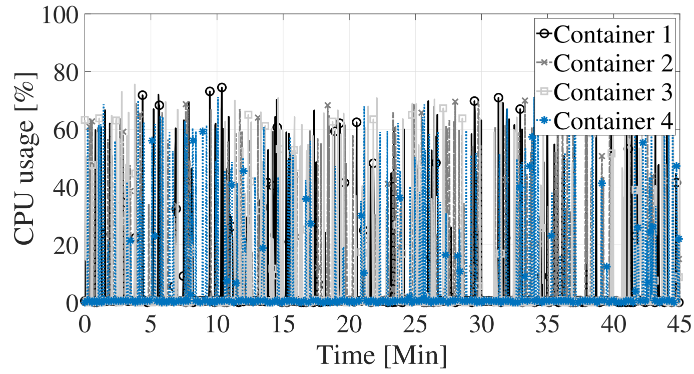
Figure 2.19 shows the CPU usage registered by the four containers deployed in the considered virtualized service infrastructure. Independently from the average number of requests generated by every single drone, it is possible to observe that the usage of the CPU follows a bursty behavior: computing tasks are highly used just during the elaboration of pictures. Contrarily from the previous campaign of tests, this time higher CPU usage is due to the complex image processing task performed by the containers. In any case, however, results fully confirm the ability of the orchestrator to uniformly distribute incoming requests among available containers. As expected, the cumulative distribution functions reported in Figure 2.20 highlight that the higher the number of requests generated by each drone in a unit of time, the higher is the registered CPU usage. But, in all the considered configurations, containers never glut their computing resources. This important result demonstrates, once again, how the development of a virtualized service infrastructure is feasible also in local computing environments.

2.4.2.2 Memory Footprint

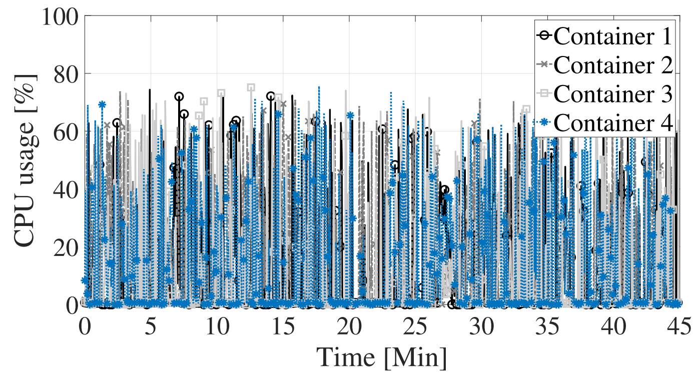
The image processing tasks implemented by containers require a higher amount of memory, as reported in Figure 2.21. The single container occupies 1.75 GB just for storing the machine learning-based application for image processing. This is the reason, during the tests, about 2 GB of RAM is consumed by each operating container. Also for the memory footprint, the whole system ensures a good balancing among the four available containers. Moreover, since the machine learning-based application always remains loaded on the RAM, the curves reported in Figure 2.21 do not present a bursty behavior. Figure 2.22 shows the cumulative distribution function of the memory footprint measured during the emulation of the smart farm use case. Here, it is possible to observe that the higher average number of requests per unit of time introduces a slight increment of the amount of consumed memory. Containers, in fact, are involved in a higher number of tasks, thus requiring more memory.

2.4.2.3 Network load

In the smart farm use case, the amount of data delivered by the network is just due to the transmission of pictures and provisioning of the resulting process. For this reason,



(a) λ=10/min



(b) λ=20/min

Figure 2.19: CPU usage measured during the emulation of the smart farm use case.

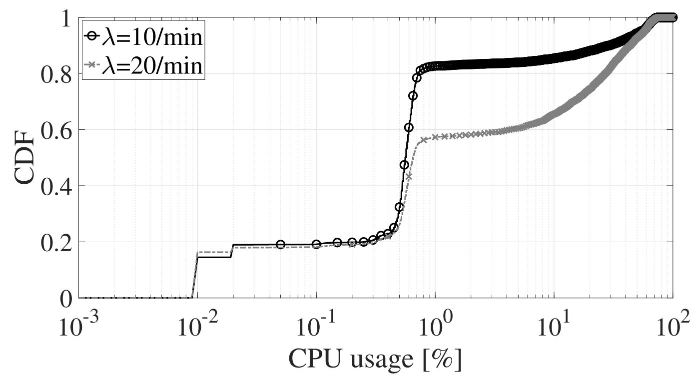
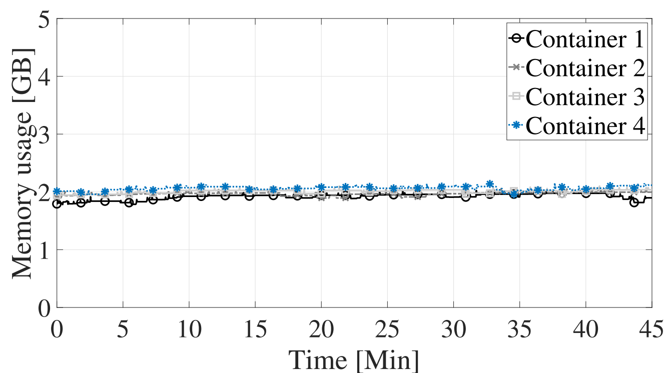
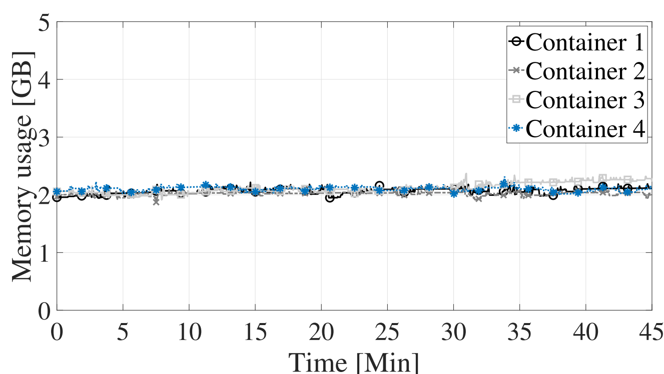


Figure 2.20: Cumulative distribution function of CPU usage measured during the emulation of the smart farm use case.



(a) λ=10/min



(b) λ=20/min

Figure 2.21: Memory footprint measured during the emulation of the smart farm use case.

the traffic load reported in Figure 2.23 presents a bursty behavior and does not achieve very high values. According to the cumulative distribution functions of the network load reported in Figure 2.24, it is possible to observe that the higher the number of messages generated by the drones, the higher the measured amount of traffic managed within the virtualized service infrastructure. From one hand, this result confirms what has already been presented before. From another hand, it shows how the smart farm use case requires less communication capabilities with respect to the scenario investigated in the initial campaign of experimental tests.

2.4.2.4 Connection delay and request completion time

To further investigate the performance of the investigated virtualized service infrastructure in the smart farm use case, the connection delay experienced by all the drones during the service provisioning is reported in Figure 2.25. Once again, results demonstrate that the virtualized service infrastructure promises a proactive response to clients. The cumulative distribution function of the connection delays measured in different traffic load conditions is reported in Figure 2.26. As expected, when the number of requests generated by drones increases, the average connection delay increases accordingly. But, in any case, it remains below 0.1 s with a probability higher than 97%.

On the other hand, the request completion time, that in this campaign of experimental tests represents the amount of time required to complete the complex image processing task, ranges from a few second to tens of seconds, depending on the amount of task load addressed by the virtualized service infrastructure (see Figure 2.27). Inevitably, and in line with all the afore discussed comments, the higher the task load, the higher the task completion time. The cumulative distribution functions reported in Figure 2.28, however, highlight that the identification of the type and the number of animals from the picture provided by flying drones is less than 20 s with a very high probability, even in a very complex scenario. This supports a final consideration: the limited computing resources adopted in the experimental tests are enough to implement complex services with acceptable levels of quality of service.

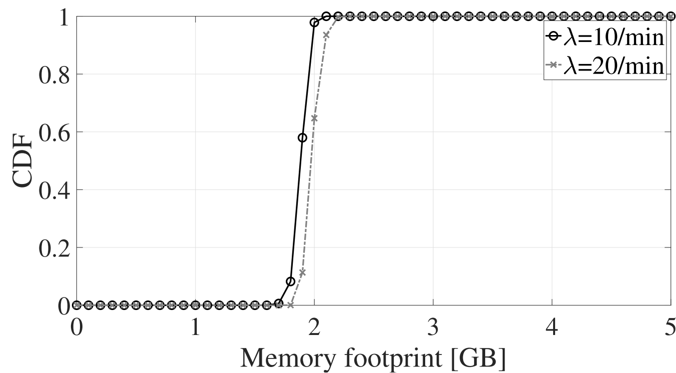
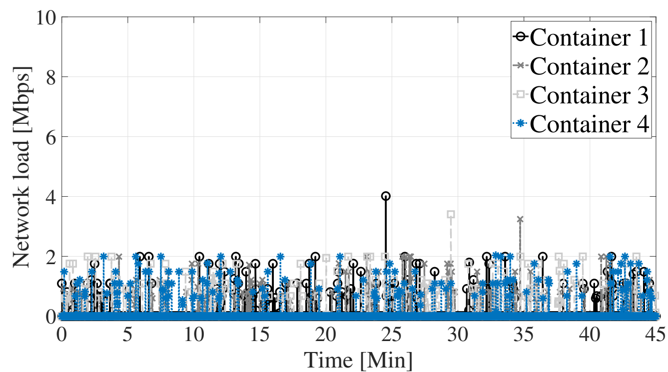
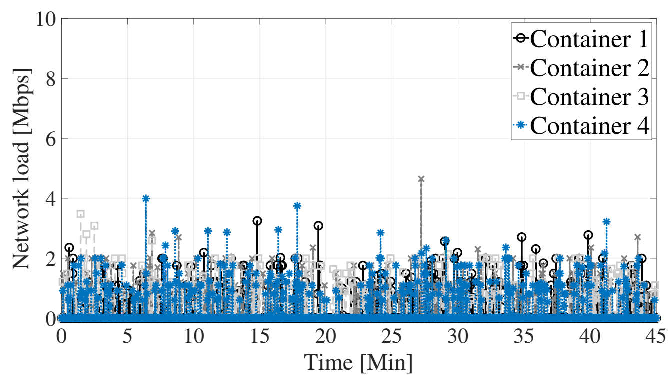


Figure 2.22: Cumulative distribution function of the memory footprint measured during the emulation of the smart farm use case.



(a) λ=10/min



(b) λ=20/min

Figure 2.23: Network load measured during the emulation of the smart farm use case.

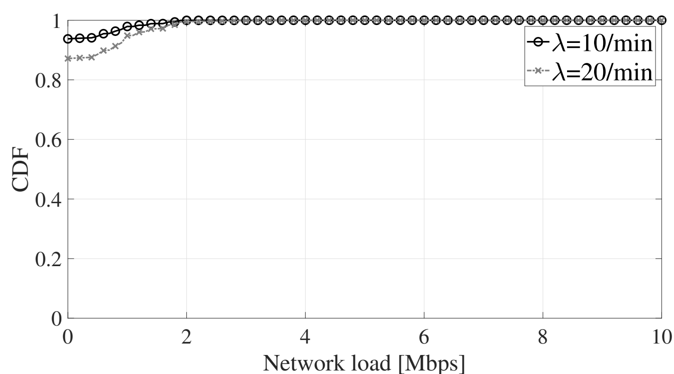


Figure 2.24: Cumulative distribution function of the network load measured during the emulation of the smart farm use case.

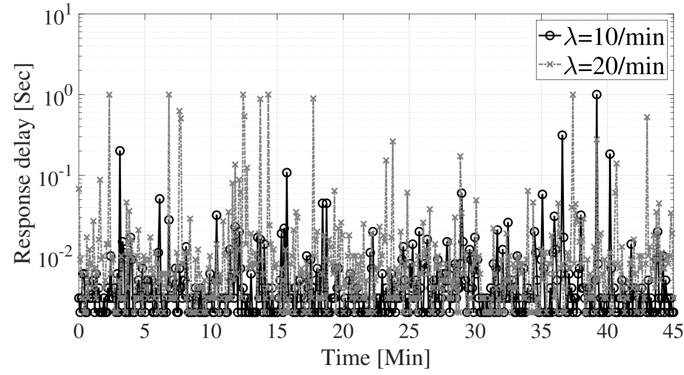


Figure 2.25: Connection delay measured during the emulation of the smart farm use case.

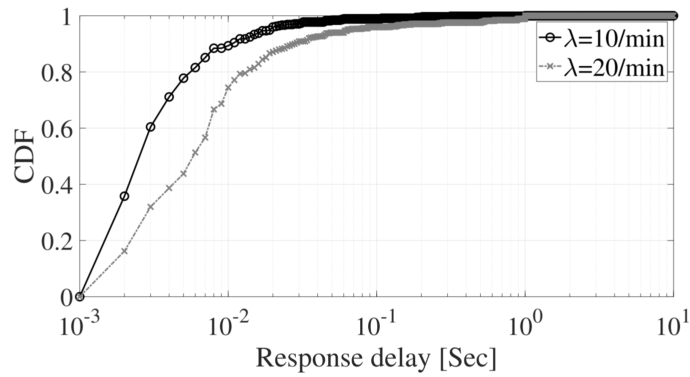


Figure 2.26: Cumulative distribution function of the conneciton delay measured during the emulation of the smart farm use case.

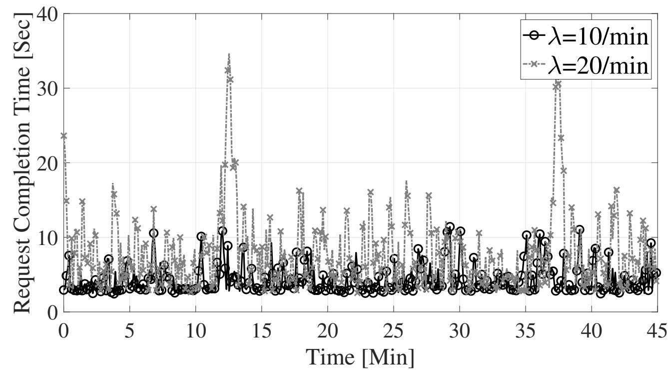


Figure 2.27: Request completion time measured during the emulation of the smart farm use case.

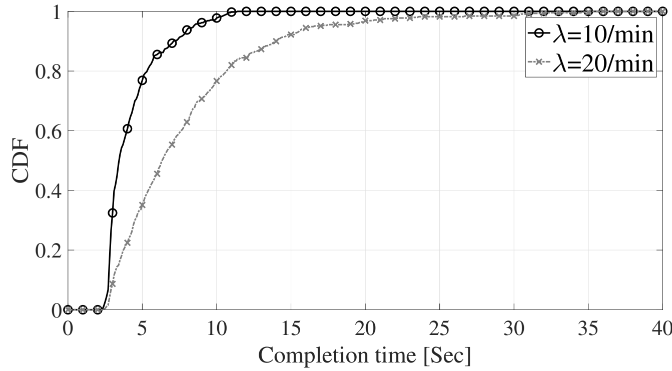


Figure 2.28: Cumulative distribution function of the request completion measured during the emulation of the smart farm use case.

2.4.2.5 Final considerations emerging from the analysis of the smart farm scenario

The results obtained from the second campaign of experimental tests can be finally used to formulated these additional comments. First, the behavior of the virtualized service infrastructure is not drastically influenced by the users' mobility: the service orchestrator is able to properly forward the requests to the available containers, independently from the origin of the requests. Based on the previous considerations, it is possible to confirm that the combination of Docker (i.e., the container engine) and Kubernetes (i.e., the service orchestrator with load-balancing functionalities) ensures better performance on the bare-metal deployment platform also in a more complex scenario with mobile users. Differently from the previous scenario, the smart farm use case, characterized by the execution of heavy tasks, requires higher computing and memory capabilities, as well as less communication bandwidth. Specifically, the higher CPU usage also translates into higher energy consumption (as already discussed in the previous section). But, despite what was explicitly declared from the performance assessment, the overall analysis clearly demonstrates that deployed infrastructure represents a suitable solution for effectively exploiting container networking capabilities in real deployments with local (i.e., limited) computing capabilities.

Of course, the study presented in this work can be generalized for implementing many other real-life scenarios, such as drones' communication in a mission-critical scenario [92], as well as for smart mobility applications in smart cities (i.e., where con-

tainers can effectively perform on the server-side to ensure quick computations of data with high efficiency and very minimal delay) [93].

2.5 Summary

This chapter presented a quantitative cross-comparison of cutting-edge technologies for container networking, properly integrated to realize a virtualized service infrastructure in local computing environments. The goal was to analyze the technologies to be deployed in the T-SDN based frame for hierarchical control plane management within this work (discussed in detail in Chapter 3). In particular, the set of investigated technologies are Docker as container engine, Docker Swarm and Kubernetes as orchestrators with load balancing and service discovery capabilities, bare-metal and OpenStack cloud as deployment platforms, and Docker-compose, Docker-Machine, Kubeadm, and Flannel as supporting tools for scheduling and deployment functionalities. First, the behavior of the four developed testbeds has been investigated through experimental tests in a high-load scenario, to evaluate the ability of the virtualized service infrastructure to provide answers to multiple requests received from the remote real-world network, as well as of reporting pros and cons of the considered technologies. The conducted study revealed that the integration of Docker engine and Kubernetes orchestrator within the bare-metal deployment platform ensures better performance. Then, the performance of the most suitable technologies was evaluated in a smart farm use case, integrating mobile drones and complex image processing tasks. The outcome of the evaluation demonstrated that the behavior of the virtualized service infrastructure is not drastically influenced by the users' mobility, the execution of heavy tasks generally requires higher computing and memory capabilities, while still guaranteeing acceptable levels of quality of service in real deployments with local (i.e., limited) computing capabilities. Indeed, the results of the cross comparison presented in this chapter would facilitate the Small and Medium Enterprises in the selection of technologies for container networking and encourage their adoption for revising business, services, and applications.

3

Design and Development of Optical Network Orchestration Framework

This chapter presents the design of an innovative hierarchical control plane and simulation framework for the T-SDN paradigm by selecting the best technologies (highlighted in chapters 1) and 2 and use them to test services and applications in a real environment. The high-level architecture of the framework has been described and a real-time complex simulation environment has been developed within the OpenStack cloud consisting functionalities of optical node simulation agents, two-levels of SDN controllers (one for managing and simulating the advanced optical nodes and other for integrating multi-vendor and third party environments), and communication protocols. A demo of creation of the simulation nodes is also explained. Additionally, it highlights the Power Management and monitoring perspective for the energy optimization of the framework, implementation at the ONOS southbound for retrieving the actual power consumption data from the optical node simulators, and the connectivity service perspectives to instruct the standby/wakeup statuses, SNMP Management of level-1 SDN controller at the ONOS, and connectivity of the ONOS with the orchestrator.

3.1 Motivation and Overview

SDN is a cutting edge technology for the deployment of programmable and virtualized service infrastructures. On the other hand, NFV emerged as a new network architecture approach to virtually deploy network functions on generic hardware[94]. The state-of-the-art demonstrates that the combination of SDN and NFV enables unprecedented levels of network control, dynamicity, and flexibility [95, 96, 97]. Telco operators are encouraged to take this opportunity by integrating SDN and NFV into their large scale geographical networks, eventually known as T-SDN. However, this integration is not straight forward and poses numerous challenges due to the large scale complexity of the telecommunication networks and the selection of suitable technology from the available open-source projects backed by different standardization bodies.

This work, as the part of INTENTO (INTElligent NeTwork Orchestration Framework) project, recently funded by the Apulia Region (Italy), addresses the aforementioned issues as part of the INTENTO project, we started off by developing an innovative simulation framework by selecting the appropriate state of the art technologies and integrate them to test applications, services, and advanced optimization algorithms in the real-time and complex T-SDN environment. A T-SDN architecture has been designed, that incorporates distributed and hierarchical monitoring and deployment of large scale optical switches and network functionalities (i.e., VNFs) by means of a two-level structure of SDN controllers. The level-1 SDN controller manages the optical nodes, whereas the role of the level-2 controller is to allow the integration with third party and multi-vendor environments. The VNFs are optimally deployed via a central orchestrator based on the network and user requirements.

Based on the proposed architecture, a real-time and complex simulation environment has been built within the OpenStack cloud, consisting of the following functionalities: (1) Optical node simulators consisting simulation agents with the emulated hardware layer, (2) the level-1 proprietary SDN controller developed as the part of this project to manage the advanced optical nodes features, and (3) Open Network Operating System (ONOS) has been adopted as the level-2 SDN controller in order to facilitate integration for third party and multi-vendor environments on standardized communication protocols like Transport API (T-API), NETCONF, and RESTCONF,

for both southbound and northbound interfaces. It is worth noting that in this architecture real equipment can be integrated into the simulation environment. Automated deployment procedure has been developed to effectively deploy the complete simulation environment.

3.2 The Proposed Framework

The goal is to implement a Telco-Cloud orchestration platform using open source software modules and standardized interfaces. The telecommunication infrastructure includes all the relevant hardware and software components of a T-SDN architecture, ranging from optical nodes through a two-level network management system, to the overall infrastructure management based on a centralized orchestrator. On top of that, VNFs are optimally deployed and managed by a centralized orchestrator in order to implement and test innovative services and applications.

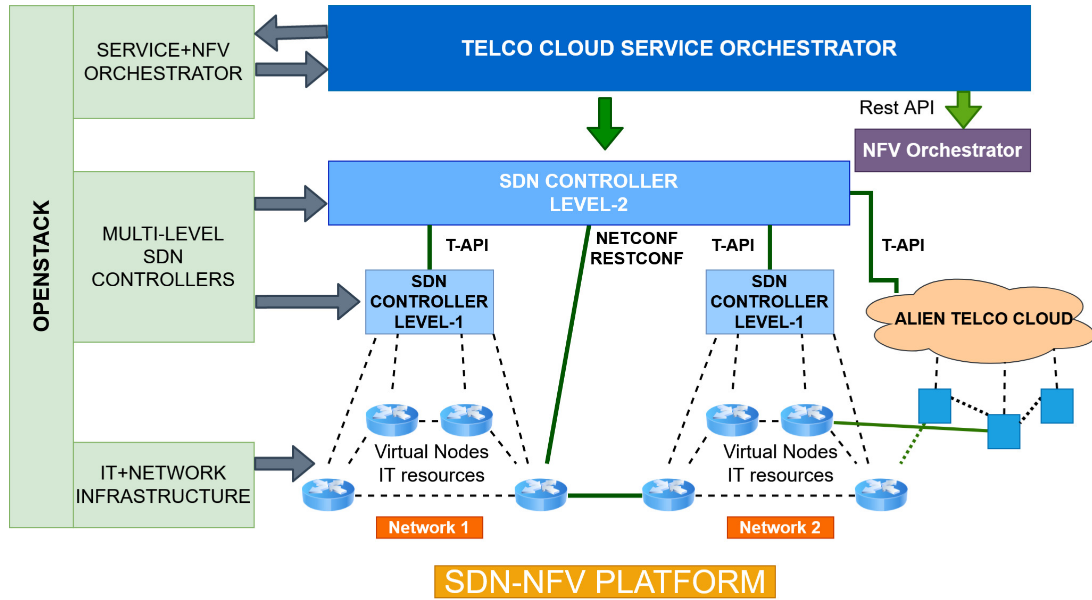


Figure 3.1: The conceived high-level framework.

Figure 1 describes the reference architecture of the simulation framework, highlighting the driving factors: 1) development of the node models according the Yet Another Next Generation (YANG) standard, 2) support of T-API interface and NETCONF/RESTCONF communication protocols to ensure the compatibility with existing

network standards, 3) support of L1-L3 network layers and related multi-layer management, and 4) implement a NFV infrastructure management enabling the development and testing of VNFs.

The extensive use of a virtualized environment, allow the mix of real nodes and simulators and can be used to reach a very high node count, to test very complex networks.

The Project conceived framework's objectives are supported by an IT infrastructure based on standard servers. More precisely, the aim is to demonstrate the overall framework capability of simulating a complex infrastructure management, including optical layer design and planning, multilayer (DWDM / OTN / Packet) management. The simulation framework can be used to carry out complex simulation scenarios, very useful to select the best solutions among alternative option and assess the overall performance. In addition, thanks to the open framework architecture, advanced applications will be selected and tested, i.e., the effectiveness of a set of VNFs, which may be hosted in the optical nodes.

3.2.1 Targeted use cases

Although embedding multi-level service orchestration architectures in nodes of a telecommunication network is often seen as a way for serving traditional Telco applications as VNFs, this vision leaves out interesting target areas that are non-purely Telco. Indeed, several ICT applications exist that either demand or greatly benefit, from the availability of edge-based processing coupled with synchronous inter-node communication, in terms of low latency, availability, survivability, as well as scalability, especially when all is cleanly modeled as VNFs and orchestrated as such. For example: Content Delivery Network (CDN) for efficient Video distribution, Blockchain processing VNF, Camera processing VNF for social distancing and face mask-wearing rule infringements detection for anti-COVID-19 precaution, IoT data collection and first-level aggregation VNF for sensor arrays, vehicle traffic support systems, and smart grid applications etc.

- Content Delivery Network (CDN) for efficient Video distribution. CDN services have always been exploiting advantages of decentralized, multilevel architecture with the central repository (usually called Origin) holding entire catalog, and one

or more levels of more and more peripheric nodes (usually called Edge) holding progressively smaller caches of content elements having progressively higher probability to be requested based on some form of statistical analysis. However, such services have been usually implemented by placing dedicated physical appliances at sites accepting the presence of such devices; this usually rules out the installation in anything lower, in the network hierarchy, than Internet exchange points. True edge computing pushed to metro nodes, often back to back with Node2/DSLAM devices, by embedding the processing and storage capability (and its enabling architecture) directly in such equipment, has the potential to provide the best cache hit ratios, lowest possible RTT and therefore, an IP throughput that best approximates the nominal rate of the underlying access network.

- Blockchain processing VNF for supporting digital currencies. The computation of digital currencies is computationally expensive and involves communication between several nodes (often heavily loaded nodes) in order to reach the necessary conditions to approve a digital cash transaction. This frequently leads to long response times. An increase in the number of nodes participating in the blockchain processing activity, with a fast and reliable communication link between them and greater proximity to the requestor, can contribute to reduce node load and accelerate inter-node communication, resulting in faster digital currency transactions.

- Face and vehicle recognition VNF for surveillance services. Large-scale security and surveillance systems capable of identifying and following suspects based on face recognition techniques can work more effectively if the conversion between high-bandwidth image streams and some form of efficient person identification (such as SSN or other ID) can occur in a delocalized manner, as close as possible to the cameras, in the edge of the network. Then a higher level in the architecture would receive not a group of video streams to analyze, quickly proving impossible to cope with several cameras and scenes increases, but a much more manageable stream of person identifiers and location/timestamp information.

- Camera processing VNF for social distancing evaluation and face mask-wearing rule infringements detection for anti-COVID-19 precaution. A special subcase

of the previous scenario involves real-time geometric estimation of inter-personal distances in a scene, combined with visual processing to recognize people not properly wearing face masks. This information must be extracted from video streams using heavy processing that is very difficult to scale if performed at a central location; once distances and violation information is extracted from scene, subsequent processing, notification, alerts, etc. can be managed by a more classically centralized system.

- IoT data collection and first-level aggregation VNF for sensor arrays, vehicle traffic support systems, smart grid applications. Large numbers of sensors overall providing a steady stream of data to be pre-processed, correlated and compressed before being actually stored and/or analyzed, places a significant load on the access and aggregation networks, and important scalability challenges for first-stage processing, if all such activities have to take place at a centralized location such as a data center. Network edge has plenty of small but ubiquitous nodes, each containing sufficient local CPU/storage resources, to offload all such preliminary computation leaving only higher-order semantics to be implemented by algorithms executed at a central location.

- Low-latency V2G support VNF for Smart Road applications. Smart road applications are more and more likely to emerge in new urban scenarios with partially, or eventually, fully self-driving vehicles, which may rely on infrastructure-originated reports about specific dangers, hidden vehicles, people crossing the road in low visibility conditions, etc.; and vice versa, V2I communication can take place as a way for each vehicle to contribute directly discovered information to other vehicles even in case they are too distant for a direct, wireless V2V communication and will reach the scene only after the previous vehicle has left the position. The necessary latency reduction must encompass all phases of the V2I/V2V communication, and this does not merely include the 5G access network, but also the transport to/from the site where information is processed. Thus, edge-based processing can contribute to mitigating IT scaling issues as well as packet network delays.

3.2.2 High-level architecture

The architectural choices of of the conceived framework are based on an extensive evaluation of the basic technologies, taking into consideration not only the pure technical performances but also additional parameters, including sustainability, usability and Opex minimization.

Referring to Figure 3.1, the items composing the overall architecture are:

3.2.2.1 Telecom Nodes

The simulator of the optical nodes is based on the SM Optics technology and are the base of the Telecom infrastructure, providing the connectivity for each architectural component.

3.2.2.2 Specialized SDN Controller (level-1 Controller)

The level-1 Controller is in charge to manage the Telecom Nodes simulation instances and represent the actual NMS solution for SM Optics nodes.

3.2.2.3 Multi-vendor/Multi-domain SDN Controller (level-2 Controller)

The level-2 controller is supposed to act as a generic SDN controller, based on standard interface, enabling the simulation to deal with multi-domain, multi-vendor environment.

3.2.2.4 VNF Orchestrator

Provide the support for the whole lifecycle of VNF instances, from library management to the actual deployment, to the activation and monitoring functions.

3.2.2.5 Framework Orchestrator

It is based on Openstack and should be intended as the general orchestrator framework needed to exploit all possible services conceived for the proposed infrastructure.

3.3 The implemented testbed

This section focuses on the technical details related to the implemented simulation framework, while presenting: selected technologies, integrated components, and auto-

mated deployment. An example showing the current usage of the simulation framework is discussed as well.

3.3.1 Components of the simulation framework

The developed simulation framework consists of:

- Network Simulation Agent: The network simulation agents (also known as optical node simulator) are developed to model virtualized optical switches in the network comprising different characteristics i.e., bus speed, number of connecting ports, and type of connectors etc. Each virtual switch can be connected to one or multiple optical switches in the simulation environment.

- Level-1 SDN controller: A proprietary SDN controller has been designed and developed as part of the INTENTO project to enable the management and control of simulated optical nodes. The core responsibility of the mentioned controller is the creation and management of the virtualized optical switches on the network simulation agents connected to it. Moreover, it is also responsible for communication with the multi-vendor supportive SDN controller ONOS on level-2. In our proposed simulation environment level-2 SDN controller is connected with a bunch of level-1 SDN controllers associated with an enormous amount of Network Simulation Agents comprising several virtualized optical switches.

- Level-2 SDN controller: For the selection of level-2 SDN controller, despite, several open-source controllers available in the industry, the most prominent ones are ONOS and OpenDaylight. The aforementioned controllers allow communication with third-party controllers through the well-known communication protocols available in the industry (i.e., OpenFlow, NETCONF, and RESTCONF). The motivation behind the selection of ONOS in the INTENTO framework is its communication mechanism. ONOS provides support for T-API protocol at the South-bound interface over the REST protocol. In the contrast, OpenDaylight earlier provided support for T-API in their UniMgr project but in the recent releases of ODL, there is no support for T-API, which is the provision in the INTENTO project for communication between the level-1 and level-2 controllers.

- Modeling language: YANG is a data modeling language for the definition of data sent over network management protocols such as the NETCONF and RESTCONF. It is used in our project to model both configuration data as well as state data of elements in the network.

- Application deployment technology: Containers technology provides an effective way for application deployment. Docker has been selected as the container engine, based on a qualitative cross-comparison of technologies for containerization discussed in [74]. All the executables and the required dependencies for the deployment of the level-1 SDN controller and optical node simulator are packed inside the containers to make the application portable and easily deployable.

- Operating environment: The OpenStack cloud has been selected as the Operating environment for the simulation framework. It can virtualize and control large pools of computing, storage, and networking resources. OpenStack has been chosen because of its opensource licensing, wide adaptation in the industry, active community support, and frequent releases of new features as per industry demands [74].

3.3.2 Communication protocols and interaction

The network configuration and communication between the components of the simulation framework is carried-out through the following protocols:

- T-API: It is a transport protocol that delivers a flexible North-Bound Interface for integrating SDN controllers in the network by facilitating transport communication through REST API following T-API models, written in YANG.

- RESTCONF/NETCONF: The purpose of these protocols is the communication between multiple controllers and network simulation agents. They provide mechanisms to install, manipulate, and delete the configuration of network devices through remote procedure calls and XML/JSON based data encoding for the configuration data as well as the protocol messages.

3.3.3 The developed driver

In the proposed architecture, ONOS works at the Control tier. It is the land of control plane where intelligent logic in SDN controllers would reside to control network infrastructure. This is the area where every network vendor is working to come up with their own products for SDN controller and framework. Here in this tier, a lot of business logic is being written in controller to fetch and maintain different types of network information, state details, topology details, and statistics details etc. Since SDN controller is intended for managing networks, it must have control logic for real world network use-cases like switching, routing, L2 VPN, L3 VPN, firewall security rules, DNS, DHCP, and clustering. Several networking vendors and even open source communities are working on the implementation of these use-cases in their SDN controllers. Once they get implemented, these services expose their APIs (typically REST based) to the upper tier (Application tier), something which makes life easy for network administrators who then use apps on top of SDN controllers to configure, manage and monitor underlying network. Control tier lies in middle, and it exposes two types of interfaces i.e., Northbound and Southbound. The routing strategy (control logic) conceived within this work has been placed on the control tier within the proposed framework for the management of high scale network and IT infrastructure. The Driver developed as part of this work represents the conjunction point between ONOS and the level-1 controllers along with Network Simulation Agents to retrieve the network and power related data from level-1 controllers and Network Simulation Agents, respectively. Moreover, it also instructs the level-1 controllers in the connectivity service between multiple Network Simulation Agents i.e., creation and deletion of links to ensure quality of service and energy efficiency in the network.

3.3.3.1 UML

Figure 3.2 represents the Driver built at the level-2 by modifying the OpenSource code of ONOS to allow the system to interact with the level-1 controller.

The UML diagram in the Figure 3.2 needs to be read from top to bottom and following the direction of the arrows. As one can see, the classes contains instances. The UML schema in the image, the main package driver.intento contains the core of the driver in the tapi package. The impl package contains the only class TapiDriverLoader required

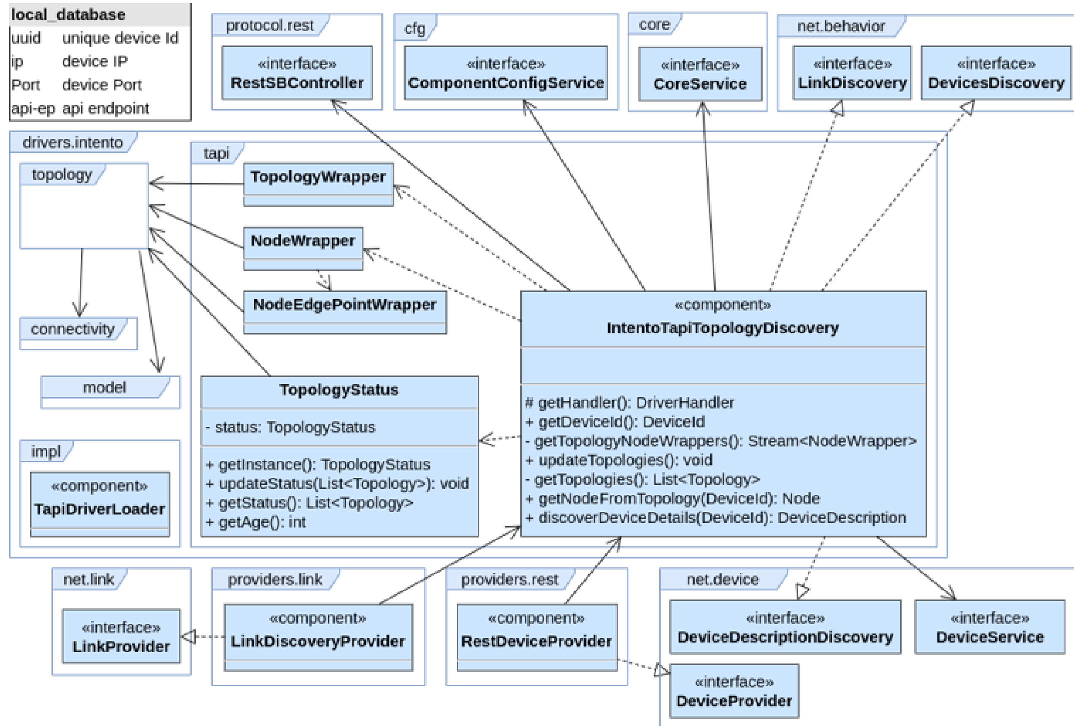


Figure 3.2: UML Class diagram of the developed ONOS Driver.

by ONOS to compile and publish the driver component. Basically, the driver is made of the component class IntentoTapiTopologyDiscovery that can be customized with the parameters registered in the system using the ComponentConfigService interface implements all the necessary behaviors to make the driver work correctly. The behaviors, namely DevicesDiscovery, LinksDiscovery, and DeviceDescriptionDiscovery are opportunely customized to match the communication with the smo-controllers. Once a new device gets connected to ONOS (the connection procedure will be shown later in this document), the RestDeviceProvider class (one of the providers in ONOS) searches for the correct driver to use with the connected device and calls the driver's methods using the Java reflection mechanism. Similarly, the LinkProvider class uses the driver to populate the archive of Links representation in the ONOS topology. The driver first retrieves the controller's topology using the RestSBController interface, then it uses the wrapper classes (TopologyWrapper and NodeWrapper) to build an instance of the controller's topology to be stored in the TopologyStatus singleton class and creates as many ONOS Device representations as the Nodes represented in the TAPI topology

received as a response from the controller. The TopologyStatus singleton class is used as an archive to store all the topologies received from the connected controllers and keep them available for the Orchestrator API App developed at ONOS' NorthBound (the details of this app will be described later in this document). The Nodes, instead, become Device instances for ONOS and store the main information needed to identify and retrieve them. These Device instances are stored in ONOS using the DeviceService interface. Periodically, according to the parameter set, published and updatable using the ComponentConfigService together with the CoreService interfaces, the request for topologies to the connected controllers gets repeated and the topologies get updated. Moreover, for each node in the topology, based on a local database (local database populated by reading the UUID-IP.json file archived by default in "\onos \drivers \intento \src \main\resources") it retrieves from each node (via RestConf API) the information specific of each network element, to be integrated in the Node model class. The wrapper classes used to build and retrieve the representation of the topology objects, use the models built and stored in the topology, connectivity, and model packages. The content of these packages is detailed in the Figure 3.3 and basically consists of a set of Serializable model classes that define the structure of each element and provide public methods to retrieve specific information.

3.3.4 Sequence Diagrams

The following sequence diagram describes with higher detail what happens in ONOS when a smo-controller's configuration is sent using the API mentioned in the previous paragraph.

Given that ONOS is running and the INTENTO Driver has been activated, using the POST API call to send a device's configuration, the net-cfg ONOS component generates a connection event that gets caught by the provider RestDeviceProvider (because the device has rest:* as header). Using a dedicated handler that acts as a factory, the component recognizes the correct driver to use and calls the method to retrieve the details of the main device (smo-controller). The IntentoTapiTopologyDiscovery component class collects the test of basic information (currently manually customized because not recognizable from the controller) and send them back as a response. Having recognized that the added device acts as a proxy, the provider asks for the identifiers of the proxied devices (i.e., the nodes in the TAPI topology). Using a series of weappers and

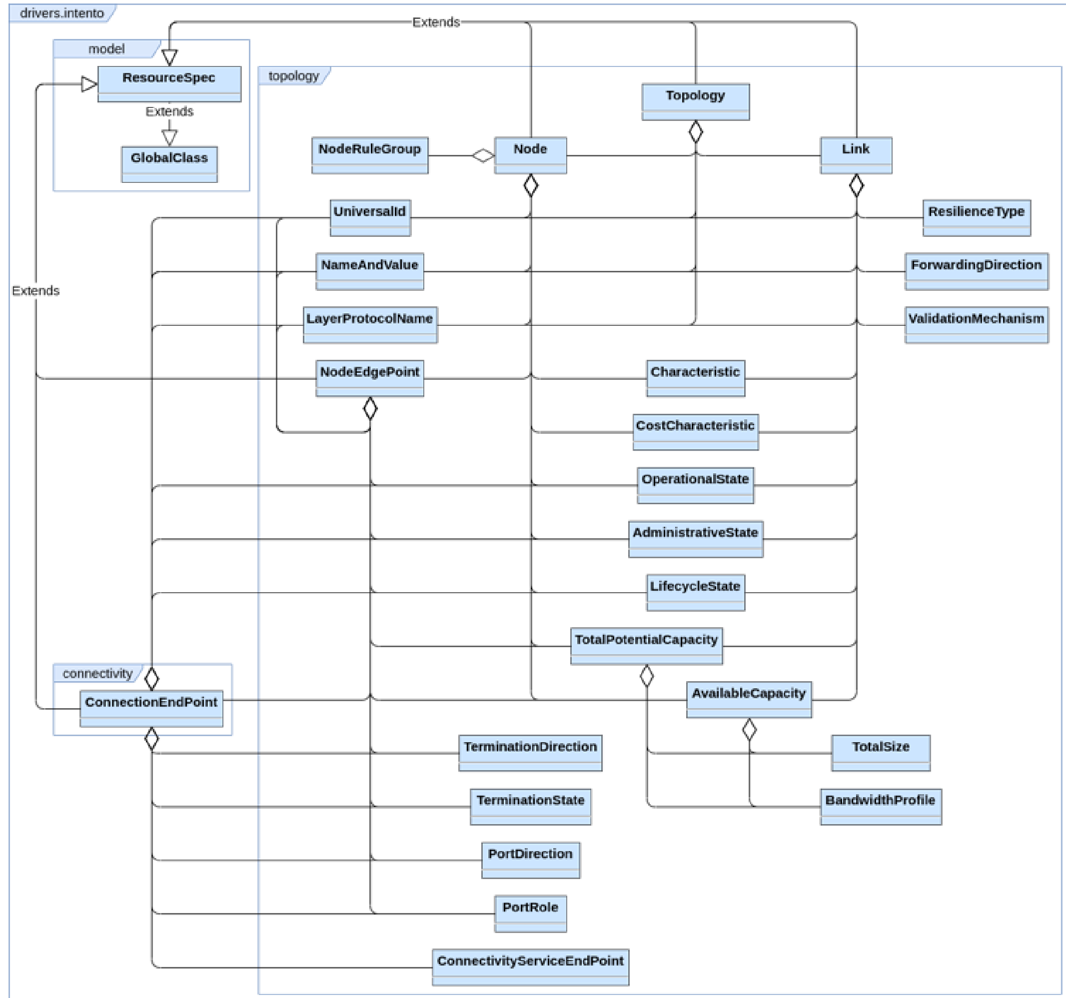


Figure 3.3: UML Class diagram of the INTENTO Driver's Model Objects.

internal methods, the driver component asks to the RestSBController, representing a boundary class in the ONOS southbound to retrieve the topology from the smo-controller. The Rest controller builds and calls the API and returns the TAPI context as a response. The context gets parsed by the wrapper classes and saved into the TopologyStatus singleton class that maintains it together with the other topologies and the last update time. The list of the node Ids is then extracted and returned back to the provider. The provider starts a loop to retrieve the description of every proxied device from the driver component, as well as the information on their ports. The provider takes care of setting the representation of each device, port, and link in the ONOS

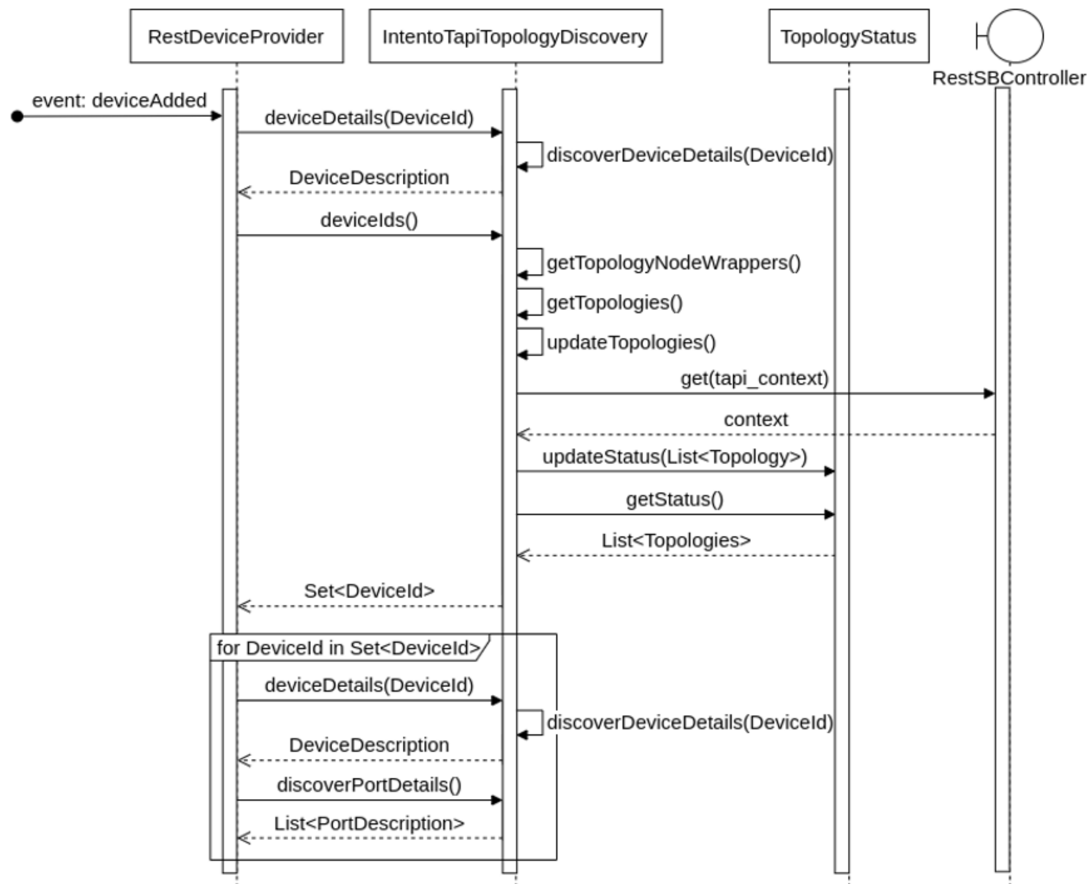


Figure 3.4: Sequence diagram of a controller connection.

system topology. Meanwhile, ONOS is running a number of scheduled threads that get repeated periodically. One of them is the LinkDiscoveryTask that gets executed by the LinksDiscoveryProvider. What happens next is described by the following sequence diagram and detailed below.

When triggered, the discoverLinksTask takes care of polling the information of every possibly existing link for every device. The LinksDiscoveryProvider starts looping among the devices and calls the discoveryLinks method from the correct driver, selected using the handler as well as what has been done by the RestDeviceProvider above. The driver component uses the wrappers to retrieve the set of existing links from the TAPI topology and returns them back to the provider. The provider takes care of setting them in the ONOS system topology.

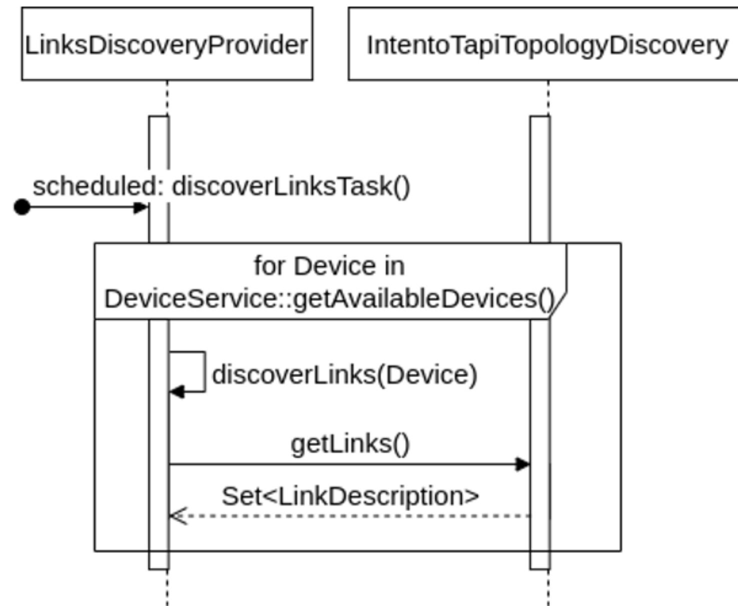


Figure 3.5: Sequence diagram of the links discovery process.

3.3.5 Orchestrator API

A new Orchestrator API has been built at the northbound of ONOS App by adding new code as part of this work project. This API works as a conjunction point between ONOS and the components at the upper levels of the Architecture i.e., the Orchestrator, that will manage the services based on the network status received from the control level (i.e., ONOS). It represents a web application made of REST API to access the available data and provide the upper component for an access point to give instructions and directives. The UML representation of the orchestration API is given in figure 3.6.

The Orchestrator API is basically a collection of API. The core class WebApplication exports in ONOS the API modeled by the set of "Resource" class like ContextResource that contain a dedicated method for each primitive (GET, POST, DELETE, etc.).

As shown in the Figure 3.7, representing the sequence diagram, once the API gets called, the correct method is called. The method represented in this case, extracts the topologies from the TopologyStatus class described in the section above, and wraps up data in a JSON Node that returns to the caller.

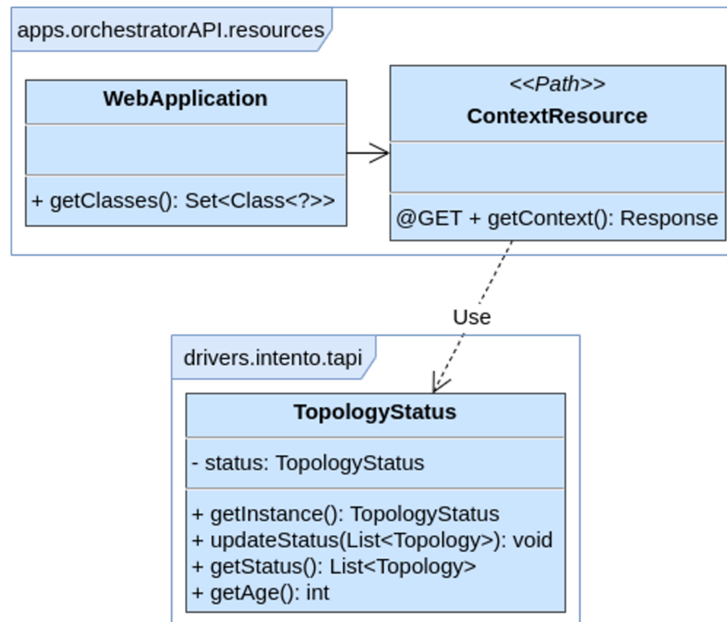


Figure 3.6: UML Class diagram of the OrchestratorAPI App.

3.3.6 Achieved implementation and the developed simulation framework

The current simulation framework being developed, integrates deployment of two-level of SDN controllers, within the OpenStack cloud. The proprietary SDN controller developed as part of the project, is deployed on level-1 and an Open-Source multi-vendor supportive ONOS SDN controller has been placed on level-2 in the framework. The optical node simulator, which is also developed as the part of this project is connected to the level-1 and level-2 controllers via the RESTCONF/NETCONF interfaces. As shown in Figure 3.9, the optical node simulator is dynamically controlled by the level-1 SDN controller. Each optical node simulator represent telecom nodes that can create a large number of virtual interfaces for communication with other telecom nodes. The level-1 SDN controller is connected with the level-2 SDN controller through the T-API interface using the REST API. The level-2 SDN controller can retrieve the information related to simulated nodes either through the level-1 SDN controller or can directly retrieve the power-related data from the optical nodes. The topology related information is retrieved through the level-1 SDN controller. The YANG language is used for the communication models between the level-1 and level-2 SDN controllers as well as the

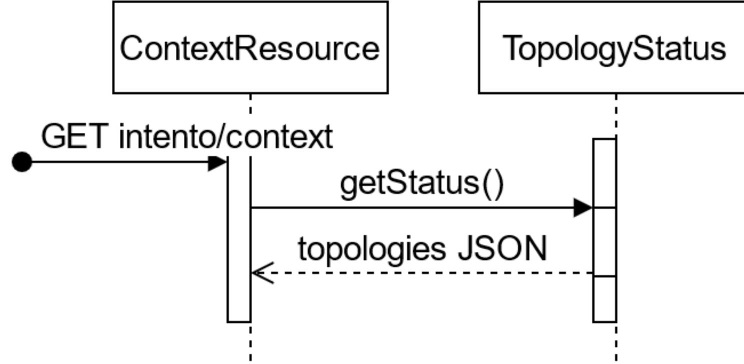


Figure 3.7: Sequence diagram of the execution after an API request.

optical node simulators. The ONOS driver, developed as part of this work, facilitates the integration of a centralized service orchestrator in the framework (i.e., OSMANO). It retrieves and combines all the network data (network service, topology, and power related data) from the Level-1 SDN controllers and Optical nodes at the Southbound and creates an orchestration API which allows the retrieval of this data at ONOS (level-2) Northbound. Additionally, the ONOS driver also facilitates the creation and deletion of the connectivity service between N number of optical nodes based on the instructions received from North-bound. Currently, the level-1 SDN controller can communicate and control the optical node simulators and perform tasks such as creating multiple interfaces on the simulation node, connecting multiple nodes, and defining a network topology. This information is made available at the controller's northbound through the T-API interface. As for the level-2 SDN controller, it is introduced to control and abstract large-scale networks individually handled by each level-1 SDN controller. An customized adapter has been developed as part of this project for the communication between the level-1 and level-2 SDN controllers based on T-API in order to transfer the knowledge of the network topology and other information from the level-1 SDN controllers to the orchestrator via the Orcestration API, developed as a part of this work. Additionally, the level-2 controller is able to perform the connectivity service mamagement i.e., creation and deletion of connectivity service between multiple nodes in order to optimize the network and ensure energy-efficiency in the network. The achieved aforementioned implementation is shown is Figure 3.8.

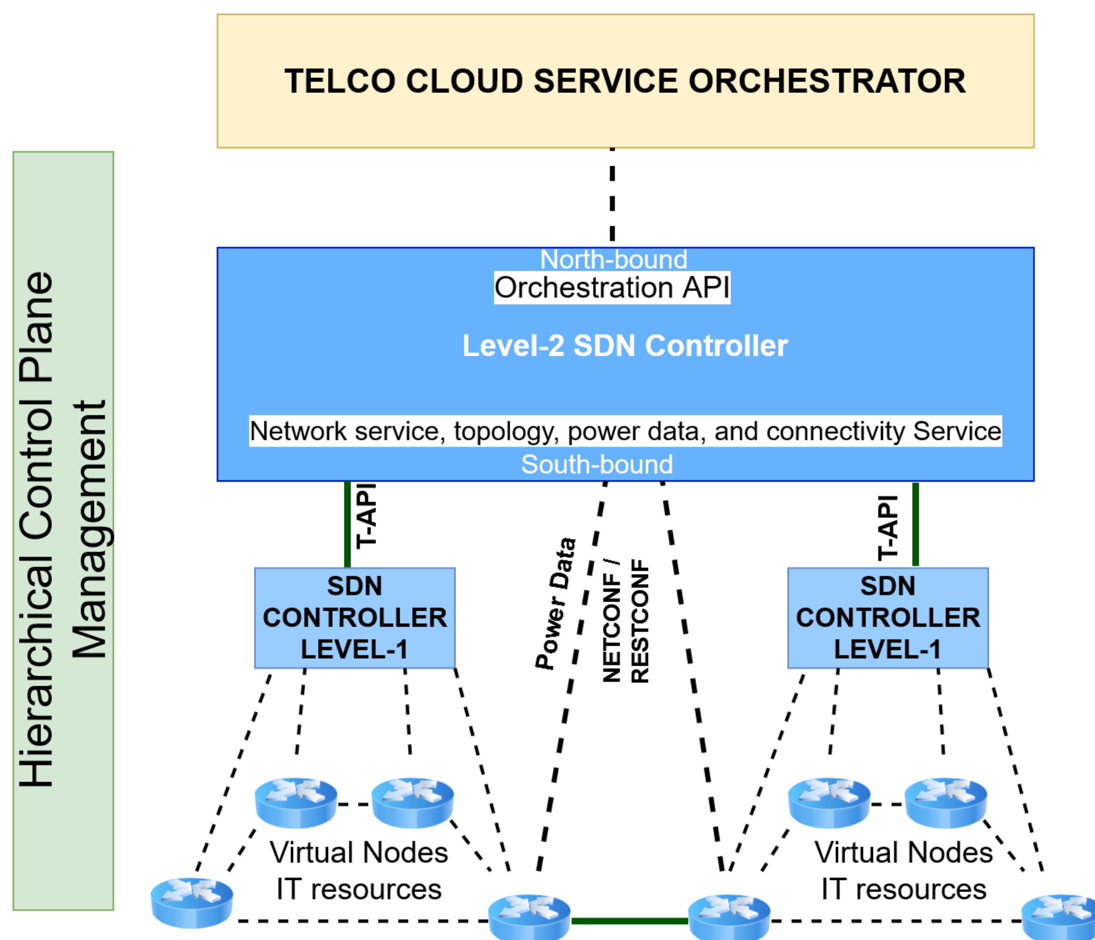


Figure 3.8: The achieved implementation.

3.3.7 Automated deployment

An automated procedure has been developed using Ansible, containing YAML files so that any layman can install the orchestration framework on their systems. It creates a 3-layer simulation environment that installs and integrates the level-1 and 2 SDN controllers along with optical node simulators within the OpenStack cloud on the given remote system as depicted in Figure 3.10. The aim of developing an assisted procedure is to minimize the installation time and reduce the complexity required for the creation of the simulation environment for testing. The base requirement for the automated procedure is a local or remote system running Ubuntu operating system.

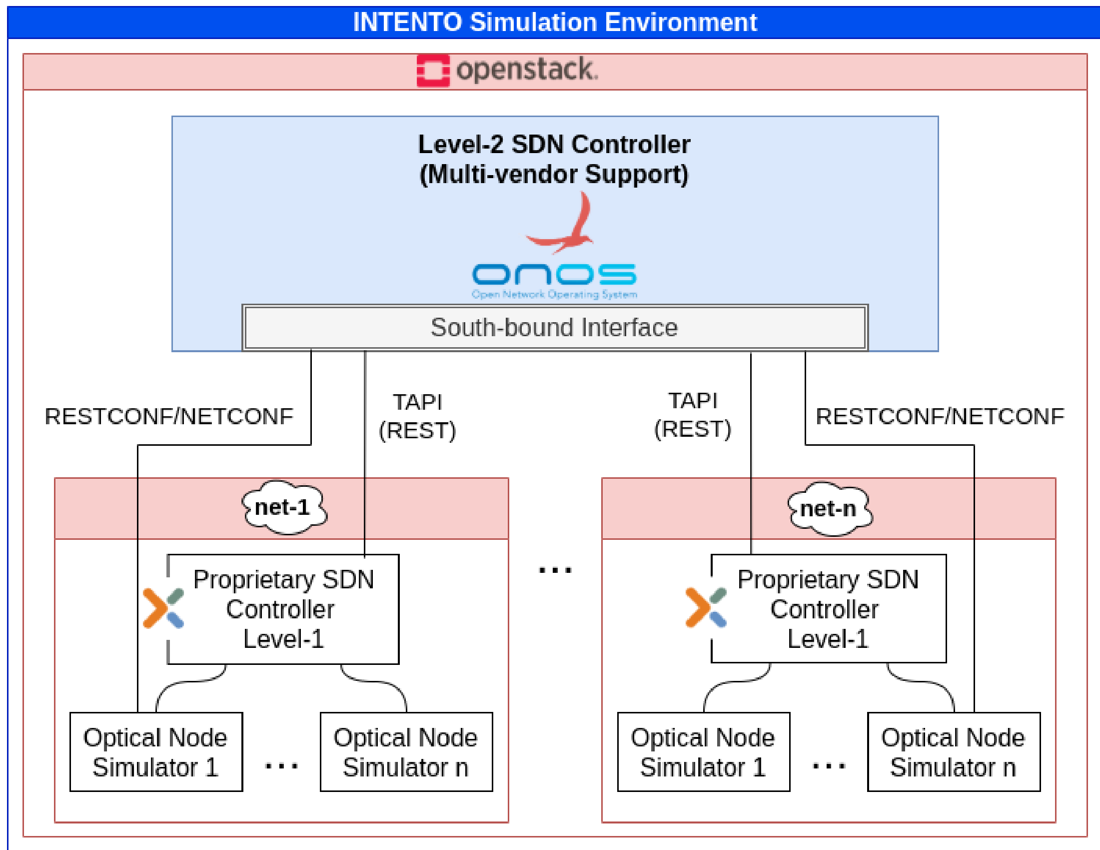


Figure 3.9: The developed simulation framework.

3.3.8 Demo of the simulated optical nodes

As shown in Figure 3.11, four interfaces are created on the optical node simulator representing a telecom node using the level-1 SDN controller within the conceived simulation environment.

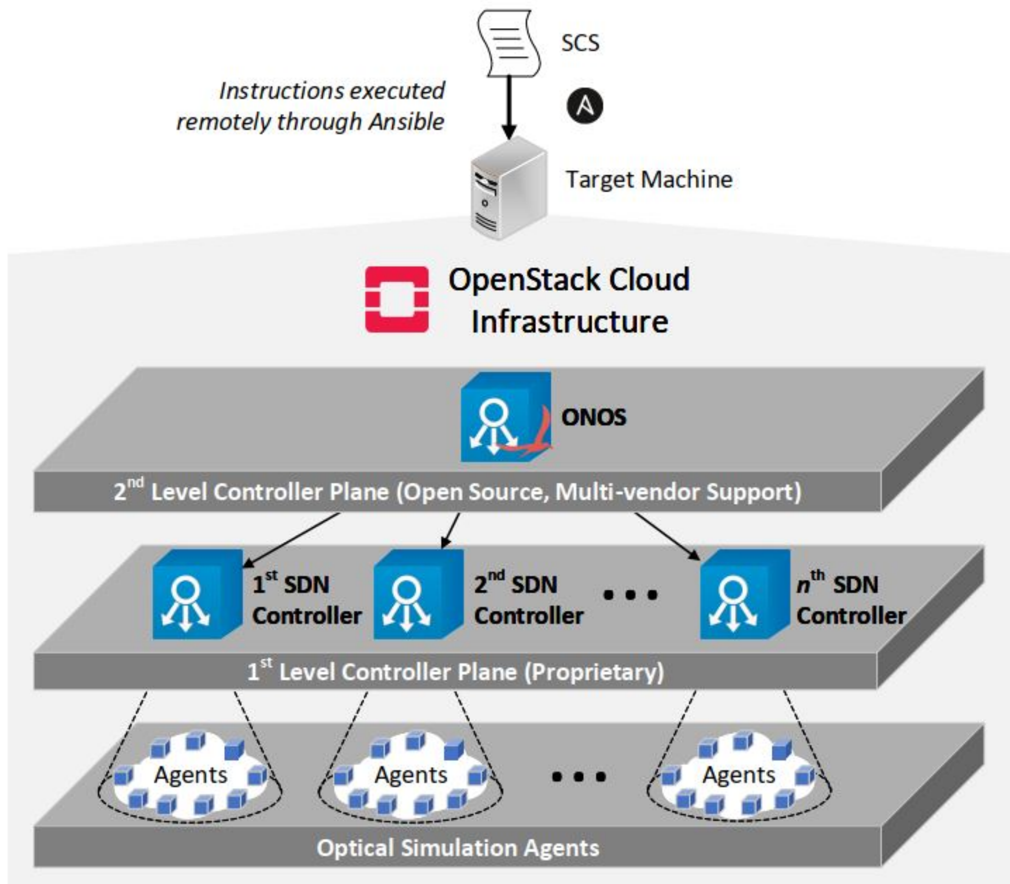


Figure 3.10: The automated deployment script.

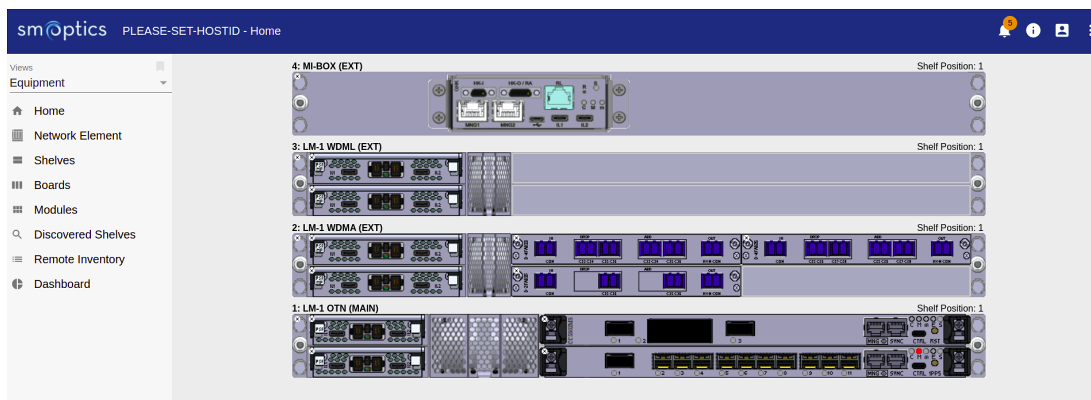


Figure 3.11: A virtual telecom node representing multiple interfaces on the optical node simulator.

3.4 Summary

Thanks to the recent advancements in the SDN and NFV research domains, telecom operators are encouraged to upgrade their optical transport networks towards programmable, energy-efficient, service-oriented, and interoperable architectures. The availability of a large set of open-source building blocks, supported by different standardization bodies makes the selection and the integration of such technologies a very complex task. In this context, the objective is to create an innovative simulation framework by selecting the best technologies and use it to test applications, services, and advanced optimization algorithms in a real environment. In this chapter, a large-scale, distributed, and hierarchical Transport SDN architecture has been designed, where optical switches and networking functionalities are monitored and dynamically configured through a two-level structure of SDN controllers. On top of that, Virtual Network Functions are optimally deployed and managed by a centralized orchestrator, based on network condition, user requests, and application requirements. Based on this architecture, a complex simulation environment has been developed, that harmoniously integrates within the OpenStack cloud: optical node simulators composed by simulation agent and a suitable hardware emulation layer; proprietary SDN network controller designed to enable the innovative optical nodes characteristics; Open Network Operating System as the second level controller, enabling the integration of third-party or standardized models (multi-vendor environment), based on standardized interfaces and communication protocols. The main components and implemented functionalities into the simulation framework are described in detail.

4

Designing of Dynamic Forwarding Strategy with Energy and Bandwidth Constraints

The management of forwarding rules into the resulting T-SDN architecture is an ambitious task. In this context, this chapter first investigates the problems presented in the current literature related to the forwarding strategies and then provides a comparison between the state-of-the-art routing techniques available in the current literature. Then, it formulates a novel methodology for the dynamic and reactive management of forwarding rules in a (potentially large-scale) T-SDN network by taking into account the energy and quality of service requirements. The effectiveness of the proposed approach has been investigated through experimental tests and compared against another reference scheme.

4.1 Routing Strategies discovered in the state of the art

This section provides a comparison between some of the important routing strategies recently published in the literature.

Several solutions in the current scientific literature address energy and bandwidth constraints almost separately. From one hand, energy-efficient schemes try to turn off as more optical switches and transport links as possible. Starting from the knowledge of network topology and the expected set of data flows (declared through the so-called

traffic matrix), available solutions configure forwarding rules by solving optimization problems [98, 99, 100, 101, 102] or by executing heuristic algorithms [103, 104, 105, 106, 107, 108]. With these mechanisms, most of the network traffic is forwarded through a reduced set of links. Therefore, flow dynamics generally bring to network congestion issues. From another hand, the rest of contributions (see [109] and [110] for example) only targets quality of service requirements, while missing the energy constraints.

At the time of this writing, the energy consumption and bandwidth constraints are jointly considered in [111], [112], and [113]. Specifically, [111] presents a multi-objective algorithm that derives the set of links to disable, while fulfilling the expected quality of service constraints. Here, forwarding rules are configured by one of the nodes of the network (acting as a controller) through in-band communications. This, however, increases the latencies of the exchange of control messages, as well as makes the resulting implementation infeasible in large-scale scenarios. In fact, the in-band communication approach is optimal in non-dynamic situations where it is not necessary to update the forwarding rules every few seconds, but not for a dynamic environment because the benefits arising from the presence of a controller interacting with optical switches by means of out-band communications are ignored during the in-band communication mode. The heuristic approach introduced in [112] configures forwarding rules by creating spanning trees of nodes with assigned weights according to their energy consumption. Unfortunately, it does not envisage to monitor the actual traffic load, thus being unable to react to data flow dynamics and congestion episodes. Finally, the work presented in [113] assumes to dynamically configure forwarding rules by taking into account the expected traffic volume and by targeting the shutdown of as many transport links as possible. This solution surely limits the energy consumption, but still lacks in reacting to the variability of the actual traffic load.

Characterizing the state-of-the-art, it is eventually desired that a novel methodology should be designed that could be dynamically reactive to the traffic variability and achieve an improved quality of service with low energy consumption in potentially large-scale T-SDN networks. Thanks to the SDN paradigm, this ambitious task can be achieved through utilizing the benefits of its architecture, like optimal communication control architecture and generalized protocols (e.g., OpenFlow, RESTCONF, NETCONF, and T-API) that facilitates the communication between network elements apart from any vendor-specific requirements.

4.2 The reference architecture and main assumptions

Figure 4.1 shows the reference T-SDN network considered in this work. According to the well-known SDN reference model, physical nodes and logical entities are grouped into three layers: infrastructure, control, and application [114]. The infrastructure layer embraces optical switches of the core network and edge routers. Optical switches forward data flows within the core network, according to the configured routing rules. Edge routers, instead, act as sources and destinations of data flows. Furthermore, a centralized controller monitors the infrastructure layer and dynamically configures forwarding rules based on the outcomes of the routing algorithm working at the application layer.

Both Software-Defined Controller and optical switches implement the OpenFlow stack (southbound interface). The controller, implemented with OpenDaylight framework, periodically queries optical switches for collecting details about the network topology and the amount of bandwidth consumed by each physical port. When needed, it also delivers the new set of forwarding rules across the network. According to OpenFlow specifications, the communication in the southbound interface is managed by means of the REpresentational State Transfer (REST)CONF protocol [115]. The application entity implementing the routing algorithm and the controller interact with each other with RESTful Application Programming Interface (API)s [115]. In this case, the exchanged messages are encoded with the YANG data model (northbound interface) [116].

The design of the novel routing algorithm discussed herein grounds its roots on the following consideration. From one hand, the network operator knows the expected volume of traffic that can be generated between all possible pairs of source and destination edge routers. Such information is stored within the traffic matrix [114] and may vary during the time (e.g., the volume of traffic manageable by the T-SDN network in daily hours may be different from the one available during the night or weekend). On the other hand, the actual traffic load generated within the network may differ from the traffic matrix, spanning from a very limited percentage of the expected volume of traffic to its upper bound. This double level of dynamicity makes challenging the task performed by the routing algorithm. Indeed, the conceived methodology intends to configure the infrastructure layer by jointly considering information stored within

the traffic matrix and the traffic load managed by optical switches during the time, periodically monitored by the controller.

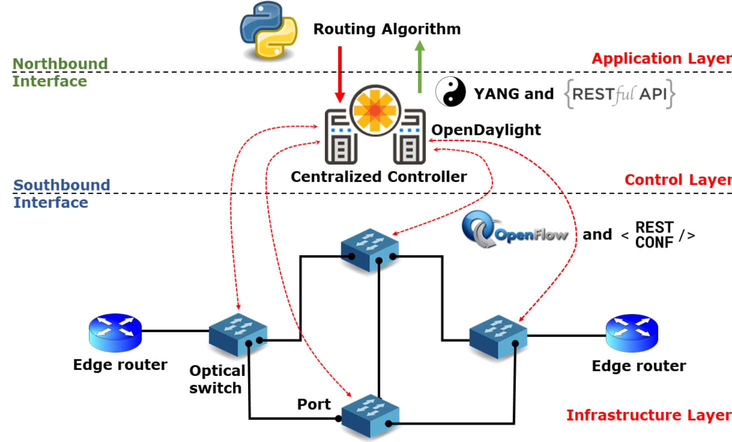


Figure 4.1: Reference T-SDN network architecture.

A power model helps to estimate the amount of power consumed by optical switches belonging to the reference T-SDN network. Without loss of generality, this work considers the model presented in [117], related to NEC PF 5240[1] OpenFlow switches. Here, the total amount of power consumed by an optical switch is given by five contributions:

- the amount of power required to keep the switch on, $P_{\text{base}} = 118.30\,\text{W}$;
- the amount of power needed to configure device settings and active ports, $P_{\text{conf}} = 0.52\,\text{W}$;
- the amount of power needed to install a new OpenFlow rule $P_{\text{flow-mod}} = 20.00\,\mu\text{W}$;
- the amount of power consumed for each control packet $P_{\text{packet}} = 711.00\,\mu\text{W}$;
- the amount of power consumption due to the processing of data flow $P_{\text{flow}} \ll 1\,\mu\text{W}$.

The analysis presented in [117] already demonstrated that the processing of data flows has a very minimal effect on the overall power consumption. Accordingly, it is possible to neglect the impact of P_{flow} and develop a strategy based on a traffic independent power model.

[1] https://www.necam.com/sdn/Hardware/PF5240Switch/ (Accessed: 2020-04-10)

4.3 The conceived approach

The routing algorithm conceived in this work periodically implements two different tasks. The first one provides an initial configuration of the T-SDN core network, based on the knowledge of the network topology and the expected volume of traffic declared by the traffic matrix. Therefore, it is executed only once, at the beginning of the validity period of the traffic matrix. The second task, instead, is implemented every congestion observation window and provides periodic updates of forwarding rules, based on the actual traffic load passing through the network. In order to effectively react to possible congestion episodes, the duration of the congestion observation window is much smaller than the validity period of the traffic matrix (i.e., tens of seconds instead of hours).

4.3.1 Initial network configuration based on the traffic matrix (Task 1).

It intends to reduce the overall power consumption by turning off as many devices and links as possible, while ensuring communication paths for any data flow reported in the traffic matrix. To this end, the network is modeled as an undirected graph G, where nodes represent optical switches and edges represent the transport links connecting optical switches. The set of demands D representing the traffic matrix is described by the pair of source node s and destination node t with their respective bandwidth demand d^{st}. Nodes belonging to the graph G are sorted according to their power consumption, from the most consuming device to the less consuming one. Links, instead, are randomly ordered. Then, an iteration on nodes is performed. At each iteration, the considered node in the ordered set and all of its links are tentatively turned off. Indeed, it is verified if at least one path exists for each traffic request declared in the traffic matrix. In the affirmative case, that node is removed from G since it is not necessary for the fulfillment of all traffic requests. Otherwise, the considered node and its links are left active into the network. Once the iteration on the nodes is completed, the same procedure is applied to the links. Also, in this case, the goal is to turn off unuseful or redundant links and leave active only a subset of links that guarantees the presence of communication paths for all data flows declared into the traffic matrix. A minimized graph G' is obtained at the end, which represents the network topology guaranteeing the greatest energy savings.

Given the minimized graph G', the shortest communication path for each data flow of the traffic matrix is identified according to the Dijkstra algorithm [118]. The calculated shortest paths are converted to forwarding rules and pushed on OpenFlow switches by the controller.

4.3.2 Redefinition of forwarding rules based on congestion episodes (Task 2).

In a dynamic environment where the actual traffic load changes, this task further adapts forwarding rules based on user demands and link capacity. To this end, the controller periodically sends OpenFlow messages to the switches, requesting information about the bandwidth consumption of their enabled ports. This helps to identify the activation of new flows that may congest transport links and provoke service degradation. This monitoring procedure allows to detect link congestion when the total bandwidth of the considered link is at least 90% occupied. Once detected the overloaded links, the data flows triggering that event are put within the congestion list. The recursive algorithm discussed before is implemented again over the network topology that excludes congested links. As a consequence, the algorithm will turn on transport links or optical switches that were turned off at the beginning. Then, a new shortest path is defined for each data flow in the congestion list, converted to forwarding rules, and pushed on OpenFlow switches. At the end of the congestion observation window, the network is configured as indicated by the first task. Therefore, congestion episodes are periodically managed, starting from a baseline network configuration.

To provide further insight, the pseudo-code describing the main functionalities of the conceived approach has been reported in Algorithm 1.

Algorithm 1 Pseudo code of the proposed methodology

Require: Graph G(nodes, links), set D of demand with traffic requirement d^{st} $\forall(s,t) \in D$

Ensure: Updated flow tables, Final graph

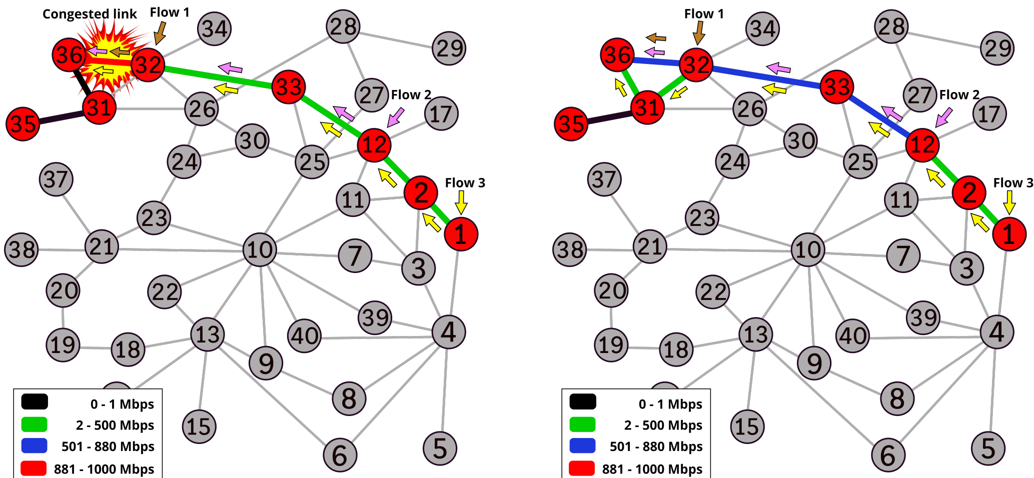
```
    TASK 1 (G, D):
 1: G' ← G
    Nodes Optimization on G'          ▷ Nodes are sorted in a most power order.
 2: for i ← 1 to nodes do
 3:     turn_off(nodes[i])
 4:     for all (s, t) ∈ D do
 5:         if !path_exists(s, t) then
 6:             turn_on(nodes[i])
 7:         end if
 8:     end for
 9: end for
    Links Optimization on G'          ▷ Links are selected in random order.
10: for i ← 1 to links do
11:     turn_off(links[i])
12:     for all (s, t) ∈ D do
13:         if !path_exists(s, t) then
14:             turn_on(links[i])
15:         end if
16:     end for
17: end for
    Push Forwarding Rules
18: for all (s, t) ∈ D do
19:     path(s, t) ← Dijkstra algorithm
20:     push_flow_rules()
21: end for
    # Controller monitors links bandwidth consumption.#
    TASK 2 (G, D):
22: if congestion_occurs then
23:     Nodes Optimization
24:     Links Optimization
25:     for all (s, t) ∈ D do
26:         path(s, t) ← Dijkstra algorithm
27:         for i ← 1 to link_in_path do
28:             if remaining_link_capacity < d^st and link_overloaded then
29:                 Congestion_list ← (s, t)
30:             end if
31:         end for
32:         if !Congestion_list.contains(s, t) or
    any path without overloaded links exists to satisfy (s, t) then
33:             update(remaining_link_capacity)
34:             push_flow_rules()
35:         end if
36:         G'' ← remove_overloaded_links(G)
37:         if !Congestion_list.empty() then
38:             Run TASK 2(G'', Congestion_list)
39:         end if
40:     end for
41: end if
```

4.4 Performance Evaluation

The performance of the proposed approach is experimentally investigated by emulating a T-SDN architecture within a desktop computer Intel Core i7-7700, RAM 16GB, with Ubuntu 18.04 64-bit. Specifically, the GÉANT topology with 40 nodes and 58 bidirectional links is implemented with Mininet, since it allows to virtualize a network of OpenFlow switches with Open vSwitch kernel. The OpenDaylight framework is used as the network controller. The routing algorithm has been developed in Python. Without loss of generality, the conducted analysis considers transport links supporting 1 Gbps of bandwidth. A traffic matrix is arbitrarily created in order to describe the data flows expected between up to 24 host pairs randomly attached to the GÉANT topology. Each data flow in the traffic matrix presents a request rate of 400 Mbps. The actual traffic load is generated by activating a percentage of requests declared in the traffic matrix. To provide further insights, the performance of the proposed approach has been compared with respect to a state of the art the algorithm presented in [108].

Figure 4.2a depicts a simplified example showing the ability of the conceived solution to successfully react to congestion episodes. The example considers three data flows, asking for 400 Mbps of bandwidth each, directed to the same destination. Figure 4.2a represents the network topology configured according to the algorithm presented in [108]. It is possible to observe that the link connecting node 32 to node 36, which only offers 1 Gbps of bandwidth, is congested. This means that the strategy presented in [108] is not able to fulfill the quality of service levels requested by the considered data flow. Note that the initial network configuration provided by Task 1 of the algorithm presented in Section 4.3 coincides with the one obtained through [108]. Differently, from [108], however, Task 2 implemented by the approach described in this work adapts forwarding rules in reaction to congestion episodes. Figure 4.2b clearly shows how the path followed by Flow 3 is updated. Accordingly, the link between node 32 and node 36 in the example is not overloaded and the quality of service requested by all the three flows is achieved. Quantitative key performance indicators discussed below include the total power consumption of the T-SDN network, the percentage of deactivated links, and the percentage of throughput degradation registered by active data flows.

The total amount of power consumed by an operating T-SDN network is evaluated by considering 6 to 24 active data flows, generating 100% of the data rate declared



(a) Network configuration based on [108].

(b) Network configuration achieved with the proposed approach.

Figure 4.2: Example showing the ability of the proposed approach to achieve energy and quality of service constraints

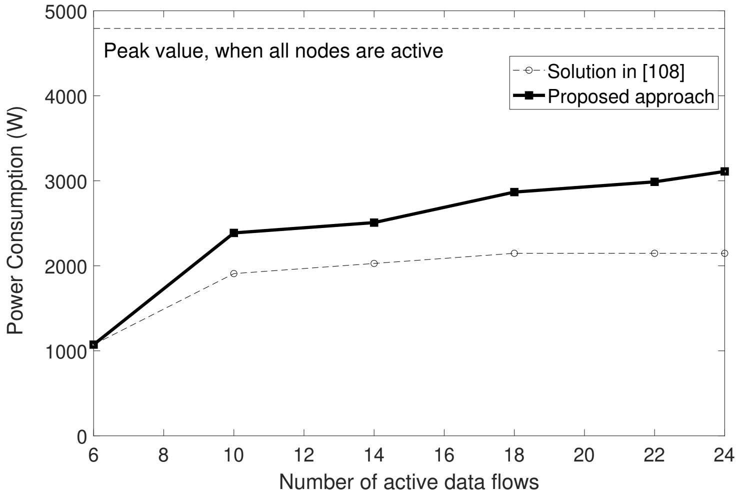


Figure 4.3: Power consumption.

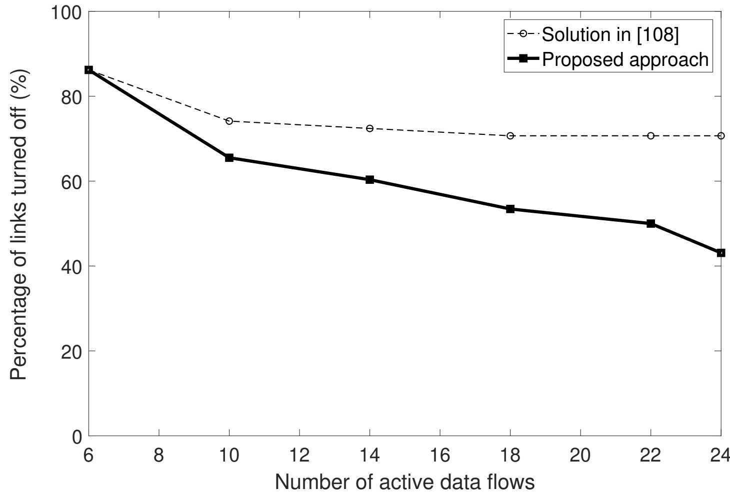


Figure 4.4: Percentage of links turned off.

in the traffic matrix (that is equal to 400 Mbps). In case, the network has all the optical switches and transport links turned on, the total power consumption is equal to 4792.32 W. This is reported in Figure 4.3 as the peak value achievable in the absence of any energy-aware routing strategy. The other two curves reported in Figure 4.3 shows the amount of power consumed by the considered T-SDN network as a function of the number of active data flows, when forwarding rules are set according to the algorithm presented in [108] and the solution conceived in this work. As expected, results show that the increment of the number of active data flows always requires higher number of optical switches and links to activate in the network. This inevitably brings to an increment of the overall power consumption. It is also evident that the algorithm presented in [108] ensures the highest power saving, thanks to its ability to shut down as many optical switches and transport links as possible, without taking care of the quality of service level offered to end-users. On the contrary, the methodology presented in the work registers a slight increment of the power consumption due to the activation of more optical switches and links, triggered in answer to congestion events.

Figure 4.4, showing the percentage of deactivated transport links, fully confirms the aforementioned discussion: the proposed solution forwards data flows through a higher number of uncongested paths. It is also possible to observe that the difference between the two investigated approaches becomes more evident when the number of active data

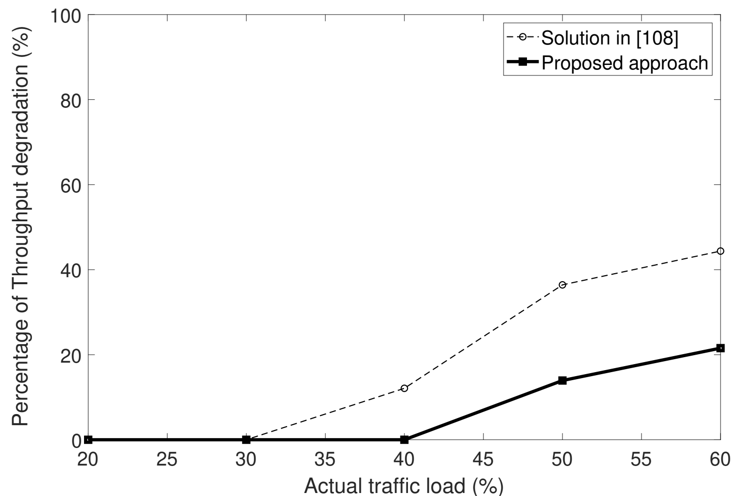


Figure 4.5: Throughput degradation registered by active data flows

flows increases. In this case, in fact, the higher the bandwidth requirement, the higher the number of paths to activate for avoiding network congestion.

The real effectiveness of the conceived solution is highlighted in Figure 4.5, which reports the degradation of the throughput registered by active data flows due to bandwidth constraints, measured as a function of the traffic load (expressed as a percentage of the bandwidth requirement declared in the traffic matrix). Since [108] does not apply any re-routing strategy after the congestion, a large traffic load seriously degrades network performance. The approach presented in this work exhibits the lowest throughput with respect to [108], thus demonstrating its successful ability to redirect data flows across uncongested paths. From these considerations, it is evident that the strategy presented in this work provides a significant gain in terms of performance at the expense of a limited decrease of registered energy-saving as compared to [108].

4.5 Summary

This chapter focuses on addressing the challenges highlighted in the state of the art related to forwarding strategies in T-SDN networks. To address the highlighted challenges, a novel methodology has been formulated for the dynamic management of forwarding rules in the presence of energy and bandwidth constraints. The proposed approach configures communication paths and decides optical switches and transport

links to be activated by jointly considering the network topology, the power consumption of optical switches, the expected volume of traffic, and variability of the actual traffic load. Experimental tests demonstrated its ability to achieve the best compromise between the power consumption of the overall network and the quality of service offered to end-users. The proposed forwarding strategy will be implemented into the conceived framework of this work in future works, to achieve quality of service and energy efficiency in the network.

5

Conclusions and Future Research Directions

The last decade has been the witness of rapid adaptation of the cloud computing and SDN paradigm. Billions of applications and services are deployed everyday in the cloud. End-users access these services using the Internet and other communication networks. To facilitate the needs of end-users, Telco operators started to enhance their network infrastructure using the SDS vision by introducing SDN and NFV based network virtualization techniques to fully utilize their existing resources. However, this implementation was not straight-forward and posed numerous challenges in terms of: selection of the state of the art technologies demonstrating the joint-integration of available tools, lack of simulation frameworks for the development and deployment of VNFs in complex real-time environments, and forwarding strategies considering low energy consumption while ensuring better quality of service to user demands.

Throughout the PhD, the focus has been on the aforementioned issues faced by the industry. With reference to the enabling technologies for SDS, a study has been carried out in Chapter 1 that deeply reviewed the state of the art technologies for SDS including container engines, orchestrators, load-balancers, service discovery, other tools etc. A qualitative cross-comparison has been performed to highlight the pros and cons of the technologies based on a set of KPIs. The comparison highlights that Docker is an emerging leading container engine, offering a strong set of features to allow the usage of a large number of supporting tools. At the same time, it is also remarked that Docker Swarm and Kubernetes appears to be a very promising container orchestrators because

it comes up with its own scheduler, load balancer, and service discovery features. This investigation was aimed at providing a clear roadmap for the selection of technologies to the SMEs who are reluctant to adopt SDS for launching their services and applications.

Taking a step ahead, to demonstrate the effectiveness of the technologies highlighted as outcome of the work performed in Chapter 2, physical deployment of these technologies was performed within in real-time virtualized service infrastructure developed inside the bare-metal and OpenStack cloud environments. In particular, the set of investigated technologies are Docker as container engine, Docker Swarm and Kubernetes as orchestrators with load balancing and service discovery capabilities, bare-metal and OpenStack cloud as deployment platforms, and Docker-compose, Docker-Machine, Kubeadm, and Flannel as supporting tools for scheduling and deployment functionalities. Four testbeds were designed to perform experimental tests in a high-load scenario in order to evaluate the ability of the virtualized service infrastructure to provide answers to multiple user requests received from real-world network. The conducted study revealed that the integration of Docker engine and Kubernetes orchestrator within the bare-metal deployment platform ensures better performance. Then, to demonstrate the effectiveness in an industrial use case with mobility, the performance of the most suitable technologies was evaluated in a smart farm use case, integrating mobile drones and complex image processing tasks. The results revealed that the virtualized service infrastructure still guarantee acceptable quality of service despite the execution of heavy tasks and influence of users' mobility, respectively. Indeed, the results presented in Chapter would facilitate the SMEs in the selection of technologies for container networking.

To further proceed towards the goal, in Chapter 3, a T-SDN based hierarchical orchestration framework was designed by selecting the best set of technologies to test services and applications in a real environment. The architecture of the framework has been developed within the OpenStack cloud consisting functionalities of optical node simulation agents, two-levels of SDN controllers (one for managing and simulating the advanced optical nodes and other for integrating multi-vendor and third party environments), and communication protocols. The framework is capable of providing the orchestration to the optical nodes thanks to the level-2 SDN controller (ONOS) managing the creation/deletion of the connectivity service between the multiple simulation nodes connected with the level-1 SDN controller. Additonally, the Power Management

and monitoring perspective for the energy optimization of the framework, implementation at the ONOS southbound for retrieving the actual power consumption data from the optical node simulators, and the connectivity service perspectives to instruct the standby/wakeup statuses, SNMP Management of level-1 SDN controller at the ONOS, and connectivity of the ONOS with the orchestrator.

Moreover, the work extended towards the design of a novel routing strategy for T-SDN based networks, that is dynamically reactive and capable of ensuring energy efficiency and quality of service within the network based of changing user demands. The details of the proposed routing strategy as presented in Chapter 4, starts by configuring communication paths and decides which optical switches and transport links to be activated by jointly considering the network topology, the power consumption of optical switches, the expected volume of traffic, and variability of the actual traffic load. The experimental analysis results demonstrated its ability to achieve the best compromise between the power consumption of the overall network and the quality of service offered to end-users.

Finally, future research activities will analyze the complexity and investigate the behavior of the conceived hierarchical framework in more complex and large scale network while considering realistic traffic matrix and flow generation statistics. Moreover, it will also investigate the adoption of developed routing strategy in hierarchical T-SDN deployments, based on two layers of controllers introduced to improve scalability and provide a comprehensive comparison with the other available state of the art approaches, respectively. Additionally, this work can be significantly extended by further investigating energy and security issues in more complex and dynamic environments, also in the presence of heterogeneous and coexisting services.

References

[1] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," Proceedings of the IEEE, vol. 103, no. 1, pp. 14–76, 2015. 1

[2] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," IEEE Communications Magazine, vol. 53, no. 2, pp. 90–97, 2015. 1

[3] Y. Jararweh, M. Al-Ayyoub, E. Benkhelifa, M. Vouk, A. Rindos et al., "Sdiot: a software defined based internet of things framework," Journal of Ambient Intelligence and Humanized Computing, vol. 6, no. 4, pp. 453–461, 2015. 1

[4] Y. Alahmad, A. Agarwal, and T. Daradkeh, "High availability management for applications services in the cloud container-based platform," in Proc. of IEEE IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), 2018, pp. 1–8. 2

[5] J. Matias, J. Garay, N. Toledo, J. Unzilla, and E. Jacob, "Toward an sdn-enabled nfv architecture," IEEE Communications Magazine, vol. 53, no. 4, pp. 187–193, 2015. 2

[6] V. Kaushik, A. Sharma, and R. Tomar, "Virtualizing network functions in software-defined networks," in Innovations in Software-Defined Networking and Network Functions Virtualization. IGI Global, 2018, pp. 26–51. 2

[7] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in Proc. of ACM International Middleware Conference, 2016, p. 1. 2, 28

[8] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," IEEE Cloud Computing, no. 3, pp. 81–84, 2014. 2, 3

[9] X. Tang, F. Zhang, X. Li, S. U. Khan, and Z. Li, "Quantifying cloud elasticity with container-based autoscaling," Future Generation Computer Systems, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18307842 2

[10] S. V. Gogouvitis, H. Mueller, S. Premnadh, A. Seitz, and B. Bruegge, "Seamless computing in industrial systems using container orchestration," Future Generation Computer Systems, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17330236 2, 6

[11] C. de Alfonso, A. Calatrava, and G. Moltó, "Container-based virtual elastic clusters," Journal of Systems and Software, vol. 127, pp. 1–11, 2017. 2, 4, 27

[12] J. Claassen, R. Koning, and P. Grosso, "Linux containers networking: Performance and scalability of kernel modules," in Proc. of IEEE/IFIP Network Operations and Management Symposium (NOMS), 2016, pp. 713–717. 2

[13] V. Marmol, R. Jnagal, and T. Hockin, "Networking in containers and container clusters," Proceedings of netdev 0.1, February, 2015. 2

[14] R. Cziva, S. Jouet, K. J. White, and D. P. Pezaros, "Container-based network function virtualization for software-defined networks," in Proc. of IEEE Symposium on computers and communication (ISCC), 2015, pp. 415–420. 2

[15] Ł. Makowski and P. Grosso, "Evaluation of virtualization and traffic filtering methods for container networks," Future Generation Computer Systems, vol. 93, pp. 345–357, 2019. 2

[16] J. P. Martin, A. Kandasamy, and K. Chandrasekaran, "Exploring the support for high performance applications in the container runtime environment," Human-centric Computing and Information Sciences, vol. 8, no. 1, p. 1, 2018. 2, 3, 4

[17] T. Adufu, J. Choi, and Y. Kim, "Is container-based technology a winner for high performance scientific applications?" in Proc. of IEEE Network Operations and Management Symposium (APNOMS),, 2015, pp. 507–510. 3, 28

[18] P. S. Kocher, Microservices and Containers. Addison-Wesley Professional, 2018. 3, 6

[19] F. Paraiso, S. Challita, Y. Al-Dhuraibi, and P. Merle, "Model-driven management of docker containers," in Proc. of IEEE International Conference on Cloud Computing (CLOUD), 2016, pp. 718–725. 4, 6

[20] C. Kniep, "Containerization of high performance compute workloads using docker," doc.qnib.org, 2014. 4

[21] J.-S. Ma, D.-J. Kang, and H.-Y. Kim, "The lxc-lxd virtualization in arm64bit x-gene2 server," in International Conference on Green and Human Information Technology. Springer, 2018, pp. 168–175. 4

[22] C. Abdelmassih, "Container orchestration in security demanding environments at the swedish police authority," 2018. 4, 6

[23] A. Mouat, Orchestrating, clustering, and managing containers. O'Reilly Media, Inc., 2016. 6, 7, 12

[24] A. Corradi, M. Fanelli, and L. Foschini, "Vm consolidation: A real case based on openstack cloud," Future Generation Computer Systems, vol. 32, pp. 118–127, 2014. 7

[25] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Openstack: toward an open-source solution for cloud computing," International Journal of Computer Applications, vol. 55, no. 3, pp. 38–42, 2012. 7, 27

[26] K. Cacciatore, P. Czarkowski, S. Dake, J. Garbutt, B. Hemphill, J. Jainschigg, A. Moruga, A. Otto, C. Peters, and B. E. Whitaker, "Exploring opportunities: Containers and openstack," OpenStack White Paper, vol. 19, 2015. 7, 27

[27] C. G. Kominos, N. Seyvet, and K. Vandikas, "Bare-metal, virtual machines and containers in openstack," in Proc. of IEEE Conference on Innovations in Clouds, Internet and Networks (ICIN), 2017, pp. 36–43. 7, 24, 27

[28] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in Proc. of IEEE International Conference on Cloud Engineering (IC2E), 2016, pp. 202–211. 7

[29] M. Hausenblas, Container Networking. O'Reilly Media, Inc., 2018. 7, 9, 10, 11, 12

[30] A. Goder, A. Spiridonov, and Y. Wang, "Bistro: Scheduling data-parallel jobs against live production systems." in USENIX Annual Technical Conference, 2015, pp. 459–471. 8

[31] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," IEEE communications surveys & tutorials, vol. 16, no. 1, pp. 493–512, 2014. 14

[32] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69–74, 2008. 14

[33] R. Enns, "Netconf configuration protocol," Tech. Rep., 2006. 14

[34] A. Bierman, M. Bjorklund, and K. Watsen, "Restconf protocol," Tech. Rep., 2017. 15

[35] T. Borangiu, D. Trentesaux, A. Thomas, P. Leitão, and J. Barata, "Digital transformation of manufacturing through cloud services and resource virtualization," Computers in Industry, vol. 108, pp. 150–162, 2019. 24

[36] A. Hughes and A. Awad, "Quantifying performance determinism in virtualized mixed-criticality systems," in 2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC). IEEE, 2019, pp. 181–184. 24

[37] F. Rodríguez-Haro, F. Freitag, L. Navarro, E. Hernánchez-sánchez, N. Farías-Mendoza, J. A. Guerrero-Ibáñez, and A. González-Potes, "A summary of virtualization techniques," Procedia Technology, vol. 3, pp. 267–272, 2012. 24

[38] A. M. Joy, "Performance comparison between linux containers and virtual machines," in 2015 International Conference on Advances in Computer Engineering and Applications. IEEE, 2015, pp. 342–346. 24, 27, 28, 29

[39] J. Bhimani, Z. Yang, M. Leeser, and N. Mi, "Accelerating big data applications using lightweight virtualization framework on enterprise cloud," in 2017 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2017, pp. 1–7. 24

[40] S. S. Tadesse, C. F. Chiasserini, and F. Malandrino, "Characterizing the power cost of virtualization environments," Transactions on Emerging Telecommunications Technologies, vol. 29, no. 8, p. e3462, 2018. 27

[41] C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures–a technology review," in 2015 3rd international conference on future internet of things and cloud. IEEE, 2015, pp. 379–386. 24

[42] I. Farris, T. Taleb, H. Flinck, and A. Iera, "Providing ultra-short latency to user-centric 5g applications at the mobile network edge," Transactions on Emerging Telecommunications Technologies, vol. 29, no. 4, p. e3169, 2018.

[43] F. E. da Silva Barbosa, F. F. de Mendonça Júnior, and K. L. Dias, "A platform for cloudification of network and applications in the internet of vehicles," Transactions on Emerging Telecommunications Technologies, vol. 31, no. 5, p. e3961, 2020. 24

[44] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," Future Generation Computer Systems, vol. 68, pp. 175–182, 2017. 24, 27, 28

[45] A. Andriyanto, R. Doss, and P. Yustianto, "Adopting soa and microservices for inter-enterprise architecture in sme communities," in 2019 International Conference on Electrical, Electronics and Information Engineering (ICEEIE), vol. 6. IEEE, 2019, pp. 282–287. 24

[46] M. Attaran and J. Woods, "Cloud computing technology: improving small business performance using the internet," Journal of Small Business & Entrepreneurship, vol. 31, no. 6, pp. 495–519, 2019. 24

[47] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, 2013, pp. 233–240. 24, 27

[48] R. Bonafiglia, I. Cerrato, F. Ciaccia, M. Nemirovsky, and F. Risso, "Assessing the performance of virtualization technologies for nfv: A preliminary benchmarking," in 2015 Fourth European Workshop on Software Defined Networks. IEEE, 2015, pp. 67–72. 27, 28

[49] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in 2016 International Conference on Computing, Networking and Communications (ICNC). IEEE, 2016, pp. 1–7. 28

[50] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai, "Using docker in high performance computing applications," in 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE). IEEE, 2016, pp. 52–57. 28

[51] S. Shirinbab, L. Lundberg, and E. Casalicchio, "Performance comparision between scaling of virtual machines and containers using cassandra nosql database," Cloud Computing, p. 103, 2019. 28

[52] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in 2015 IEEE international symposium on performance analysis of systems and software (ISPASS). IEEE, 2015, pp. 171–172. 28

[53] C. C. Spoiala, A. Calinciuc, C. O. Turcu, and C. Filote, "Performance comparison of a webrtc server on docker versus virtual machine," in 2016 International Conference on Development and Application Systems (DAS). IEEE, 2016, pp. 295–298. 28

[54] J. Higgins, V. Holmes, and C. Venters, "Orchestrating docker containers in the hpc environment," in International Conference on High Performance Computing. Springer, 2015, pp. 506–513. 28

[55] E. Casalicchio and V. Perciballi, "Measuring docker performance: What a mess!!!" in Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. ACM, 2017, pp. 11–16. 28

[56] E. Preeth, F. J. P. Mulerickal, B. Paul, and Y. Sastri, "Evaluation of docker containers based on hardware utilization," in 2015 International Conference on Control Communication & Computing India (ICCC). IEEE, 2015, pp. 697–700. 28

[57] B. Ruan, H. Huang, S. Wu, and H. Jin, "A performance study of containers in cloud environment," in Asia-Pacific Services Computing Conference. Springer, 2016, pp. 343–356.

[58] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, "Performance overhead comparison between hypervisor and container based virtualization," in 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA). IEEE, 2017, pp. 955–962. 28

[59] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," IEEE Communications Surveys & Tutorials, vol. 20, no. 3, pp. 2429–2453, 2018. 28

[60] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in 2015 IEEE International Conference on Cloud Engineering. IEEE, 2015, pp. 386–393. 28

[61] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in Proceedings of the 26th Symposium on Operating Systems Principles. ACM, 2017, pp. 218–233. 28

[62] A. Acharya, J. Fanguède, M. Paolino, and D. Raho, "A performance benchmarking analysis of hypervisors containers and unikernels on armv8 and x86 cpus," in 2018 European Conference on Networks and Communications (EuCNC). IEEE, 2018, pp. 282–9. 28

[63] J. Struye, B. Spinnewyn, K. Spaey, K. Bonjean, and S. Latré, "Assessing the value of containers for nfvs: A detailed network performance study," in 2017 13th International Conference on Network and Service Management (CNSM). IEEE, 2017, pp. 1–7.

[64] F. Ramalho and A. Neto, "Virtualization at the network edge: A performance comparison," in 2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM). IEEE, 2016, pp. 1–6.

[65] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Disaggregated fpgas: Network performance comparison against bare-metal servers, virtual machines and linux containers," in 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2016, pp. 9–17.

[66] R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," in 2016 International Conference on Computing, Communication and Automation (ICCCA). IEEE, 2016, pp. 1204–1210. 28

[67] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding performance of i/o intensive containerized applications for nvme ssds," in 2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC). IEEE, 2016, pp. 1–8. 28

[68] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," Advanced Science and Technology Letters, vol. 66, no. 105-111, p. 2, 2014. 27, 28

[69] Y. Yamato, "Openstack hypervisor, container and baremetal servers performance comparison," IEICE Communications Express, vol. 4, no. 7, pp. 228–232, 2015. 27, 28

[70] ——, "Performance-aware server architecture recommendation and automatic performance verification technology on iaas cloud," Service Oriented Computing and Applications, vol. 11, no. 2, pp. 121–135, 2017. 27, 28

[71] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: a state-of-the-art review," IEEE Transactions on Cloud Computing, 2017. 28, 29

[72] E. Truyen, D. Van Landuyt, B. Lagaisse, and W. Joosen, "Performance overhead of container orchestration frameworks for management of multi-tenant database deployments," in Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. ACM, 2019, pp. 156–159. 24, 28, 29

[73] C. M. Chan, S. Y. Teoh, A. Yeow, and G. Pan, "Agility in responding to disruptive digital innovation: Case study of an sme," Information Systems Journal, vol. 29, no. 2, pp. 436–455, 2019. 24

[74] A. A. Shah, G. Piro, L. A. Grieco, and G. Boggia, "A qualitative cross-comparison of emerging technologies for software-defined systems," in 2019 Sixth International Conference on Software Defined Systems (SDS). IEEE, 2019, pp. 138–145. 25, 29, 67

[75] R. Kumar, N. Gupta, S. Charu, K. Jain, and S. K. Jangir, "Open source solution for cloud computing platform using openstack," International Journal of Computer Science and Mobile Computing, vol. 3, no. 5, pp. 89–98, 2014.

[76] X. Wen, G. Gu, Q. Li, Y. Gao, and X. Zhang, "Comparison of open-source cloud management platforms: Openstack and opennebula," in 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery. IEEE, 2012, pp. 2457–2461.

[77] T. Rosado and J. Bernardino, "An overview of openstack architecture," in Proceedings of the 18th International Database Engineering & Applications Symposium. ACM, 2014, pp. 366–367.

[78] Y. Yamato, Y. Nishizawa, M. Muroi, and K. Tanaka, “Development of resource management server for production iaas services based on openstack,” Journal of Information Processing, vol. 23, no. 1, pp. 58–66, 2015.

[79] R.-A. Cherrueau, A. Lebre, D. Pertin, F. Wuhib, and J. M. Soares, “Edge computing resource management system: a critical building block! initiating the debate via openstack,” in {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18), 2018. 27

[80] T. Yanagawa, “Openstack-based next-generation cloud resource management,” Fujitsu Sci. Tech. J, vol. 51, no. 2, pp. 62–65, 2015. 27

[81] S. Yeoman, “How secure are bare metal servers?” Network Security, vol. 2019, no. 2, pp. 16–17, 2019. 27

[82] G. Merlino, R. Dautov, S. Distefano, and D. Bruneo, “Enabling workload engineering in edge, fog, and cloud computing through openstack-based middleware,” ACM Transactions on Internet Technology (TOIT), vol. 19, no. 2, p. 28, 2019.

[83] A. Calinciuc, C. C. Spoiala, C. O. Turcu, and C. Filote, “Openstack and docker: building a high-performance iaas platform for interactive social media applications,” in 2016 International Conference on Development and Application Systems (DAS). IEEE, 2016, pp. 287–290.

[84] A. Lingayat, A. Singh, V. Naik, R. R. Badre, and A. K. Gupta, “Horizon, a web-based user interface for managing services in openstack: An introspection,” in 2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT). IEEE, 2018, pp. 1–6. 27

[85] P. Xu, S. Shi, and X. Chu, “Performance evaluation of deep learning tools in docker containers,” in 2017 3rd International Conference on Big Data Computing and Communications (BIGCOM). IEEE, 2017, pp. 395–403. 28

[86] P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M. L. Heuer, and C. Notredame, “The impact of docker containers on the performance of genomic pipelines,” PeerJ, vol. 3, p. e1273, 2015. 28

[87] R. Morabito, "A performance evaluation of container technologies on internet of things devices," in 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2016, pp. 999–1000. 28

[88] ——, "Virtualization on internet of things edge devices with container technologies: a performance evaluation," IEEE Access, vol. 5, pp. 8835–8850, 2017. 28

[89] R. Morabito, I. Farris, A. Iera, and T. Taleb, "Evaluating performance of containerized iot services for clustered devices at the network edge," IEEE Internet of Things Journal, vol. 4, no. 4, pp. 1019–1030, 2017. 28

[90] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud," in Proceedings of the Practice and Experience on Advanced Research Computing. ACM, 2018, p. 11. 28

[91] N. G. Bachiega, P. S. Souza, S. M. Bruschi, and S. d. R. de Souza, "Container-based performance evaluation: A survey and challenges," in 2018 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2018, pp. 398–403. 28

[92] F. Al-Turjman, "A novel approach for drones positioning in mission critical applications," Transactions on Emerging Telecommunications Technologies, 2019. 56

[93] ——, "Smart-city medium access for smart mobility applications in internet of things," Transactions on Emerging Telecommunications Technologies, p. e3723, 2019. 57

[94] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "Nfv: state of the art, challenges, and implementation in next generation mobile networks (vepc)," IEEE Network, vol. 28, no. 6, pp. 18–26, 2014. 60

[95] B. Lakshmi and J. Lakshmi, "Integrating service function chain management into software defined network controller," in 2019 IEEE World Congress on Services (SERVICES), vol. 2642. IEEE, 2019, pp. 160–165. 60

[96] M. Mechtri, C. Ghribi, O. Soualah, and D. Zeghlache, “Nfv orchestration framework addressing sfc challenges,” IEEE Communications Magazine, vol. 55, no. 6, pp. 16–23, 2017. 60

[97] M. Garrich, F.-J. Moreno-Muro, M.-V. B. Delgado, and P. P. Mariño, “Open-source network optimization software in the open sdn/nfv transport ecosystem,” Journal of Lightwave Technology, vol. 37, no. 1, pp. 75–88, 2019. 60

[98] A. K. Al Mhdawi and H. S. Al-Raweshidy, “iPRDR: Intelligent Power Reduction Decision Routing Protocol for Big Traffic Flood in Hybrid-SDN Architecture,” IEEE Access, vol. 6, pp. 10 944–10 955, 2018. 82

[99] F. Giroire, J. Moulierac, and T. K. Phan, “Optimizing Rule Placement in Software-Defined Networks for Energy-Aware Routing,” in Proc. of IEEE Global Communications Conference, 2014, pp. 2523–2529. 82

[100] H. Li, G. Jiang, and R. Chai, “Energy Consumption Optimization Based Joint Routing and Flow Allocation Algorithm for Software Defined Networking,” in Proc. of 19th International Symposium on Wireless Personal Multimedia Communications (WPMC), 2016, pp. 311–316. 82

[101] N. Vasić, P. Bhurat, D. Novaković, M. Canini, S. Shekhar, and D. Kostić, “Identifying and Using Energy-Critical Paths,” in Proc. of the Seventh COnference on emerging Networking EXperiments and Technologies, 2011, pp. 1–12. 82

[102] Z. Wu, X. Ji, Y. Wang, X. Chen, and Y. Cai, “An Energy-Aware Routing for Optimizing Control and Data Traffic in SDN,” in Proc. of 26th International Conference on Systems Engineering (ICSEng), 2018, pp. 1–4. 82

[103] B. G. Assefa and O. Ozkasap, “Link utility and traffic aware energy saving in software defined networks,” in Proc. of IEEE International Black Sea Conference on Communications and Networking, 2017, pp. 1–5. 82

[104] ——, “A Novel Utility Based Metric and Routing for Energy Efficiency in Software Defined Networking,” in Proc. of International Symposium on Networks, Computers and Communications, 2019, pp. 1–4. 82

[105] ——, "RESDN: A Novel Metric and Method for Energy Efficient Routing in Software Defined Networks," IEEE Transactions on Network and Service Management, pp. 1–1, 2020. 82

[106] M. K. Awad, Y. Rafique, S. Alhadlaq, D. Hassoun, A. Alabdulhadi, and S. Thani, "A Greedy Power-Aware Routing Algorithm for Software-Defined Networks," in Proc. of IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), 2016, pp. 268–273. 82

[107] Heller, B. and Seetharaman, Srini and Mahadevan, Priya and Yiakoumis, Y. and Sharma, P. and Banerjee, S. and McKeown, Nick, "Elastictree: Saving Energy in Data Center Networks," 7th USENIX NSDI, pp. 17–17, 2010. 82

[108] R. Maaloul, R. Taktak, L. Chaari, and B. Cousin, "Energy-Aware Routing in Carrier-Grade Ethernet Using SDN Approach," IEEE Transactions on Green Communications and Networking, vol. 2, no. 3, pp. 844–858, 2018. 82, 88, 89, 90, 91

[109] S. Sathyanarayana and M. Moh, "Joint Route-Server Load Balancing in Software Defined Networks Using Ant Colony Optimization," in Proc. of International Conference on High Performance Computing Simulation (HPCS), 2016, pp. 156–163. 82

[110] A. D. Stefano, G. Cammarata, G. Morana, and D. Zito, "A4SDN - Adaptive Alienated Ant Algorithm for Software-Defined Networking," in Proc. of 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2015, pp. 344–350. 82

[111] A. Fernández-Fernández, C. Cervelló-Pastor, and L. Ochoa-Aday, "A Multi-Objective Routing Strategy for QoS and Energy Awareness in Software-Defined Networks," IEEE Communications Letters, vol. 21, no. 11, pp. 2416–2419, 2017. 82

[112] H. Wang, Y. Li, D. Jin, P. Hui, and J. Wu, "Saving Energy in Partially Deployed Software Defined Networks," IEEE Transactions on Computers, vol. 65, no. 5, pp. 1578–1592, 2016. 82

[113] J. Ba, Y. Wang, X. Zhong, S. Feng, X. Qiu, and S. Guo, "An SDN energy saving method based on topology switch and rerouting," in NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium, 2018, pp. 1–5. 82

[114] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, "A Survey on Software-Defined Networking," IEEE Communications Surveys Tutorials, vol. 17, no. 1, pp. 27–51, 2015. 83

[115] R. Alvizu, G. Maier, N. Kukreja, A. Pattavina, R. Morro, A. Capello, and C. Cavazzoni, "Comprehensive survey on t-sdn: Software-defined networking for transport networks," IEEE Communications Surveys & Tutorials, vol. 19, no. 4, pp. 2232–2283, 2017. 83

[116] G. Parladori, G. Gasparini, F. Ruggi, A. D. Broi, V. Simone, and F. Nicassio, "YANG Modelling of Optical Nodes," in Proc. of 20th Italian National Conference on Photonic Technologies (Fotonica 2018), 2018, pp. 1–4. 83

[117] F. Kaup, S. Melnikowitsch, and D. Hausheer, "Measuring and Modeling the Power Consumption of OpenFlow Switches," in Proc. of 10th International Conference on Network and Service Management (CNSM) and Workshop, 2014, pp. 181–186. 84

[118] Y. Deng, Y. Chen, Y. Zhang, and S. Mahadevan, "Fuzzy dijkstra algorithm for shortest path problem under uncertain environment," Applied Soft Computing, vol. 12, no. 3, pp. 1231–1237, 2012. 86