Monitoring and Orchestration of Network Slices for 5G Networks

Ramón Pérez Hernández

A dissertation submitted by in partial fulfillment of the requirements for the degree of Doctor of Philosophy in

Telematics Engineering

Universidad Carlos III de Madrid

Tutor and Director:
Albert Banchs Roca

Leganés, January 2021

Este trabajo se ha realizado bajo la ayuda concedida por la **Comunidad de Madrid** en la **Convocatoria de 2017 de Ayudas para la Realización de Doctorados Industriales en la Comunidad de Madrid** (Orden 3109/2017, de 29 de agosto), con referencia **IND2017/TIC-7732**.

This work was partly funded by the **European Commission** under the **European Union's Horizon 2020 program - grant agreement number 815074 (5G EVE project)**. The Ph.D thesis solely reflects the views of the author. The Commission is not responsible for the contents of this Ph.D thesis or any use made thereof.

This thesis is distributed under license "Creative Commons **Attribution – Non Commercial – Non Derivatives**".

Los pequeños detalles
marcan la diferencia

Acknowledgements

En primer lugar, quiero expresar mi total agradecimiento a mi tutor, Albert Banchs, por iluminarme en todo momento en este largo camino para ir abriéndome paso en los temas e ideas que hemos ido discutiendo con el paso de los meses. También por introducirme en el mundo de la investigación y por permitirme el lujo de conocer a fantásticos profesionales dentro de la Universidad Carlos III de Madrid. Entre ellos, dar también mi especial agradecimiento a Pablo Serrano, Marco Gramaglia y Jaime García-Reinoso por sus consejos y su tiempo.

Agradecer también a todo el equipo de Telcaria Ideas S.L., con el que comencé este camino del Doctorado Industrial, la confianza depositada en mí. En Telcaria, he tenido la oportunidad de aprender numerosos conceptos, herramientas y tecnologías desde cero, y gracias a todo este trabajo recorrido, puedo decir que han sido unos años muy enriquecedores en lo profesional y lo personal; y, por supuesto, con ganas de que siga siendo así.

I would like to thank professors Gianluca Reali and Mauro Femminella, from the Università degli Studi di Perugia, for all their support (despite the difficult situation for everybody due to the COVID'19) during my Ph.D Visit in Perugia, extending also my gratitude to Matteo Pergolesi and Priscilla Benedetti for the joint work that we have been doing together during all the 2020, learning from each other and also helping between us for any problem arisen. And finally, I thank professor Ferdinando Treggiari for allowing me to stay in his lovely flat in Perugia's city center. It was such an amazing experience.

A todos mis amigos, tanto los más cercanos como los más lejanos, e incluso los que han aparecido por mi vida durante estos meses, por poner su granito de arena para que esta tesis saliese adelante con todos sus ánimos y fuerzas transmitidas. Pero, sobre todo, por iluminar mi vida y por estar siempre ahí, sin importar distancia, tiempo y momento.

A mi familia, que sufrió mi marcha de casa para empezar a estudiar Ingeniería de Telecomunicaciones, pero que vivió con enorme alegría todos los éxitos cosechados durante estos años de trabajo y esfuerzo donde la lucha no se negoció. Por todas las llamadas, todas las visitas, todos los buenos momentos vividos y que viviremos, y también esos momentos donde no pudimos hacer cosas porque tenía que sentarme a estudiar. Ahora, sí que sí, toca vivir.

Y, por qué no, gracias a mí mismo, por sobreponerme a todos los problemas, por creer en mis posibilidades y porque, finalmente, el resultado de todo el esfuerzo, de toda la ilusión y de todo el camino transcurrido, con momentos buenos y malos, ya está por fin terminado.

Published and Submitted Content

This Ph.D Thesis covers contributions from the following **published papers**, in which the author has directly participated in, also including the chapters in which the content is presented:

- M. Gramaglia, V. Sciancalepore, F. J. Fernandez-Maestro, **R. Perez**, P. Serrano, and A. Banchs, *"Experimenting with SRv6: a Tunneling Protocol supporting Network Slicing in 5G and beyond,"* in 2020 IEEE 25th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), 2020, pp. 1–6 [1].
  - Status: Published.
  - Role: writing some sections of the paper, redesign of all the figures and review of the final paper.
  - Chapter in which this material is included: Chapter 1.
  - Level of inclusion: partly included in the thesis.
- **R. Perez**, J. Garcia-Reinoso, A. Zabala, P. Serrano, and A. Banchs, *"A Monitoring Framework for Multi-Site 5G Platforms,"* in 2020 European Conference on Networks and Communications (EuCNC), 2020, pp. 52-56 [2].
  - Status: Published.
  - Role: leader of the paper, responsible for the design, implementation and testing of the platform.
  - Chapters in which this material is included: Chapters 2, 3 and 4.
  - Level of inclusion: wholly included in the thesis.
- D. Bega, M. Gramaglia, **R. Perez**, M. Fiore, A. Banchs, and X. Costa-Pérez, *"AI-based Autonomous Control, Management, and Orchestration in 5G: from Standards to Algorithms,"* IEEE Network, vol. 34, no. 6, pp. 14–20, 2020 [3].
  - Status: Published.
  - Role: contribution to the architecture design and its alignment with the standards.
  - Chapters in which this material is included: Chapters 2 and 4.
  - Level of inclusion: partly included in the thesis.
- M. Gramaglia, P. Serrano, A. Banchs, G. Garcia-Aviles, A. Garcia-Saavedra, and **R. Perez**, *"The case for serverless mobile networking,"* in 2020 IFIP Networking Conference (Networking), 2020, pp. 779-784 [4].
  - Status: Published.

  - Role: execution of experiments to validate the liquid scalability approach, review of the final paper.
  - Chapters in which this material is included: Chapters 2 and 7.
  - Level of inclusion: partly included in the thesis.

- W. Nakimuli, G. Landi, **R. Perez**, M. Pergolesi, M. Molla, C. Ntogkas, G. Garcia-Aviles, J. Garcia-Reinoso, M. Femminella, P. Serrano, F. Lombardo, J. Rodriguez, G. Reali, and S. Salsano, *"Automatic deployment, execution and analysis of 5G experiments using the 5G EVE platform,"* in 2020 IEEE 3rd 5G World Forum (5GWF), 2020, pp. 372-377 [5].

  - Status: Published.
  - Role: review of all the content related to the Monitoring platform, also participating in the tests reported on the paper, and review of the final paper.
  - Chapter in which this material is included: Chapter 3
  - Level of inclusion: partly included in the thesis.

Moreover, the following **papers**, which were **submitted for publication** and are still **under review**, are also part of this Ph.D thesis:

- **R. Perez**, J. Garcia-Reinoso, A. Zabala, P. Serrano, and A. Banchs, *"A Distributed Framework Based on Publish-Subscribe to Monitor Beyond 5G Networks,"* in EURASIP Journal on Wireless Communications and Networking, 2020 [6].

  - Status: Submitted for publication.
  - Role: leader of the paper, responsible for the design, implementation and testing of the platform.
  - Chapters in which this material is included: Chapters 2, 3 and 4.
  - Level of inclusion: wholly included in the thesis.

- **R. Perez**, A. Zabala, and A. Banchs, *"Alviu: An Intent-Based SD-WAN Orchestrator of Network Slices for Enterprise Networks,"* in 2021 IEEE 7th International Conference on Network Softwarization (NetSoft) (NetSoft 2021), 2021 [7].

  - Status: Submitted for publication.
  - Role: leader of the paper, responsible for the design, implementation and testing of the platform.
  - Chapters in which this material is included: Chapters 2, 5 and 6.
  - Level of inclusion: wholly included in the thesis.

And finally, from the **joint work** with the **Università degli Studi di Perugia** during the **Ph.D Visit** of the thesis author between September and November 2020, it is expected to produce at least **one paper (or two)** related to

the **serverless topic** (currently without title and list of authors confirmed, and that is the reason why they are not included in this list). Some of the expected content of **these papers** are presented in Chapters 2, 7 and 8.

Note that this **Ph.D thesis** has been **based** on the different **papers and work** referred above. Moreover, all the material taken from all the sources commented before that has been included in this thesis, it is indicated with an explicit reference.

Other Research Merits

It is also worth to describe **other related work**, apart from the papers written and commented in the Published and Submitted Content's section, and that mainly derives from the work done in **Telcaria Ideas S.L.**, aligned with the objectives of the **Industrial Ph.D.** These are the following:

- Participation in the **5G EVE project** [8] and related deliverables, being the **responsible** for the **Monitoring platform** presented in Chapter 3, and which is also extended in Chapters 4 (for its adaptation in beyond 5G networks), 7 and 8 (for its transformation to a serverless architecture).
- Design, development and testing of **Alviu's SD-WAN orchestrator** [9], a **commercial product** from **Telcaria Ideas S.L.** which is explained in detail in Chapters 5 and 6.

Resumen

El concepto de Network Slicing ha estado siempre ligado a la evolución de las arquitecturas de red, proporcionando la capacidad de soportar múltiples redes lógicas sobre la misma infraestructura. Esta tecnología resulta de capital importancia en el ámbito de las redes móviles de nueva generación, o redes 5G, en las que se pretende soportar un amplio ecosistema de tecnologías relativas a la virtualización de servidores y de redes, entre otros, siendo también potenciales herramientas para mejorar las funcionalidades del Network Slicing.

En este ámbito, este trabajo pretende abordar el estudio de estas arquitecturas basadas en Network Slices sobre redes 5G, en las que múltiples usuarios pueden hacer uso de la misma plataforma, requiriendo el cumplimiento de ciertos requisitos en cuanto a métricas de rendimiento de red, entre otras.

Para ello, el estudio se dividirá en tres grandes ámbitos, perteneciendo todos al ecosistema creado por las redes 5G: la monitorización efectiva de redes para recopilar métricas de red susceptivas de ser utilizadas para el aseguramiento de las slices de red, la orquestación de redes basadas en el paradigma SDN, y la virtualización avanzada de componentes mediante el uso de la tecnología serverless.

Dichas herramientas, en su conjunto, permiten el desarrollo de un sistema inteligente, utilizando mecanismos innovadores de virtualización de servidores, capaz de recabar métricas de rendimiento de la red para su aplicación posterior en mecanismos de orquestación de redes, con funciones que pueden ir desde mecanismos de encaminamiento de tráfico hasta la aplicación de políticas personalizadas de red, todas basadas en las métricas recolectadas.

Dicho sistema es el resultado último y principal vía de futuro de este trabajo, que se limitará al análisis en detalle de cada uno de los tres aspectos mencionados anteriormente por separado.

Palabras clave: monitorización, orquestación, virtualización, redes 5G, Network Slicing.

Abstract

The concept of Network Slicing has always been linked to the evolution of networking architectures, providing the ability to support multiple logical networks on the same infrastructure. This technology is of capital importance in the field of new generation mobile networks, or 5G networks, in which it is expected to support a wide ecosystem of technologies related to the virtualization of servers and networks, among others, being also potential tools for improving Network Slicing functionalities.

In this scope, this work aims at addressing the study of these architectures based on Network Slices over 5G networks, in which multiple users can make use of the same platform, requiring the fulfillment of certain requirements related to network performance metrics, among other.

In this way, the study will be divided into three main areas, all belonging to the ecosystem created by 5G networks: the effective monitoring of networks to collect network metrics that can be used for the assurance of network slices, the orchestration of networks based on the SDN paradigm, and the advanced virtualization of components through the use of the serverless technology.

These tools, as a whole, allow the development of an intelligent system, using innovative server virtualization mechanisms, capable of collecting network performance metrics for their later application in network orchestration mechanisms, with functions that can range from traffic routing mechanisms to the application of custom network policies, all based on the collected metrics.

The system aforementioned is the last and main future result of this work, which will be limited to the detailed analysis of each of the three aspects mentioned above separately.

Keywords: monitoring, orchestration, virtualization, 5G networks, Network Slicing.

Table of Contents

Acknowledgements i

Published and Submitted Content iii

Other Research Merits vii

Resumen ix

Abstract xi

Table of Contents xiii

List of Figures xvii

List of Tables xxi

List of Acronyms and Abbreviations xxiii

1. Introduction 1
- 1.1. 5G Networks Characterization . . . 2
- 1.2. Summary of Thesis Contributions . . . 3
- 1.3. Thesis Overview . . . 4

2. Related Work 7
- 2.1. Monitoring Architectures in 5G and Beyond 5G Networks . . . 7
- 2.2. Orchestration Solutions in SD-WAN and 5G Networks . . . 9
  - 2.2.1. Orchestration of SD-WAN Networks . . . 10
  - 2.2.2. Intent-Based Networking . . . 12
  - 2.2.3. Integration with Legacy Networks . . . 13
- 2.3. Virtualization Mechanisms in 5G Networks . . . 14
  - 2.3.1. Evolution of Cloud Computing . . . 14
  - 2.3.2. Evolution of Mobile Networking . . . 15
  - 2.3.3. Introducing the Serverless Paradigm . . . 16

Part 1. Monitoring of Network Slices 19

3. An Adaptable Monitoring Framework for 5G Environments 21
- 3.1. System Design . . . 22
  - 3.1.1. System Requirements . . . 22
  - 3.1.2. Proposed Architecture . . . 23
- 3.2. Implementation Based on the Publish-Subscribe Paradigm . . . 25
- 3.3. Performance Evaluation . . . 28
  - 3.3.1. System Assumptions . . . 29
  - 3.3.2. Testbed Setup . . . 30
  - 3.3.3. Preliminary Evaluation Process for a Single-Topic Experiment 31

3.3.4. Performance Impact Assessment for Simultaneous Multi-Topic Experiments . . . 32
3.4. Demonstration of the Platform in a Real Case Scenario . . . 35
3.4.1. Experiment Design, Definition and Preparation . . . 36
3.4.2. Experiment Execution and Results Analysis . . . 36
3.5. Summary . . . 38

4. Towards a Distributed Monitoring Framework for Beyond 5G Networks 39
4.1. New System Requirements . . . 40
4.2. Revision of the System Design and Implementation . . . 42
4.2.1. Adaptation of the Architecture . . . 42
4.2.2. Extensions to the Architecture . . . 43
4.2.3. Implementation Update . . . 45
4.3. Performance Evaluation . . . 46
4.3.1. Testbed Setup . . . 46
4.3.2. Singe-Broker Experiments . . . 47
4.3.3. System Scalability Validation . . . 49
4.3.4. Multi-Broker Experiments . . . 50
4.4. Summary . . . 53

Part 2. Orchestration of 5G Transport Networks 55

5. Novel Network Orchestration Techniques Based on SDN and Intent-Based Capabilities 57
5.1. Network Infrastructure Model . . . 59
5.2. Alviu Orchestrator's Architecture . . . 60
5.2.1. Intent-Based Networking Characterization . . . 62
5.2.2. Alviu Specification . . . 64
5.3. Intent States Management . . . 69
5.3.1. States Specification . . . 69
5.3.2. States Workflow . . . 70
5.4. Performance Evaluation . . . 73
5.4.1. Testbed Setup . . . 73
5.4.2. Deployment Time Evaluation . . . 75
5.5. Summary . . . 76

6. Evaluation and Demonstration of Intent-Based Orchestration Capabilities in Real Scenarios 77
6.1. Interconnection with External IGP Domains . . . 78
6.1.1. Use Case Overview . . . 78
6.1.2. Integration of Quagga in OVS-based Switches . . . 79
6.1.3. System Workflow . . . 82
6.2. Load Balancing with Dual Link Between SDN Switches . . . 88
6.2.1. Use Case Overview . . . 88
6.2.2. Updates Needed in OVS-based Switches . . . 89
6.2.3. System Workflow . . . 90
6.3. Proof of Concept . . . 93

6.3.1. Testbed Setup . . . 93
6.3.2. Basic Connectivity Between SDN Switches . . . 94
6.3.3. Interconnection with External Domains . . . 94
6.3.4. Testing Load Balancing Capabilities . . . 96
6.3.5. Testing Network Slicing Features with Firewall and QoS Policies . . . 97
6.4. Summary . . . 98

Part 3. New Virtualization Techniques **99**

7. Integration of the Serverless Paradigm within 5G Networks **101**
7.1. Serverless Mobile Architectures' Overview . . . 102
7.1.1. Concept . . . 102
7.1.2. Advantages . . . 104
7.1.3. Challenges to Address . . . 106
7.2. Introducing Serverless Techniques in the Monitoring Platform . . . 107
7.2.1. Problems Found in the Legacy Architecture . . . 107
7.2.2. Transformation from Legacy to Microservices Architecture . 108
7.2.3. Transformation from Microservices to Serverless Architecture 114
7.3. Workflow's Validation . . . 121
7.4. Summary . . . 122

8. Evaluation of the Monitoring Platform Deployment Based on Different Virtualization Techniques **123**
8.1. Testbed Setup . . . 124
8.1.1. Servers' Description . . . 124
8.1.2. Testbed Specification for each Virtualization Technique . . . 125
8.2. Test Cases' Description . . . 128
8.3. Single Server Performance Evaluation . . . 129
8.4. Horizontal Scaling Performance Evaluation . . . 135
8.4.1. Update of the Testbed Specification . . . 135
8.4.2. Results Obtained . . . 136
8.5. Summary . . . 138

9. Conclusions and Future Work **139**
9.1. Conclusions . . . 139
9.2. Future Work . . . 140

Appendices **145**

A. Examples of Descriptors **145**
A.1. General Configuration Descriptor . . . 145
A.2. Branch Intent Descriptor . . . 147
A.3. Connection Intent Descriptor . . . 152
A.4. Policy Intent Descriptor . . . 153

References **157**

List of Figures

1. Monitoring and orchestration solutions applied over the 5G network data plane. . . . 2
2. Major transitions in the adoption of softwarization. . . . 15
3. Monitoring metrics architecture. . . . 23
4. Component chain that implements the general Monitoring metrics architecture in the 5G EVE platform. . . . 25
5. Data Collection Manager architecture. . . . 26
6. Data Collection and Storage-Data Visualization architecture. . . . 28
7. Calculations made for system setting. . . . 30
8. Testbed architecture. . . . 31
9. Batch write latency distribution in one experiment with 20 topics. . 33
10. CPU consumption and batch write latency evolution for 100 B data traffic in different experiments, modifying a different design parameter in each case whereas the other one remains fixed. . . . 34
11. CPU consumption and I/O message rate evolution for 100 B data traffic in different experiments, modifying both number of experiments and total throughput in all cases. . . . 35
12. Blueprints used for collecting the information related to monitoring data and generaetion of the messages sent to the DCM. . . . 37
13. Dashboards that present the evolution over time of a metric and a KPI that belongs to a given experiment. . . . 38
14. Monitoring architecture, highlighting the components to be deployed in the Cloud and in the Edge. . . . 42
15. A network data analytics framework proposal with AI-driven capabilities. . . . 44
16. Enhanced network analytics framework with the integration of the Monitoring framework for data collection purposes. . . . 45
17. Testbed architecture. . . . 46
18. Effect of saturation in performance parameters when limiting *Kafka* vCPU allocated in different experiments. . . . 49
19. Evolution of the I/O message rate related to 100 B data traffic in one experiment when vertical scaling mechanisms are enabled. . . . 50
20. Evolution of the three types of latency in multi-broker experiments. 51
21. Evolution of Data Collection Manager CPU consumption and I/O message rate in multi-broker experiments. . . . 52
22. Network infrastructure model implemented by Alviu. . . . 59
23. Alviu's high-level architecture. . . . 61
24. Mapping between the type of descriptors that can be defined and the configuration applied by Alviu in the managed network infrastructure. 63
25. Alviu architecture, including the building blocks that compose the SDN Orchestrator and the SDN Controller. . . . 64
26. General intent states present during the intent operation. . . . 69
27. Branch Intent installation operation workflow. . . . 71
28. Branch Intent withdrawal operation workflow. . . . 72
29. Connection Intent installation operation workflow. . . . 72

30. Connection Intent withdrawal operation workflow. 73
31. CI environment architecture, including the technologies used in each component. 74
32. Star topology evaluated in the performance evaluation analysis. . . 75
33. Evolution of the deployment time, in seconds, varying the number of branches deployed in the star topology. 76
34. Example of topology with interconnection with external domains. . 78
35. Typical internal architecture of a WAN-type SDN switch. 80
36. Connection of Quagga to OVS to allow the exchange of information with external domains. 81
37. IGP connection establishment workflow. 82
38. Introduction of Next Hop and Prefix entities in the data model, showing how all the entities related to IGP match in a real scenario. 84
39. Prefix learning workflow. 85
40. Prefix deletion workflow. 87
41. Backup prefix activation workflow. 88
42. Practical scenario with switches connected by more than one link. . 89
43. Update of the OVS-based switches to include the components needed for the load balancing use case. 90
44. Example of traffic flows captured in a scenario with dual link between switches. 90
45. Load balancing workflow. 91
46. Testbed built with the Continuous Integration environment to do the proof of concept. 93
47. Flows related to basic connectivity in the SDN switch of branch X. 94
48. Flows that includes the interconnection with external domains in the SDN switch of branch X. 95
49. Flows that includes the interconnection with external domains in the SDN switch of branch Y. 96
50. Update of the flows related to the interconnection with external domains in the SDN switch of branch X. 96
51. Load Balancing Overlay flows in the switch from branch X. 97
52. Load Balancing Overlay flows in the switch from branch Y. 97
53. Example of firewall rules installed in the switch of branch Y. 97
54. Example of QoS rules installed in the switch of branch Y. 98
55. Mobile network architecture evolution. 103
56. Representation of the liquid scalability. 105
57. Data Collection Manager microservices architecture. 109
58. Data Collection and Storage-Data Visualization microservices architecture. 110
59. Topic creation workflow for the Monitoring microservices architecture. 110
60. Topic deletion workflow for the Monitoring microservices architecture. 113
61. Serverless architecture of the Monitoring platform. 115
62. Topic creation workflow for the Monitoring serverless architecture. . 116
63. Topic deletion workflow for the Monitoring serverless architecture. . 119
64. Testbed for validating the Monitoring serverless architecture. 121
65. Physical testbed for the evaluation of the Monitoring platform. . . . 126

66. Virtual testbed for the evaluation of the Monitoring platform. . . . 126
67. *Docker* (*runc*) testbed for the evaluation of the Monitoring platform. 127
68. *Kubernetes* (*containerd*) testbed for the evaluation of the Monitoring platform. . . . 127
69. *Kata (Firecracker/QEMU)* testbed for the evaluation of the Monitoring platform. . . . 128
70. CPU consumption evolution for all the testbeds of the Monitoring platform (100 B messages). . . . 130
71. (Top) Batch write latency evolution for all the testbeds of the Monitoring platform, (bottom) also detailing the results for the testbeds with lower values (100 B messages). . . . 131
72. I/O message rate evolution for all the testbeds of the Monitoring platform (100 B messages). . . . 132
73. CPU consumption evolution for the *Kata* testbeds with a workload lower than 1 experiment (100 B messages). . . . 133
74. Batch write latency evolution for the *Kata* testbeds with a workload lower than 1 experiment (100 B messages). . . . 134
75. I/O message rate evolution for the *Kata* testbeds with a workload lower than 1 experiment (100 B messages). . . . 135
76. CPU consumption evolution for the *Kubernetes* testbed with two *Kafka* brokers (100 B messages). . . . 135
77. CPU consumption evolution for the *Kubernetes* testbed with two *Kafka* brokers (100 B messages). . . . 136
78. Batch write latency evolution for the *Kubernetes* testbed with two *Kafka* brokers (100 B messages). . . . 137
79. I/O message rate evolution for the *Kubernetes* testbed with two *Kafka* brokers (100 B messages). . . . 137

List of Tables

1. Comparison of commercial SD-WAN products, also positioning Alviu in the current state of the art. Information extracted from [31] and [32]. 11
2. Specification of the servers used in the testbed. 124

List of Acronyms and Abbreviations

2G Second Generation.

3GPP 3rd Generation Partnership Project.

4G Fourth Generation.

5G Fifth Generation.

5Gr-VoMS 5Gr-Vertical-oriented Monitoring System.

6G Sixth Generation.

ACK Acknowledgement.

ACL Access Control List.

AI Artificial Intelligence.

AI-LTF Artificial Intelligence-Long-Term Forecast.

AI-MTF Artificial Intelligence-Mid-Term Forecast.

AI-STF Artificial Intelligence-Short-Term Forecast.

API Application Programming Interface.

ARP Address Resolution Protocol.

AWS Amazon Web Services.

B Byte.

BGP Border Gateway Protocol.

C-RAN Cloud Radio Access Network.

CI Continuous Integration.

CIDR Classless Inter-Domain Routing.

CNF Cloud-Native Network Function.

CPU Central Processing Unit.

CtxB Context Blueprint.

DCM Data Collection Manager.

DCS Data Collection and Storage.

DevOps Development and Operations.

DHCP Dynamic Host Configuration Protocol.

DN Data Network.

DNS Domain Name System.

DPDK Data Plane Development Kit.

DSCP Differentiated Services Code Point.

DV Data Visualization.

eBPF extended Berkeley Packet Filter.

ELK Elasticsearch Logstash Kibana.

ELM Experiment Lifecycle Manager.

eMBB Enhanced Mobile Broadband.

ENI Experiential Networked Intelligence.

ESP Encapsulating Security Payload.

ETSI European Telecommunications Standards Institute.

ExpB ExperimentBlueprint.

FaaS Function as a Service.

GB Gigabyte.

Gbps Gigabits per second.

GHz Gigahertz.

gNB Next Generation Node B.

gNB-CU Next Generation Node B-Central Unit.

gNB-DU Next Generation Node B-Distributed Unit.

GRE Generic Routing Encapsulation.

GUI Graphical User Interface.

HQ Headquarters.

HTTP Hypertext Transfer Protocol.

I/O Input/Output.

IBM International Business Machines.

IBN Intent-Based Networking.

IGP Interior Gateway Protocol.

IIoT Industrial Internet of Things.

IoT Internet of Things.

IP Internet Protocol.

IPsec Internet Protocol security.

ISG Industry Specification Groups.

ISP Internet Service Provider.

IT Information Technology.

JSON JavaScript Object Notation.

KB Kilobyte.

Kbps Kilobits per second.

KPI Key Performance Indicator.

KVM Kernel-based Virtual Machine.

LAN Local Area Network.

LCM Life Cycle Manager.

LTS Long Term Support.

LXC LinuX Containers.

MAC Media Access Control.

MANO Management and Orchestration.

MB Megabyte.

Mbps Megabits per second.

MDAF Management Data Analytics Function.

MEC Multi-access Edge Computing.

MEF Metrics Extractor Function.

ML Machine Learning.

mMTC Massive Machine-Type Communications.

MPLS Multiprotocol Label Switching.

ms millisecond.

MT/s Million Transfers per second.

NAT Network Address Translation.

NBI Northbound Interface.

NEMO Network Modeling.

NFV Network Function Virtualization.

NOP Network Operator.

NPN Non-Public Network.

NSaaS Network Slice as a Service.

NTP Network Time Protocol.

NWDAF Network Data Analytics Function.

O-RAN Open Radio Access Network.

ONAP Open Network Automation Platform.

ONF Open Networking Foundation.

ONOS Open Network Operating System.

OSPF Open Shortest Path First.

OVS Open vSwitch.

P-GW Packet Gateway.

PDU Protocol Data Unit.

PNF Physical Network Function.

PoP Point of Presence.

QEMU Quick EMUlator.

QoS Quality of Service.

RAM Random Access Memory.

RAN Radio Access Network.

RAV Results Analysis and Validation.

REST REpresentational State Transfer.

RNIB Radio Network Information Base.

s seconds.

S-GW Serving Gateway.

S-NSSAI Single Network Slice Selection Assistance Information.

SaaS Software as a Service.

SBA Service Based Architecture.

SBI Southbound Interface.

SD-WAN Software-Defined Wide Area Network.

SDN Software-Defined Networking.

SDO Standard Development Organization.

SLA Service Level Agreement.

SMF Session Management Function.

SR-IOV Single-Root Input/Output Virtualization.

SRv6 Segment Routing over IPv6.

SSH Secure SHell.

SUT System Under Test.

TB Terabyte.

TCB Test Case Blueprint.

TCP Transmission Control Protocol.

TP TRansit Point .

UDP User Datagram Protocol.

UE User Equipment.

UFW Uncomplicated Firewall.

UPF User Plane Function.

URL Uniform Resource Locator.

URLLC Ultra-Reliable and Low Latency Communications.

UUID Universally Unique IDentifier.

vCPU virtual CPU.

veth virtual Ethernet.

VM Virtual Machine.

VNF Virtual Network Function.

VSB Vertical Service Blueprint.

WAN Wide Area Network.

WE West-East.

ZSM Zero touch network & Service Management.

1 Introduction

The **success** of the upcoming **Fifth Generation** of **mobile networks (5G)** and **beyond** is heavily **tied** with the implementation of the **Network Slicing** paradigm [1]. Strongly supported by the **virtualization** and **programmability** concepts, this represents a turning point that enables the **capability** of **flexibly assigning virtual instances** of a **mobile network** to diverse **services**. In this way, **Network Operators** (NOPs) are expected to (*i*) **increase** the **revenues** obtained from their **infrastructure** by also (*ii*) achieving an **overall higher utilization** due to **resource sharing**.

However, this higher flexibility and increased revenues might come at a **price**. **Technical challenges** have to be solved while **deploying Network Slicing** along all **network domains**, such as **Radio Access, Transport or Core Networks**. Such domains need a **simultaneous and efficient interaction** to properly provide **Service Level Agreement (SLA) guarantees**, which is accomplished by a novel architectural block: the **Management and Orchestration (MANO)**, able to **control, monitor and trigger actions** onto each **network (virtual) function**. As a result, networks supporting slicing require **advanced orchestration solutions** that have attracted interest from both **industry** and **academia** showing **advantages and drawbacks** in current deployments.

And what is more, these **MANO platforms** need the **joint interaction** with **systems and mechanisms** that provide the **data** needed by the orchestration technologies for their own **decision-making process**. Among the tools that can be identified in this vast ecosystem, **efficient monitoring platforms** are having now its momentum, focusing on **heterogeneous sources of network** or **compute metrics** from which the **MANO** components can extract useful **information** about the **status of the network** for the consequent **actions** to be performed.

These **two topics** aforementioned (*i.e.* **monitoring and orchestration**), together with the **research** on **new virtualization techniques** for implementing **applications and services** adapted to the **current and future trends** on **mobile network architectures**, are the **main topics** covered in this thesis.

Next, some **general concepts** will be explained before providing a deeper analysis on each topic.

1.1. 5G Networks Characterization

The **study** done in this thesis analyzes with depth **three different parts** of a given **network architecture**, focusing on (*i*) mechanisms to **gather network metrics** from a given **5G infrastructure and application components**, in order to be **provided** to other **elements** that may be **interested in using these values**, (*ii*) **orchestration** solutions for **transport network**, *i.e.* the User Plane Functions and the Data Networks deployed in the 5G Core Network, using the **new trends** in terms of **network softwarization**, and (*iii*) the research on **virtualization mechanisms** to **facilitate** the **deployment** of **network functions** in a given architecture.

The **application** of these **trends** in a **general scenario** based on 5G networks can be observed in Figure 1, which depicts the **key components** implied on the **5G network data plane** with the **3GPP-related reference points**. This way, in a typical **workflow**, the **User Equipment (UE)** establishes a **data session** and gets **assigned** the corresponding **User Plane Function (UPF)** in the **Core Network**, being responsible for **routing and forwarding** the **traffic back and forth** between the **UE** and a **Data Network (DN)**. The **UPF** selects the **appropriate transport network** for the **user traffic** by means of **N4** reference point, which provides an **interface** with the **Session Management Function (SMF)**. This interface enables selecting a **Network Instance ID** based on the **S-NSSAI** of the **PDU session**. Considering the actual transmission of user data traffic, this arrives from the **gNB** to the **UPF** via the **N3 interface**, while the **UPF** is connected to the **DN** via the **N6 interface**. Finally, the interface **N9** is defined in case there is **communication between UPFs** (*e.g.* from an intermediate to an anchor UPF) [1].

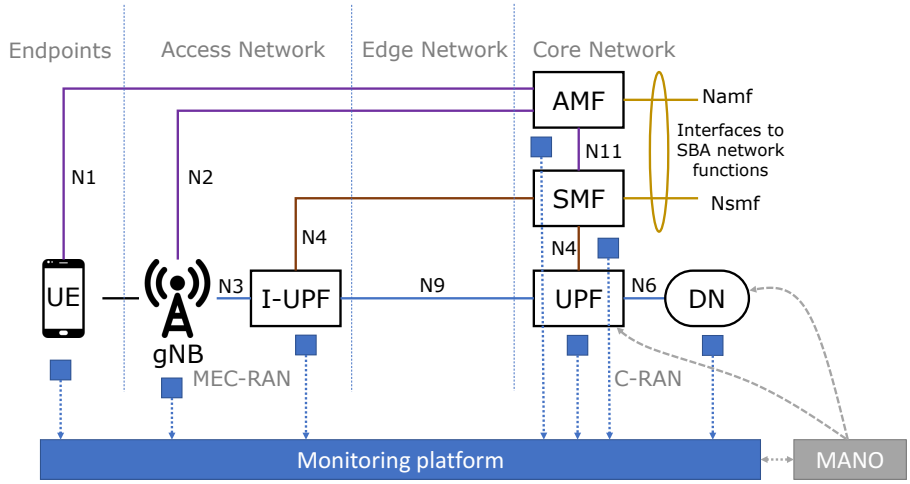

Figure 1: Monitoring and orchestration solutions applied over the 5G network data plane.

In this scope, each **component of the platform** may have specific **processes running** on them to **gather metrics** about **different aspects of the network**, depending on the monitored components involved. These processes may be **proprietary software** implemented by the **vendors**, or what is more interesting, they may be **small pieces of standard software**, probably implemented with **lightweight virtualization tools**, that can be **adapted to different components** to extract a specific type of metric. In any case, these elements **feed a platform** in charge of **managing the monitoring** of the **collected data**, also offering them to **interested elements** of the network that may require their usage.

The **main example** of this **last topic** is the **MANO component** presented on Figure 1, which is **directly connected to the Monitoring platform**, using the **metrics** to control the **lifecycle** of, for example, the networking components used in the Core Network.

This introduction has been done to clearly **identify** the **parts** of the **5G network** on which this thesis focuses. There are **more technologies or concepts** that **may be applied** in this kind of networks, *e.g.* the analysis of **tunneling protocols** for user data traffic in 5G networks, as analyzed by the author in [1] for the case of SRv6, but they are **out of the scope** of this thesis.

1.2. Summary of Thesis Contributions

To address the **three topics** mentioned above, this thesis has been distributed into **three main parts**, each of them containing **two subchapters** for delving into each specific subject.

First of all, in terms of **monitoring processes** on **5G networks**, a complete **Monitoring architecture** will be **proposed**, starting with its general **design**, covering **multi-site and multi-stakeholder scenarios** with a **flexible solution** based on a **multi-brokering architecture**, which allows to **distribute the metrics** captured on each site with **easy mechanisms**. In particular, its **implementation** is based on the **publish-subscribe paradigm**, in which this Monitoring framework has been **implemented**. This contribution is mainly based on the work done in [2], which presents this **design and implementation**, together with a **complete performance evaluation** process to check its suitability for the scenarios in which it is expected to be deployed. Moreover, its **validation** in a **complete test executed** on the **5G EVE platform** is also reported in [5].

As an **improvement of this platform**, [6] proposes the **implementation** of this **Monitoring platform** in **Edge environments**, having in mind the future **Beyond 5G scenarios** that are currently under study and evaluation in the research community. This deployment has also been **evaluated** in terms of **performance**, checking again that the system is **able to manage** this kind of **unpredictable scenarios**. Moreover, the **platform** can be also **extended** with other **modules** to provide **enhanced capabilities**. This is the case of the inclusion of the **Data Analytics framework** proposed in [3].

Secondly, it comes the **orchestration of transport networks**, focusing on the description of **Alviu's SD-WAN orchestrator**, a **commercial product**

that belongs to **Telcaria Ideas S.L.** [9] and that has been **improved** during this **Ph.D. thesis**, covering the objective of the **Industrial Ph.D. Mention** as a result. The **base** of this work, which has been reported in [7], consist of the **description of the platform** in terms of **design** and **specification of use cases**, also including some **performance evaluation tests** to measure the **capability** of the platform to **deal with a given workload** in the system. Note that the **source code** of the orchestrator is **not going to be released**, as it belongs to Telcaria Ideas S.L. and it is under a **confidentiality agreement**.

Finally, regarding the **new virtualization techniques** topic, the **serverless paradigm** is studied in depth, starting with the **introduction** commented on [4], and continuing with the **transformation** of the **Monitoring platform** already explained to a **serverless architecture**, explaining the **workflows** to be **followed** and **comparing** it with other **architectures** (*e.g.* a microservices-based one). After this, it is performed a complete **performance evaluation**, following the **same procedures** used for the testing of the **Monitoring platform**, in order to **compare** the **serverless technologies** used with **other virtualization and containerization technologies**. All this work has been done during the **Ph.D. International Visit from September to November 2020**. Currently, it is **about to start** with the **writing of some papers** including these results.

1.3. Thesis Overview

The **content** presented in Section 1.2 is **presented** in the following way:

- First of all, the **state of the art** of all the **three topics** aforementioned is presented in Chapter 2.
- Starting with the **monitoring topic**, Chapter 3 presents the **Monitoring platform** in terms of **design, implementation and preliminary performance evaluation** and **application in a real experiment** executed on the **5G EVE platform**.
- In Chapter 4, the **Monitoring platform** is **extended** to deal with **Edge and Beyond 5G environments**, also making some references to the possibility of **extending** the platform to allow the **introduction** of **enhanced features**, such as **AI and Data Analytics**. A **new performance evaluation** is also done, also **validating some preliminary scalability mechanisms** based on **vertical scaling**.
- Continuing with the **orchestration topic**, Chapter 5 presents **Alviu's architecture**, with a full **description** of the **Intent-Based Networking approach** selected to implement this platform, and also performing a **performance evaluation** process to evaluate the **deployment time of the intents**.
- To conclude Alviu's evaluation, Chapter 6 **analyzes and validates two particular use cases** in depth, which are related to the **interconnection** with **external IGP domains** and the **load balancing with dual link** between **SDN switches**.

- Regarding the **new virtualization techniques topic**, Chapter 7 presents the **serverless paradigm in general terms**, then **applying** it to **transform the Monitoring platform** to a **serverless-based platform**. The **workflows** for the **new architecture** are also detailed, presenting a **testbed** that allows to **validate** these **workflows**.
- To conclude with the serverless part, Chapter 8 performs a **complete performance evaluation** process of different **testbeds**, in which **each** of them **uses a different virtualization technique**. This is done to **compare** the **performance** obtained in the **serverless** scenario with the **other cases**. Furthermore, **horizontal scaling capabilities** are also **evaluated**.
- Finally, Chapter 9 presents the **conclusions and future work** for **each topic**.

2
Related Work

In order to **position** the **research work** related to each **topic** studied in this Thesis within the state of the art, the **related work** for these topics will be analyzed in this Chapter. To do this, the state of the art of each topic will be presented **separately**, in different sections, following this **order**:

- First of all, in Section 2.1, the related work for the **Monitoring topic** in both **5G and Beyond 5G networks** will be studied.
- After this, Section 2.2 will focus on **Orchestration**, delving into different **topics of interest** related to 5G network orchestration solutions, such as **intent-based networking**.
- Finally, Section 2.3 will present the state of the art of **novel Virtualization technologies**, focusing on the **serverless paradigm**.

2.1. Monitoring Architectures in 5G and Beyond 5G Networks

Triggered by the complexity and novelty of **5G**, several **research initiatives** have started to gather an **understanding** of the envisioned **features** of these types of networks, in order to be **applied** for effective **Monitoring solutions**, among others. In this way, the **related work** in terms of **Monitoring platforms** designed and provisioned for **5G** (and also **Beyond 5G**) **networks** can be grouped in **three main categories**, according to the **environment** in which the solution presented in this Thesis has been involved.

First of all, the proposed Monitoring platform has been designed and implemented within the scope of (*i*) **European projects** related to the **research** on **5G networks**; being more precise, the **5G EVE project** [10], which aims at deploying a **validation 5G multi-site platform**, involving four main facilities located in Spain, Italy, France, and Greece, where verticals and other projects can execute **extensive trials** [11]. In this context, the **Monitoring solution** has to **collect** all the **metrics** generated by the different elements involved in an

experiment to show their **evolution** over time to the experimenter, and to feed such data to **KPI validation tools** to confirm the **achievement of the KPIs**, or also **enabling new workflows** like the optimization of network performance.

While a number of other 5G projects (European and International) have **addressed monitoring functionalities**, **limited work** in this context have addressed the **publish-subscribe paradigm**, a messaging pattern which can be commonly found in the communication between distributed systems. This paradigm was, in fact, the **option selected by 5G EVE** to implement its Monitoring architecture, and this idea was **also considered** in the **5GROWTH project**, integrating some ideas and concepts present in the 5G EVE Monitoring platform with the so-called **Vertical-oriented Monitoring System (5Gr-VoMS)** [12], which is an **extension** of the Monitoring solution already proposed in the **5G-TRANSFORMER project** [13].

Another present **context** in these environments is (*ii*) **standardization**. In order to integrate monitoring and data collection features in the **mobile network architecture**, **3GPP** and other **SDOs** are working in **data analytics frameworks** that take advantage of the collection of monitoring data related to the network infrastructure in order to enable the **autonomous and efficient control, management and orchestration** of **mobile networks**. In this working line, **3GPP** defined the **Network Data Analytics Function (NWDAF)** [14] and the **Management Data Analytics Function (MDAF)** [15] for **5G networks**.

Other organizations, such as the **O-RAN alliance**, also contemplates similar components in their architectures [16], and **ETSI** has also defined comparable assisting elements within the **Industry Specification Groups (ISGs)** on **Experiential Networked Intelligence (ENI)** and **Zero touch network & Service Management (ZSM)** [17]. Moreover, **open-source initiatives** such as **ONAP** [18] are also including **data analytics** into their architecture. All these ongoing efforts are, however, at an **early stage**, so that the integration of the Monitoring architecture presented in this Thesis may be useful to **steer the work** of these **initiatives**, as proposed in Section 4.2.2.

And finally, moving to (*iii*) **Beyond 5G networks**, requiring **flexible scenarios** that may be probably oriented to **Edge environments**, there are already several proposals that include the definition of a **publish-subscribe mechanism** to **distribute data** between different **entities** in **Edge-based deployments**. This is the case of [19] or [20], although they are mostly focused on **IoT and pure Edge platforms**, **not including 5G** communications. There are also other proposals **not related to** the **publish-subscribe system**, such as [21], which analyzes the **optimal placement and scaling** of **monitoring functions** in **Multi-access Edge Computing (MEC)** environments, but it does not consider **multi-site** nor **multi-stakeholder scenarios**, which is a feature that characterizes the solution presented in this Thesis.

In summary, while **substantial work** has been conducted to design **publish-subscribe platforms** in **distributed systems**, and to devise **Monitoring solutions specific for Beyond 5G systems**, the **key novelty** of the approach proposed in this Thesis is to bring the **publish-subscribe paradigm** into a **Beyond 5G Monitoring platform**, and to **implement and evaluate experimentally** the performance of the platform devised.

2.2. Orchestration Solutions in SD-WAN and 5G Networks

The **management and operation** of **traditional mobile transport networks** have been following the same trend as **Wide Area Networks (WAN)**, both being **limited** by two important **factors**: (*i*) **cost** and (*ii*) **flexibility**. Firstly, legacy architectures are built using **expensive** and **specialized vendor equipment**, meaning that they are **costly** to deploy and to maintain. Secondly, as each component of the network equipment (*e.g.* routers, firewalls, etc.) bases its functionality mainly on **hardware**, in conjunction with its own embedded control software, they are forced to take **control decisions** based only on **local information**, even though these decisions affect globally to the whole network.

Consequently, it is **difficult** to perform **global changes** in the **network configuration** accurately, fast and dynamically, thus making legacy architectures **rigid and static**. Furthermore, the flexibility is also **limited** in terms of adding **new functionalities** to the network deployments, since they are limited to the capabilities provided by specialized network equipment. Then, performing upgrades or acquiring new equipment that performs the desired functionality entails an additional and significant cost.

The **advances in network virtualization** over the past decade have tried to overcome these limitations. This process resulted in the emergence of **Software-Defined Networking (SDN)**, a technology which **decouples** the **network's control logic** from the **underlying network hardware** in charge of forwarding the traffic, **centralizing** the **control logic** in a **software-based controller entity** [22]. As a result, its implementation is meant to be a potential **enabler** to facilitate the **automation of network configurations** and, eventually, fully program the network [23].

In this way, the **introduction of SDN in mobile networks** has been considered one of the **key technologies** for the development of automated, software-based 5G networks, together with other enablers such as NFV, Machine Learning or Big Data [24]. Specifically, the ***plasticity*** that SDN can offer for the control and management of 5G networks allows, among other features, to **model** both the **control and data plane** according to the **requirements** of specific **applications and devices** connected to the mobile network [25], thus being a facilitator for enabling the **mobile network programmability**.

In fact, SDN has made its appearance into different **5G deployments** related to the research of this integration between 5G and SDN. This is the case of some solutions proposed in different **H2020 European projects**; such as **5G-Crosshaul** [26], which proposes an adaptive, flexible and software-based architecture for 5G transport networks **integrating multi-technology fronthaul and backhaul** segments, leveraging on **SDN and NFV** for this purpose, or **SELFNET** [27], also integrating **SDN and NFV** in order to enable a **fully autonomic and intelligent network management framework** for 5G networks.

In the case of the **implementation of SDN in WAN architectures** with the so-called **SD-WAN technology**, the truth is that it is becoming **increasingly used**. According to [28], practically all the global IT leaders have

already deployed SD-WAN or expect to deploy it within 24 months to automate the network infrastructure, remarking **network automation, SDN and intent-based networking** as the technologies that will have the **most impact** on networking in the following years. Furthermore, there are even proposals to **merge 5G and SD-WAN features** into single, commercial solutions to bring better control to both the SD-WAN software and the cellular network. In this line, the **partnerships between AT&T and VMware**, or **Verizon and Cisco**, both in United States, expects to achieve this ambitious goal [29].

Next lines will be focused on **SD-WAN**, as it is the basis used for building **Alviu**, the orchestration solution proposed in this Thesis. In particular, this section will focus on the following **state-of-the-art topics**, which are included in Alviu's design and implementation: **SD-WAN orchestration** (which generalizes the terms of network automation and SDN), **intent-based networking** and **integration with legacy networks**.

2.2.1. Orchestration of SD-WAN Networks

The **new generation of networks**, presented as **datacenters** of varying size which contain, among other elements, **virtualized networking devices** or **networking functions** that are **decoupled** from the **hardware**, require a **network orchestration** entity to **control and manage** this set of **hardware** and **software** equipment and, eventually, to **enable network automation**.

Currently, the **SDN market** is in **high demand** but fairly **difficult to predict**, due to the **costs** involved in making the move to this new architecture due to the **difficulty** in **integrating SDN** into the **existing networks**, even having the ambition of covering the control of peripheral devices separated from the core network, or the novelty of the technology [30]. To overcome these issues, the different SD-WAN solutions that have been appearing in the last years have tried to **target specific features** of the network, thus creating a very wide catalog of solutions, depending on users' needs.

Furthermore, the **stakeholders** participating in this market have also **changed** in the last decade, due to the growth and proliferation of **Cloud Computing** services. In this manner, preliminary SD-WAN solutions were offered mainly by specialized **service providers**, but nowadays there is a good distribution between these providers, **companies offering datacenter services**, **Cloud providers** and, to a lesser extent, **companies' access networks**.

To summarize the **heterogeneity** of **SD-WAN solutions** available in today's market, Table 1 presents some of the **most important SD-WAN platforms**, according to the last magic quadrant for WAN edge infrastructure made by Gartner [31], with their **main characteristics** [32], reflecting the idea that **each solution** tries to focus on **particular aspects of the network** to be eventually controlled and orchestrated, and **not finding a solution** that covers **all the features** of the network. In this comparison, Alviu is also positioned in the state of the art, checking that it is aligned with the current trends in the market.

Table 1: Comparison of commercial SD-WAN products, also positioning Alviu in the current state of the art. Information extracted from [31] and [32].

SD-WAN product	Gartner status	Orientation	WAN architecture	Form factor	Firewall	WAN optimization	Appl. path selection
VeloCloud	Leaders	Software	100+ Global PoP gateways	Physical, virtual, Cloud	Basic (advanced via partners)	No	Yes
Silver Peak	Leaders	Software	Edge based	Physical, virtual, Cloud	Basic (advanced via partners)	Yes	No
Fortinet	Challengers	Hardware	Edge based	Physical	Advanced	Yes (in some models)	No
Cisco Meraki	Challengers	Hardware	Edge based	Physical, Cloud	Advanced	Limited	No
Cisco Viptela	Challengers	Hardware	Edge based	Physical, virtual, Cloud	Basic (Viptela hw.), advanced (Cisco hw.)	Yes (Cisco hw.)	Yes (Cisco hw.)
Citrix	Challengers	Hardware	Edge based	Physical, virtual, Cloud	Advanced	Yes (in some models)	Yes
CloudGenix	Visionaries	Hardware	Edge based	Physical, virtual, Cloud	Basic (advanced via partners)	No	Yes
Aryaka	Visionaries	Software	25 Global PoP backbone	Physical	Basic (advanced via partners)	Yes	Yes
Alviu	**Not ranked**	**Software**	**Edge based**	**Physical, virtual, Cloud**	**Basic**	**No**	**Yes**

In general terms, according to manufacturers' vision, **two tendencies** can be identified in the design and development of SD-WAN solutions: (*i*) ***software-orientation***, led by software-related companies like VMware (with VeloCloud), which proposes a **software-based architecture** that relies on the hardware of the network equipment manufacturers and on Cloud providers, making it **compatible with different hardware technologies**, and (*ii*) ***hardware-orientation***, where specialized hardware manufacturers (*e.g.* Cisco) build the environment based on their **own proprietary hardware**[1], achieving a solution whose **complete value chain** is **built by a single stakeholder** with **great performance** results, in exchange for **losing flexibility** in the installation on different hardware equipment.

In any case, **all the commercial SD-WAN platforms** analyzed in the comparative depend, to a greater or lesser extent, on **proprietary hardware**, moving away from generic, off-the-shelf networking hardware due to performance, price or simplicity. In fact, it is expected that, through 2021, more than the 80 % of the SD-WAN solutions will still be delivered on dedicated hardware, maintaining this trend [31].

Unlike most enterprise network solutions, **Alviu** differs from them as it **does not lock** to a certain **vendor** and range of **telecommunications equipment**, this being one of the key aspects from Alviu's value proposal. This way, and coupled with its **open-source nature**, Alviu provides a SDN-driven, unified WAN and LAN with end-to-end network slicing to enterprise and academic networks based on **low-cost** but **high performing commodity network appliances** and **white-box switches**, which can be purchased and combined depending on user's needs, combining performance and low cost in one single solution.

Furthermore, in terms of simplicity, Alviu offers a **flexible and movable platform** with an **easy-to-use interface** to **define the SD-WAN scenario** to be managed, which eventually triggers the **installation** of the **dynamic**

[1]However, physical machines from other manufacturers could be also used to supplement the hardware part.

intent-based routes that provide connectivity to the network and, also, activates specific **value-added capabilities**, such as policy-based control, monitoring and error detection services, network virtualization, zero-touch deployment, distributed firewall rules, data encryption, dedicated QoS policies, load balancing in redundant topologies, or the integration of legacy routing protocols (*e.g.* OSPF or BGP) [9].

2.2.2. Intent-Based Networking

Intent-Based Networking (IBN) systems are expected to be used by more than 1000 large enterprises in production by 2020, up from less than 15 in 2018's second quarter [33]. This paradigm expects to provide a **full lifecycle management** of **network systems**, being able to automatically convert, verify, deploy, configure and optimize the network in order to **achieve a target network state**, according to the **intent** expressed by the user, also **performing whatever it takes** (*e.g.* solve abnormal events) to ensure the **network reliability**.

These intents, unlike the traditional configuration of networks, which is mostly manual and technology-dependent, just require to **input** the **desired business requirements** to be achieved by the network in an **abstract way** (*e.g.* I want the network to meet a specific QoS level for a given service). As a result, the IBN technology can automatically **translate** the **intents** into **real-time operations** to be performed in every networking phase (*i.e.* provisioning, deployment, management, troubleshooting and remediation) to satisfy the requirements exposed and to provide more intelligence and autonomy [28] [34].

Since its first official specification by the **Open Networking Foundation (ONF)** [35], this technology has proven to be of **interest** to different **research entities**, including **standardization organizations, open source communities, industry and academia**, which are actively studying the mechanisms and applications of IBN [34]. In what concerns **Alviu**, it leverages the **lessons learned** from open source's and academia's efforts related to the integration of IBN in the **ONOS SDN controller** [36]. This integration have been traditionally studied from two different perspectives: (*i*) the construction of an **IBN architecture over** the **ONOS** controller, and (*ii*) the **enhancement of ONOS core** by including specific **modules and applications**.

For the **first case**, there are examples like [37], which defines an **intent-based northbound interface (NBI) architecture** based on micro-services and service-oriented architectures, relying on the **ONOS Intent Framework** [38], to build more powerful SDN applications, and [39] introduces an hierarchical, intent-based architecture that plays the role of **network hypervisor** and **SDN controller**, being able to manage and virtual networks through intents. In the **second case**, [40] expects to provide an **ONOS application** which is able to **calculate optimal paths**, using the Dijkstra algorithm, in terms of the number of hops and the bandwidth value on the link, and [41] proposes an **ONOS module** that allows to **monitor and reroute** specific ONOS intents created beforehand.

Alviu's philosophy is closer to the **first perspective**, building an **orchestration solution on top** of the **ONOS** controller that uses the **states** and the **transition between states** proposed by the ONOS Intent Framework to

create its **own vision** of the **network intent**. In this way, the **input** information received by Alviu are the so-called **Network Intent Descriptors**, which contains the information related to the SD-WAN scenario in JSON format, triggering the execution of different **operations** in Alviu to fulfil the configuration proposed in the descriptors. One of the purposes of this approach is to make the **network management and orchestration (MANO)** operations **easier** for system and network administrators, as the current IBN commercial solutions in the industry market are mainly focused on solutions based on the traditional telco operator's MANO mechanisms, whose workflows are too complex for these types of networks, advocating simplicity and agility instead.

2.2.3. Integration with Legacy Networks

With the help of the increasing adoption of hybrid Cloud and enterprises' expanding bandwidth requirements, **business WAN traffic flow patterns** are becoming more **software-based** and **hybrid** by nature [28], **coexisting legacy routing protocols** (*e.g.* OSPF or BGP) and **mechanisms** (*e.g.* MPLS) with novel **SDN technologies**. This makes sense in **operator-controlled networks** with a significant networking infrastructure, where it is **not an option** to demolish everything and **rebuild from scratch** a **full SDN-based architecture**, as changes cannot be made overnight. In this context, **SD-WAN** should be introduced in **progressive deployments** such as the renovation of a part of the network or the construction of a new localized infrastructure [30].

The **current solutions** from the state of the art that take this topic into account are mostly based on **multi-domain SDN scenarios**, where there are **different domains** controlled by **SDN** that may **need to be connected**. For instance, [42] introduces a **West-East Bridge (WE-Bridge)** mechanism to enable different **SDN** administrative **domains** to **peer and cooperate**, [43] performs a strict analysis of **multi-domain SDN interconnections** focusing on the **programmability properties** and their effect on the **performance** of connectivity services, [44] proposes a **multi-domain SDN provisioning framework** on top of an ONOS controller that uses **BGP**, through a specific ONOS app based on the **ONOS SDN-IP application**, for managing the interconnection between SDN domains, and and [45] presents **B4**; the **Google's private SD-WAN**, which uses **BGP** to interconnect the different **WAN sites**.

However, there are also **other approaches** that consider the **connection between SDN and non-SDN domains**, achieving a full integration between SDN and legacy networks, as already done in [46], which proposes a **topology-based hybrid model**, separating the nodes controlled by each paradigm (*i.e.* IP and SDN networks) in different **regions** that may be interconnected. This is, in fact, the **approach selected by Alviu**, allowing to maintain the network granularity with **domains** based on **different technologies** that can be **interconnected**, also introducing the concept of **transit network** describing a **SDN domain** that **interconnects** with **multiple legacy domains**.

For this purpose, the **Quagga routing suite** [47] is used in the **edge SDN switches** (*i.e.* the switches from a SDN domain that are connected to legacy domains) to handle **legacy protocols messages** exchanged with **edge routers**

from **other domains**, whose relevant information is eventually used by the **SD-WAN orchestrator** to enable or disable the traffic between domains. **Quagga** is, actually, a well-known, open-source tool **present** in **SDN deployments** frequently combined with legacy routing scenarios in the state of the art, as it can be seen in [43]–[44][46][48][49], therefore it can be considered a **mature technology** for this purpose.

2.3. Virtualization Mechanisms in 5G Networks

There is a wide **consensus** among the **research and industrial communities** that **future mobile networks** will be **software networks**, due to **flexibility** and **cost reasons** (in fact, some functionality such as the Evolved Packet Core is already provided in specialized software running over general-purpose hardware). However, the **ability** to **match** the **network demand** at **any point in time** is still **missed**, requiring a **technology** that is (*i*) **re-configurable** over **very fast periods**, and (*ii*) **very granular**, to **reduce** the **cost** of **inaccuracies** in the **re-configurations** in *e.g.* the access network [50].

A **similar problem** has already been tackled by the **Cloud Computing community**, which has continuously provided **faster** and **more scalable solutions** over the last decade. Additionally, these solutions have also made the system **more flexible and open**, enabling the appearance of **new business models**.

In what follows, the current **landscape** of **network *softwarization* and *modularization*** is described, **comparing** the **advances** in **Computer science** and **mobile networking** to finally introduce the **serverless paradigm** and the **advances** made in that field.

2.3.1. Evolution of Cloud Computing

The **first major achievement** in **Cloud Computing** took place in the **early 2000s**, with the appearance of **new virtualization solutions** such as *Xen*, *VMWare* or *KVM*. With these technologies, which efficiently exploited the **novel virtualization extensions** supported by the **hardware**, a **new way** of **providing services** "conquered" the Cloud Computing environment. It consisted of a **more modular architecture** that supported a **higher re-configuration frequency** but also requiring a **higher management complexity**. This achievement is marked with an **'A'** in Figure 2, where the different **transitions** considered in this Section are depicted along **three dimensions: architecture, re-configuration frequency and complexity**. For **this first transition**, the Figure illustrates how the architecture evolved from **monolithic functions** to **modular ones**, supporting a change of **operational timescales from years to months**, but also increasing the **complexity** in the **operation**.

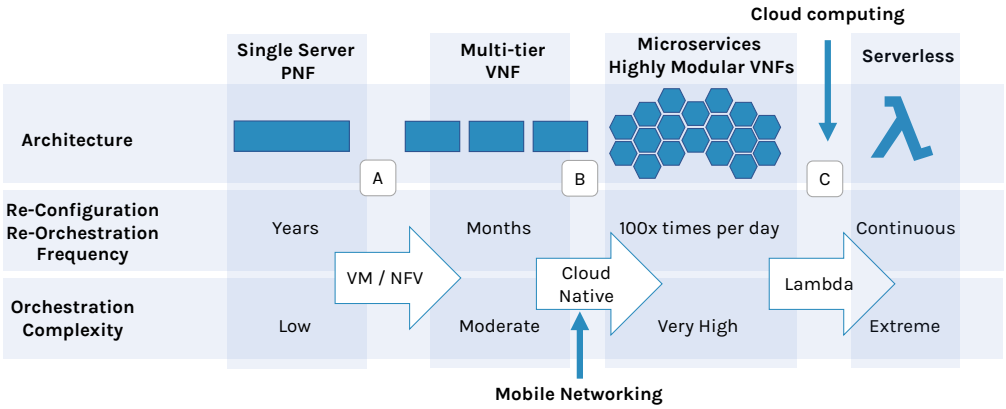

Figure 2: Major transitions in the adoption of softwarization.

The **second transition** identified is marked with a **'B'** in Figure 2 and happened in the **early 2010s**. It was caused by the arrival of the so-called **microservices paradigm** [51], introduced by **software architects** to support a much **finer granularity**. This paradigm supports, for example, that a database server can be split into many tailored microservices, each one fulfilling a specific functionality, *e.g.* an account manager or the data storage system. This transition was driven by the availability of **new virtualization technologies**, such as *Docker* and *LXC Containers*, which allow the **deployment and scaling** of **small virtual applications** in a much more **lightweight fashion**, enabling also **new coding practices** such as **DevOps** [51].

Finally, the **last transition** to be remarked in Figure 2 is the **'C'**, namely, **serverless architectures** [52]. This recent paradigm, also known as **Function as a Service (FaaS)**, is an **extremely liquid approach** to **scalability and resource usage**. With this approach, a **tenant** creates **calls** to **functions**, *i.e.* the **minimum building block** of a **software component**, which are **served** by the **infrastructure provider**. In this way, the **software component** becomes both **platform- and server-independent**, as the different functions of the same program could be served by different providers.

2.3.2. Evolution of Mobile Networking

There is currently a **huge research effort** on the ***softwarization*** of the **mobile network**. Among other efforts, **5G mobile communications** are working towards the introduction of a fully **softwarized architecture** [53]. However, as compared to the Cloud evolution, the telecommunications world is still **half-way** in this transition, **despite the adoption** of technologies such as **Software Defined Networking (SDN) and Network Function Virtualization (NFV)**, which have **helped towards** the ***softwarization*** of network architectures, and the architectural trend towards their ***modularization***, with a clean separation between the control and the user planes. This way, the trend is to **split**, already at the **architectural level**, the formerly **monolithic nodes** into **several smaller logical entities**. Thus, the 5G Next Generation NodeBs (gNBs) can be split into centralized and distributed units, denoted as gNB-CU and

gNB-DU, respectively [54], while core network components grow both in number and in functionality.

As depicted in Figure 2, the **telco world** is lagging in the adoption of novel software paradigms, being **approximately at mark 'B'** of **Cloud Computing**, with **achievements** such as:

- The **standardization of 3GPP Release 15** [55], which specifies the **Service Based Architecture (SBA)**. This represents a **new paradigm** for the **5G Core network** and is driven by the trend towards the ***modularization*** of the **network**. With this approach, the formerly **static interface** between **different elements** has evolved into a **flexible bus**, which hosts **HTTP REST primitives between modules**.
- The concept of **Cloud-Native Network Functions (CNF)**, which is making its way into the current technology. In fact, there are already proposals for the design of **Cloud-native VNFs**. However, they are in a **very early stage** and mostly involve **Core Network VNFs only**.

Despite these achievements, mobile networks are still **in the middle of this transition** as the **Cloud-native paradigm** has **not** been **fully adopted** into **operational networks**. This is caused by the **poor agility** of the **current state-of-the-art solutions**, and the fact that **current VNFs** are **not truly agnostic** to the **underlying NFV infrastructure**. While **dynamic Cloud resources orchestration algorithms** are currently **under** study [56], the **VNFs** that would be running on such resources are still **not optimized** for this type of operation.

So even if the **efforts** towards the Cloud-native transition of the NFV are still **ongoing**, the **research** community shall prepare for the **next transition**. This will introduce a complete **re-design** of the **whole mobile protocol stack**, which will certainly facilitate a **dynamic resource orchestration and assignment**, being (*i*) **more efficient** in terms of both **resource and time granularity scalability**, and (*ii*) capable to **elastically adapt** to the **instantaneous demand**. With such a protocol stack, the **deployment and operational costs** of the network would be **reduced to their minimum**. Given that this flexible and on-demand operation of the network closely **resembles** the current operation of **Cloud Computing platforms**, it should also **follow similar principles**, hence the name **Serverless Mobile Architectures** introduced on Section 7.1. As a result, in the near future, **serverless computing** would become a **possibility** for implementing a **wide range** of **communication services**, being also considered by **service providers** for implementing **new networking services** [57].

2.3.3. Introducing the Serverless Paradigm

For the **deployment** of **Cloud applications**, **serverless** is emerging as a **popular paradigm** in the **industry**. It started becoming **popular** thanks to **Amazon**, with its related product called ***AWS Lambda***, which was followed by **similar products** from other Cloud providers, such as ***Google Cloud Functions***, ***Microsoft Azure Functions*** or ***IBM OpenWhisk*** [58].

All of these technologies **share** the concept of **deploying and executing small code snippets without any control** over the **compute resources** on

which the **software** is **running**. This way, it has also been extended to **open-source projects** such as ***OpenLambda*** [59], an **open-source serverless computing platform** for building **next-generation web services** and **applications**, creating a **ecosystem** based on the concept of **Lambda**, which is the equivalent to the idea of **serverless function** itself.

Apart from that, it is also attracting the attention of the **academia** and the **research world**, with projects such as **SAND** [60], which is a **serverless platform** based on the introduction of **application-level sandboxing** and a **hierarchical message bus** for the communication between processes, or **SOCK** [61], a **container system** specifically **optimized for serverless workloads**.

However, these serverless platforms and techniques are somehow **tied** to the **platform** in which they are implemented, as providers indend to **lock-in their serverless clients** by also offering **extra services** that assorts the provisioning of serverless applications. For this reason, the presence of **serverless frameworks** are becoming increasingly **popular**, having the purpose of **abstracting the technical features** of the serverless **platform** or Cloud **infrastructure** for **application developers**, making then **easier** the process of **designing, developing and deploying** the **serverless functions** [62].

Examples of these frameworks, in terms of **open-source solutions**, are ***OpenFaaS***, ***Kubeless***, ***Fission*** or ***OpenWhisk*** (this last one already mentioned before), among others, existing already **studies** that **compare** some of these **frameworks** from **different points of view**; *e.g.* taking into account the **support** offered by the framework on **each software's lifecycle phase**, like in [62], or performing **tests** for measuring some interesting **metrics** such as the response time, the ratio of successful responses or the impact of auto-scaling on some of these metrics, as done in [63]. The decision of **choosing one solution or another** will depend on the **stakeholders** implied on the system to be developed, as **each framework** offers **different capabilities** that make it suitable for particular use cases.

Furthermore, for **complementing** this serverless ecosystem, other **technologies** have been proposed in the state of the art for **solving specific needs** in this kind of platforms. This is the case, for example, of ***Kata Containers*** [64], which brings together the **speed of containers** and the **security offered by virtual machines** into a single product that expects to propose a **two-layer system-wide isolation** for **improving the security capabilities of containerized components**. These containers can be used for **handling serverless workloads**, being deployed over specific **hypervisors** adapted to the **serverless** trend, such as ***Firecracker*** [65], a **Virtual Machine Monitor** that uses ***KVM*** virtualization infrastructure to provide minimal virtual machines (also known as **MicroVMs**).

Part 1. Monitoring of Network Slices

3

An Adaptable Monitoring Framework for 5G Environments

The evolution of mobile networks from 2G to 4G generations was mainly focused on providing a better quality of experience to end users, by increasing the bandwidth offered by the network at the radio link segment. However, **5G networks** have a broader target, shifting traditional communication networks to a new generation mobile network that embraces other business sectors.

In the case of 5G, the authors of [66] have reported the **service requirements** expected by verticals, which is the terminology used by 5G to define these business sectors moving to 5G as the main transport infrastructure. Due to the stringent and different requirements imposed by all these potential verticals deploying their services on top of 5G networks, the most important SDOs tackling the 5G standardisation, like the 3GPP, have introduced the concept of **Network Slicing** [67], which provides multiple isolated logical networks from a single physical one.

In this approach, each logical network may support a particular type of 5G service; *e.g.* Enhanced Mobile Broadband (eMBB), Massive Machine-Type Communications (mMTC) or Ultra-Reliable and Low Latency Communications (URLLC). As a matter of fact, 5G telecommunication operators have to design their networks to support all these services and to guarantee that the **Key Performance Indicators (KPIs)** demanded by their verticals are satisfied.

To support this, 5G networks will require a **flexible and efficient monitoring system** to **guarantee all Service Level Agreements (SLAs)** between operators and users. In this aspect, the collected network metrics would serve to optimize the performance of the network, and to **confirm the achievement of the KPIs**.

This Chapter presents a **Monitoring framework** capable of meeting the above requirements. In particular, this Monitoring system has been fully designed, implemented, evaluated and deployed within the scope of the **5G EVE** European project [10], providing a platform which allows the distribution and consumption of metrics and KPIs (based on a formula that is applied to one or more metrics) in multi-site 5G scenarios, where different verticals from different stakeholders are

implemented over a shared infrastructure. This platform is also flexible enough to be implemented in other projects as well as by telecommunication operators within the scope of advanced 5G networks.

The main **topics** that will be discussed in this Chapter are the following:

- First of all, Section 3.1 presents the **Monitoring general architecture**, which has been designed as a scalable, reliable, low-latency, distributed, multi-source data aggregation and re-configurable architecture.
- Secondly, Section 3.2 justifies and details the **implementation** selected in the 5G EVE project to instantiate the proposed architecture, based on the **publish-subscribe paradigm**.
- Then, Section 3.3 **validates such implementation** against the requirements imposed to the Monitoring architecture from the 5G EVE project specifications.
- To confirm that the system works properly in a **real scenario**, Section 3.4 summarizes the execution of a given experiment in the 5G EVE platform, focusing on the Monitoring and Data Collection workflow.
- Finally, Section 3.5 **summarizes and concludes** the work related to this Monitoring platform.

3.1. System Design

3.1.1. System Requirements

The **characteristics** to be offered by the Monitoring service which will be used as input for the design of the platform, according to the thorough analysis of the 5G EVE infrastructure and service requirements done in [68], are the following:

1. The Monitoring distribution architecture must support **multi-site experiments** involving distant sites.
2. The platform must deal with experiments that may generate **monitoring data** in the order of **gigabytes**.
3. Monitoring data has to be **available** to experimenters after the experiment has concluded, estimating a retention time of at least 2 weeks.
4. **Redundancy** is needed to offer a fault-tolerant system.
5. The architecture must be **flexible** enough to accommodate a wide variety of elements to be monitored.
6. The support of some **pre-processing techniques** (*e.g.* translation across formats) may be needed for an efficient subsequent processing.
7. The collected metrics may be used and post-processed by a **KPI Validation Framework**, also defined within the 5G EVE project, which can also distribute the calculated KPIs' values from a specific set of metrics using this platform.

3.1.2. Proposed Architecture

The features presented in Section 3.1.1 result in the **architecture** for the **collection, distribution and pre-processing of monitoring data** presented in Figure 3, which satisfies all the requirements described above.

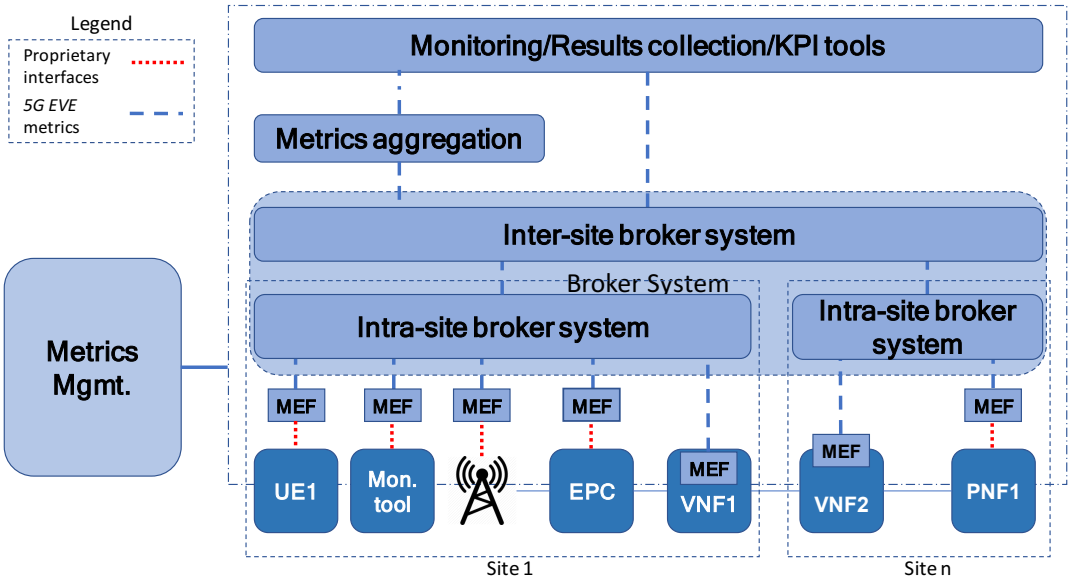

Figure 3: Monitoring metrics architecture.

In this general-purpose architecture, **two sets of components** can be distinguished:

- In dark blue, some **elements of the experiment infrastructure to be monitored**, included here for the sake of completeness, and which may be User Equipment devices (UEs), monitoring tools, (4G or 5G) radio antennas, Physical Network Functions (PNFs) or Virtual Network Functions (VNF).
- In light blue, the **elements that compose the Monitoring platform** itself, which will be presented next by following a bottom-up/west-east approach.

The first component of the architecture to be described is the ***Metrics Management*** entity, whose main role is to properly **configure the other components of the architecture**, providing the configuration of the necessary data service function chains in order to enable metrics to be gathered, filtered, normalized and relayed to upper layers in the architecture to be further processed.

The component of the architecture directly connected to each experiment's infrastructure is the ***Metrics Extractor Function (MEF)***, which takes care of **extracting and translating** (if required) the **metrics** generated by a heterogeneous set of infrastructure components.

In the proposed architecture, it is assumed that there is a **one-to-one logical relationship** between a particular ***MEF*** and its **monitored infrastructure**

component, although this may be implemented in different ways, mainly depending on if it is fully, partially or not integrated in the monitored components, as presented in Figure 3.

This modular design allows to have **dedicated *MEFs* per infrastructure device**, which satisfies the requirement (5) explained in Section 3.1.1. This way, it would be possible to implement dedicated *MEFs* to handle any kind of proprietary interfaces (dotted red lines in Figure 3). Then, the *Metrics Management* entity instructs each *MEF* to extract metrics from its monitored component and to make them available to the upper layer (*i.e.* the *Broker system*, which will be described next).

It is important to remark that all these metrics have to follow the **5G EVE format** [69] to satisfy constraint (6) presented before. This might require a translation from a proprietary or different standard formats to the 5G EVE one, in order to handle all the messages received from the *MEFs* in an unified way. This format is presented later in Section 3.2.

The monitoring data is then received by the ***Broker system***, which is in charge of **storing and distributing** not only the **metrics** obtained from different sites, but also the **KPIs' values** generated in upper layers. For accomplishing requirement (1), **two brokering levels** have been defined:

- The ***Intra-site broker***, deployed per site, whose role is to eventually **harmonize the metrics' format** to provide data in an unified way, preserving the data privacy of each site.
- The ***Inter-site broker***, which **interconnects all sites together** to both:
  - Aggregate metrics through the ***Metrics aggregation*** component, generating **new metrics** automatically based on those provided by the *MEFs*. For example, a given function may receive the instantaneous transmission rate at a given network interface every second, to then compute the mean rate in a window of ten seconds. More complex functions may estimate the average rate between two points in a defined window time.
  - Directly provide them to the different tools grouped in the ***Monitoring/Results collection/KPI tools*** entity, which is the entity consuming metrics from the *Metrics aggregation* or the *Inter-site broker system*, laying the ground for a set of **value-added additional components** that range from the KPI Framework for performance diagnosis already commented, which allows to fulfill requirement (7), to more complex modules such as data analytics platforms, SLA enforcement mechanisms or data visualization services, which can be fed from the monitoring data provided by the system. The first example, related to data analytics, will be further explained in Section 4.2.2.

Finally, in order to satisfy requirements (2), (3) and (4), the *Metrics Management* entity is the responsible for properly **configuring all levels** of the broker system in a **per-experiment basis**, also enabling the necessary security mechanisms to ensure that only the actors belonging to a given experiment can manage the monitored data of their experiment and not others.

3.2. Implementation Based on the Publish-Subscribe Paradigm

The instantiation of the Monitoring architecture presented in Section 3.1 over the 5G EVE architecture [10] [70] results in the composition of a specific **component chain**, depicted in Figure 4.

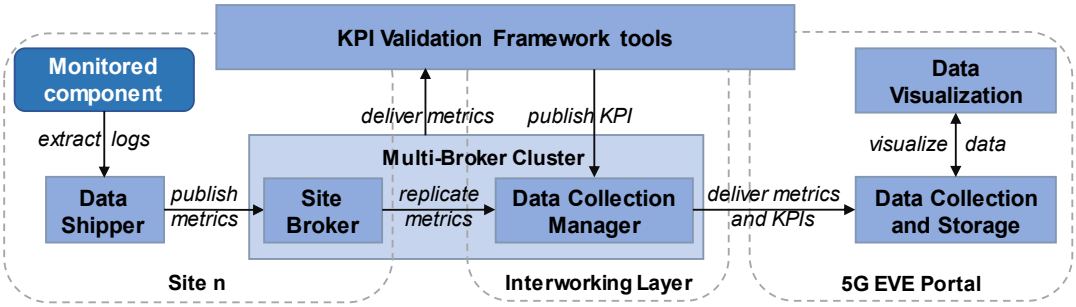

Figure 4: Component chain that implements the general Monitoring metrics architecture in the 5G EVE platform.

The keystone of this chain is the **publish-subscribe messaging pattern**, providing a distributed system with parallel data processing capabilities which allows to meet the requirements imposed to the Monitoring platform. This paradigm suits the multipoint-to-multipoint monitoring data flow of the 5G EVE project, closer to a big data pipeline rather than to a classic relational database model, as a massive volume of data is pushed from site facilities without a specific format, which is not suitable to be stored in a relational model [71].

Following the above, the ***Broker system*** is mapped into a set of **publish-subscribe queues**, starting from **local queues** deployed in **each site facility** (*Intra-site broker*) that aggregate metrics to the **Interworking publish-subscribe queue** (*Inter-site broker*), which provides a **transparent and seamless access to metrics' and KPIs' values** from all sites to components from upper layers. In Figure 4, each *Intra-site broker* is represented by a ***Site Broker*** entity, and the *Inter-site broker*, together with the *Metrics Management* service, are implemented by the ***Data Collection Manager*** component in 5G EVE architecture.

All the *Site Brokers* and the *Data Collection Manager* are based on *Apache Kafka* [72], an open-source, industry-proven publish-subscribe tool that manages data pipes and **forwards the published data to the different components subscribed**, providing a higher maximum sustainable throughput than other broker-based message-oriented middleware technologies [73]. Moreover, it also implements several useful functionalities related to data transformation and normalization (*Kafka Streams*), security (*Kafka ACL*) or data persistence (*Kafka Store*), among others [74]. This makes *Kafka* an optimal solution for data-movement, frequently adopted as pipe to different processing systems [75].

This hierarchical architecture can be encompassed with the so-called ***Multi-Broker Cluster***. In this way, the *Site Brokers* located in each site **replicate the data** received towards the *Data Collection Manager*, which is in charge of

providing the data that come from different sources to the entities interested in consuming that data.

The specific **building blocks** that composes the *Data Collection Manager* can be seen in Figure 5, reflecting the usage of ***Apache Kafka*** as the **core component** of this module, which is **interconnected** to all the ***Site Brokers*** from each **site facility**. All the brokers are **coordinated and orchestrated by *Apache ZooKeeper*** [76], an open-source cluster coordinator for distributed systems. Additionaly, a ***Python* script** [77] is also running in the DCM for **managing the data** handled by this component, which are the names of the processes to be orchestrated in *Kafka*, called **topics**. The steps to deploy and configure all these modules are available in [78].

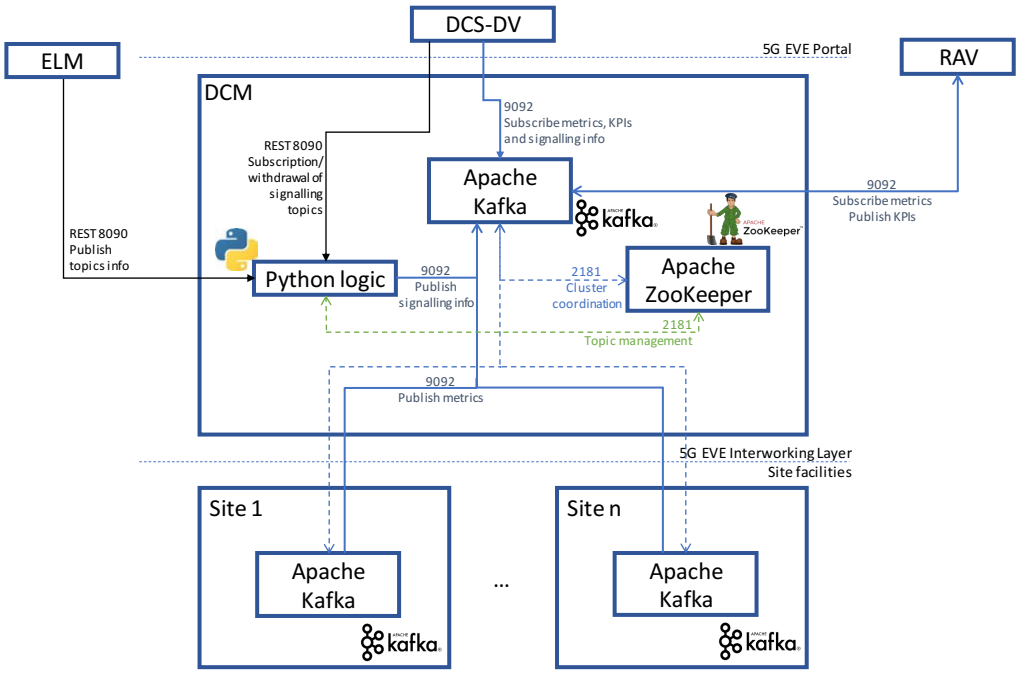

Figure 5: Data Collection Manager architecture.

In particular, the **information model** that defines the different topics that are handled by the *Multi-Broker Cluster* in a concurrent way is described in the so-called **Topic framework proposal** [74]. In that way, each **topic** is designed to manage a specific **set of data** (mainly related to a single metric or KPI to be monitored) that will be different to the data consumed by the other topics, enabling dataset isolation.

There are two main types of **topics** defined in the Topic framework, which are:

- **Data topics**, where each of them **transports the values of the metric or KPI** they refer to, followed by some extra information that may be useful for other modules. In particular, this information corresponds to the **5G EVE format** mentioned in Section 3.1.2, which specifies the **fields** that the message containing the data related to the metrics' and KPIs' values to be handled by the Monitoring platform must have. The **data structure** to be followed in the case of the 5G EVE project, encoded in a JSON format, is the following [69]:

Code 1: Information model used in 5G EVE to define the messages containing monitoring data.

```
{
  "records": [{
    "value": {
      "[metric_value|kpi_value]": <value captured from
         the execution>,
      "timestamp": <time in which the value has been
         captured>,
      "unit": <unit used for the value>,
      "device_id": <ID of the device, to be used in
         upper layers if needed>,
    }
  }]
}
```

- **Signalling topics**, used to **deliver the name of data topics related to each metric or KPI** to be monitored for a given experiment, process triggered by the Experiment Lifecycle Manager entity (ELM in Figure 5). This is a function that fits in the scope of the *Metrics Management* service, which automates the process of creation and deletion of topics.

The components that interact with the *Multi-Broker Cluster* can be classified as **publishers and subscribers**, depending on whether they **produce data** to the publish-subscribe platform or they **consume** it. This distinction allows to simplify the workflow during the experiment execution, as subscribers only need to be subscribed to the topics related to the metrics and KPIs they want to consume data from (*i.e.* the ones used in the experiment), and then, when a publisher produces data to these topics, the information is automatically delivered to the subscribers that are listening to the topics.

The main component which performs the **metrics' data publishing operation** is the ***Data Shipper***, playing the role of the *MEF* component from the general architecture, and whose objective is to execute the **log-to-metric operation** that transforms the heterogeneous, raw logs obtained from components and collection tools into metrics with a common, homogeneous format. These Data Shippers can be placed within each component as a **lightweight software** (ranging from general-purpose solutions already developed and packaged like *Beats* [79] to more complex solutions programmed for specific-purpose cases) or can be deployed in a **separated server**, but in both cases, they must be connected to the *Multi-Broker Cluster* with a logical connection.

Moreover, the ***KPI Validation Framework tools***, *e.g.* the Result Analysis and Validation (RAV) component in Figure 5, also contain **publishers** providing **KPIs related to a given set of metrics** received from the *Multi-Broker Cluster* after being published by specific *Data Shippers*, which means that these *KPI tools* also implement a **subscriber for each metric** to be consumed.

Finally, the ***Data Collection and Storage-Data Visualization component*** (DCS-DV in Figure 5) performs the expected functionalities provided by the *Monitoring/Results collection* entities with a solution based on the *Elastic (ELK)*

Stack [80] [81]. This component, whose complete architecture is presented in Figure 6, receives the metrics' and KPIs' values through a specific **subscriber for each metric and KPI**, and it is separated logically in two main blocks [82] [83]:

- The ***Data Collection and Storage*** component (DCS), which collects the metrics and KPIs to which this component is subscribed through *Logstash*, from the *Elastic Stack*, and provides **data persistence, searching and filtering capabilities** (related to the *Metrics aggregation* functionality from the general architecture) for obtaining the useful data to be monitored during the experiment thanks to *Elasticsearch*, also from the *Elastic Stack*. In this sense, a ***Python* script** [84] is in charge of **automating the process** of correctly configuring both *Logstash* and *Elasticsearch*.
- The ***Data Visualization*** component (DV), in charge of enabling the monitoring of the progress of the experiment in terms of that **monitoring data displayed** through *Kibana* from the *Elastic Stack*. For this purpose, a set of **dashboards** are created for experiment, presenting the graphs related to each metric or KPI monitored. These dashboards are created by a **Java application** placed in the DCS [85] [86], which directly interacts with Kibana.

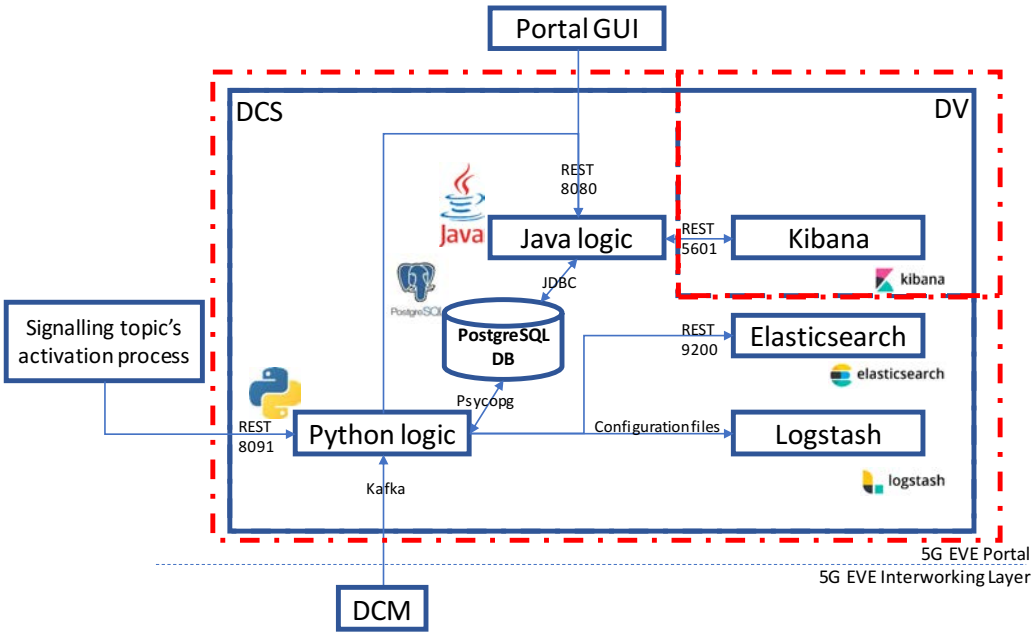

Figure 6: Data Collection and Storage-Data Visualization architecture.

The steps to deploy and configure all the DCS-DV modules are available in [87].

3.3. Performance Evaluation

To assess and validate the proposed Monitoring framework implementation, the **testing process** described below has been followed, based on the application

of a top-down approach. All the tests performed are based on **single-broker experiments** (*i.e.* only the *Data Collection Manager* is used, without interacting with *Site Brokers*) to characterize the platform in terms of several performance parameters.

3.3.1. System Assumptions

The definition of the System Under Test (SUT) parameters is bound to the **5G EVE multi-site platform's operation**, in which a set of **experiments** derived from the different use cases defined in the project **may be running simultaneously** at a specific time, sharing all the computing and network resources provided by both the 5G EVE platform and the site facilities.

As a first approach to the evaluation, the following **assumptions** were made:

- The Monitoring platform must be prepared to deal with extreme conditions, such as the simultaneous execution of a considerable amount of experiments. As the 5G EVE project initially proposes the validation of experiments from **six specific use cases** [68], the **execution of an experiment from each use case at the same time** can be taken as the worst case study to validate, resulting in **six simultaneous experiments** to be handled by the Monitoring platform. For testing purposes, each experiment will last 5 minutes.

- Each experiment can define a different **number of metrics and KPIs** to be collected and monitored during the experiment execution, depending on vertical's needs. For this evaluation process, as these metrics can be extracted from different sources (*e.g.* UEs, VNFs, PNFs, etc.), and each source may have several related metrics or KPIs, it can be assumed that **each experiment** will require the monitoring of an average of **20 parameters**. Furthermore, as each **monitored parameter has a topic assigned** for the transport and delivery of their corresponding collected data, **each experiment** on average will create **20 topics** in the Monitoring platform. As a result, the **maximum number of topics**[2] created in the platform would be $20 \times 6 = \mathbf{120}$ in this case.

- The **size and the publication rate of the messages** containing the values of metric or KPI managed by the Monitoring platform depend on the **nature of the data transported**. As a result, **four different alternatives** have been considered for the tests:

  - **B** and **1 KB** messages for **data traffic** (*i.e.* numeric or string values), representing the **80 %** of all the monitoring traffic (**40 %** for **each case**). The **publication rate** for both options is set to **1000** $messages/s$.
  - **100 KB** and **1 MB** messages for **multimedia traffic** (*i.e.* images or video frames), which would be the remaining **20 %** (**10 %** for **each case**). The **publication rate** for both cases is less than the data traffic

[2] This figure does not include the signalling topics presented in Section 3.2, whose footprint is not significant compared to these data topics.

one, as the received throughput almost never reached that value due to the message size, with **10** *messages/s* for **100 KB messages** and **1** *message/s* for **1 MB messages**.

The percentages have been selected assuming that **most of the data will be small-side messages**, but also considering that there may be larger messages, mainly related to multimedia data. As a result of the figures selected for each kind of message, this results in a **concurrent publication rate** of approximately **102,4** *Mbps* **per experiment**.

All these assumptions are summarized in Figure 7 for better understanding:

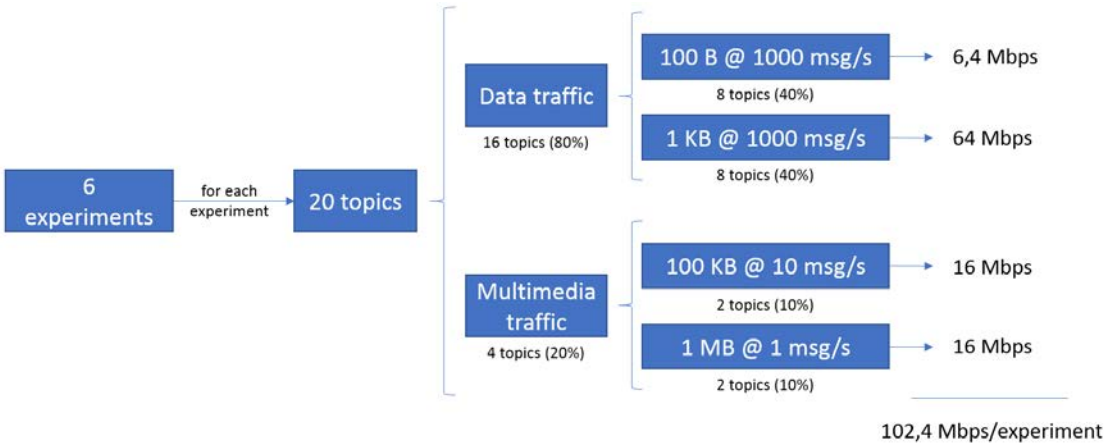

Figure 7: Calculations made for system setting.

- Another important parameter related to the publishers is the **message batch size**, which controls the **amount of messages to collect before sending messages** to the *Multi-Broker Cluster*, and which was **set to 1** after validating that higher values of this parameter cause worse results in terms of latency.
- The selected values of publication rate for each message size are also coherent for the subsequent calculation of the **disk size estimation** for each *broker node*, which is computed as $D = s*r*t*f/b$, where s is the message size, r is the publication rate, t is the retention time (at least 2 weeks, as discussed in Section 3.1.1), and f and b are both the replication factor and the number of brokers in the system, typically $f = b - 1$, this leading to a **value slightly below 100 TB**, which is an estimation of the expected amount of data handled in the project.

3.3.2. Testbed Setup

The **testbed** used for the evaluation of the architecture consists of **two Virtual Machines (VMs)** deployed in a host located in the **5G EVE Spanish site facility, 5TONIC** [88], using *Proxmox* [89] as virtualization environment. This host is equipped with 40 Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20 GHz and

128 GB RAM. The distribution of components in each VM can be seen in Figure 8:

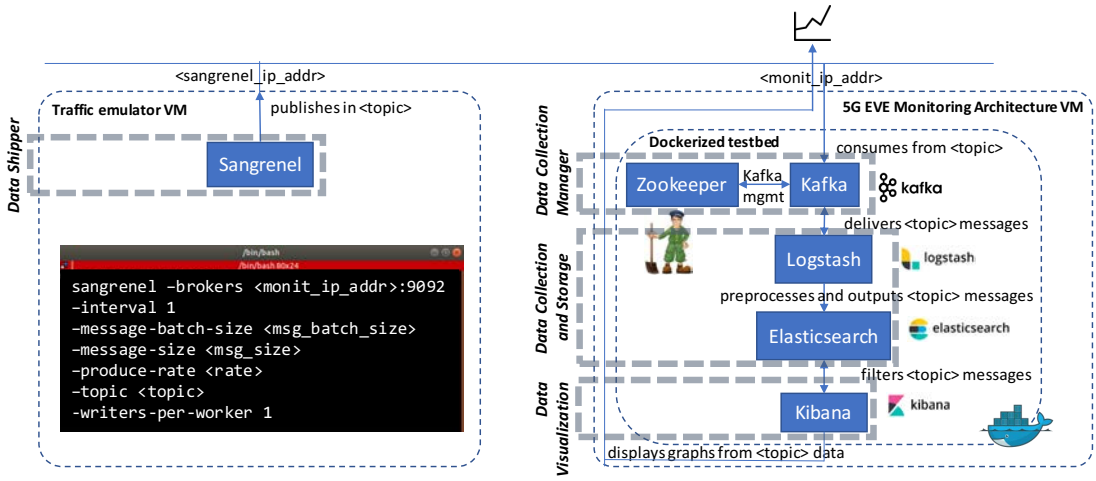

Figure 8: Testbed architecture.

The proposed scenario **simulates** the **monitoring and data collection process** of the metrics and KPIs related to a **set of experiments** where **only one VNF** (VM#1) is **publishing monitoring data** in the **Monitoring architecture** (VM#2). The characteristics and software deployed in each VM, both based on Ubuntu Server 16.04 [90] and executed with 16 virtual CPU cores and 32 GB of RAM, are the following:

- **VM#1:** data publishers are emulated with ***Sangrenel*** [91], a ***Kafka* cluster load testing tool** that allows to **configure** parameters such as the **message/batch sizing** and other settings, writing messages to a specific topic and obtaining, as **output**, the **input message rate** or the **batch write latency**, which are the **performance parameters under study**, being dumped every second.
- **VM#2:** for emulating a single-broker version of the **5G EVE Monitoring architecture**, a ***Dockerized*** [92] **environment** for testing the 5G EVE Monitoring and Data Collection tools [93] has been used, implementing the *Data Collection Manager*, *Data Collection and Storage* and *Data Visualization* components from Figure 4 with a **solution based on *Apache Kafka* and the *Elastic Stack***. For **monitoring the resource consumption** of each container, ***Docker* native tools** (*e.g. docker stats*) have been used.

3.3.3. Preliminary Evaluation Process for a Single-Topic Experiment

To start with the performance analysis of the Monitoring platform, **experiments with only one topic** created were performed, checking that the **system operates correctly and consistently** for each message size and publication rate proposed in Section 3.3.1 without limit of resources, and also with the objective

of defining the **minimum set of computing resources** (RAM and vCPU) for the most critical components of the architecture.

In this set of tests, some of the **assumptions** from the system characterization were **confirmed**, *e.g.* the **poor results for multimedia traffic** when its **publication rate is 1000** *messages/s*, where the Input/Output (I/O) message rate (*i.e.* the received throughput divided by the publication rate) falls from 1 (obtained when the reduced publication rate is used) to 1/4 in the best-case scenario, or that the **optimal value** for the **message batch size parameter is 1** for all types of traffic, as increases in their order of magnitude cause exactly the same increase in the order of magnitude of latency. For example, for a 100 B message size, the batch write latency goes from 0,8 *ms* with a message batch size of 1 to 500 *ms*, where the message batch size is 1000.

Apart from that, it was also observed that the **resource consumption** in the components of the Monitoring architecture is **CPU-intensive** for the **most critical components** of the platform, which are *Kafka*, *Logstash* and *Elasticsearch*, leaving the RAM for working as buffer and cache before saving data to disk. As a consequence, this fact **facilitates the sizing** of these components, as the **RAM value can be fixed** with a specific value (in this case, with **2 GB of RAM** is enough for working properly during the testing process), whereas the **CPU value is the only variable term.**

In terms of **CPU**, for a **single-topic experiment**, ***Logstash*** is the **most critical component**, with a consumption that ranges from 100 to 200 %, needing **4 vCPU** in order not to degrade the performance. However, the CPU consumption in ***Kafka* and *Elasticsearch*** stays below 100 % for all types of traffic, so **1 vCPU** for both of them should be **enough** to cover single-topic experiments. However, in **multi-topic experiments**, which will be studied next, ***Kafka*** becomes the **most critical component** with a noticeable increase in its CPU consumption, whereas *Logstash* and *Elasticsearch* approximately maintain the same consumption profile.

3.3.4. Performance Impact Assessment for Simultaneous Multi-Topic Experiments

In **multi-topic experiments**, the **distribution of performance parameter values** between **topics** of the **same type** (*i.e.* that handle the same type of data, message size and publication rate) in a given experiment is expected to be **uniform** in general conditions, where there are no more priority topics than others.

This assumption is **confirmed** in Figure 9 for the **batch write latency analysis** in **one experiment with multiple topics**, according to the per-experiment topic distribution described in Section 3.3.1. As a result, this confirmed assumption is used in subsequent tests for **accumulating and averaging** the values obtained from **performance parameters** in **topics of the same type**, as if they were a single topic, which allows to **simplify the performance analysis**. Moreover, in Figure 9, it can be also observed that **latency is higher in larger message traffic**, also increasing the **deviation** of the results, represented with the higher values of the 95 % confidence interval estimated for multimedia traffic,

for example. This reflects that **smaller messages** result in better and more precise values of latency.

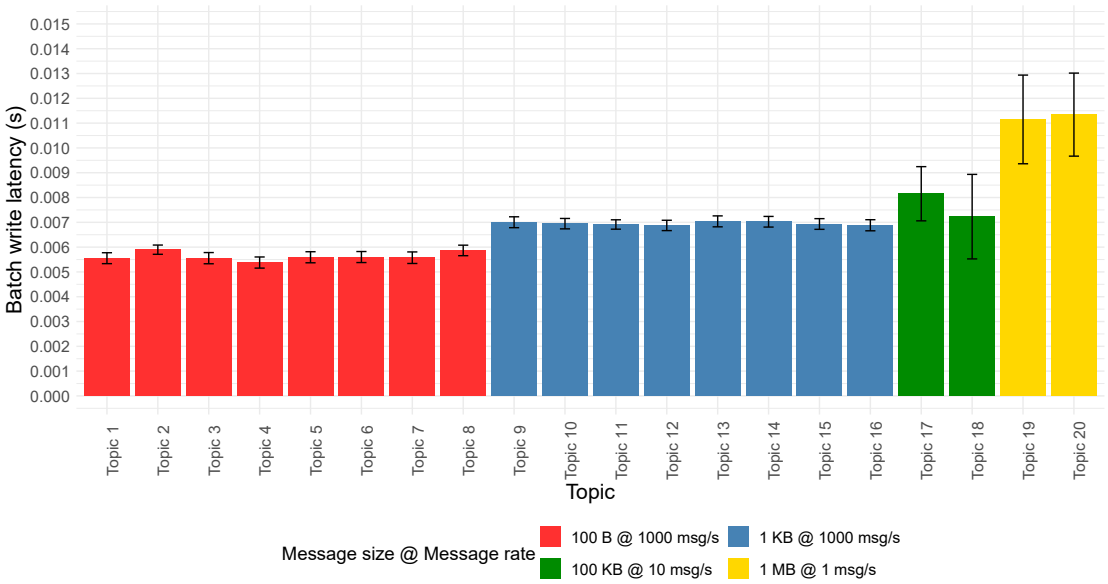

Figure 9: Batch write latency distribution in one experiment with 20 topics.

Continuing with the different **tests** carried out related to multi-topic experiments, they aim at **evaluating two design parameters** that causes variations in the Monitoring platform's workload: (*i*) the **number of topics** created and running in the system as concurrent processes, due to the execution of simultaneous experiments, and (*ii*) the **total throughput received** by the Monitoring system, calculated as the sum of all input message rates received from each topic.

However, a **variation** in any of these design parameters may cause different **effects** in the system in terms of **CPU consumption** or **performance** that must be **characterized**, also checking if the **superposition property can be applied** when both parameters are modified simultaneously. For doing this, the study was divided in **two parts**:

1. A first analysis where **one of the design parameters is modified** while **the other one stays fixed.**

2. A final test including the **modification of both parameters at the same time**, checking if the **superposition** of individual effects is **present**.

Part (1) is presented in Figure 10, where the **CPU consumption** and the **batch write latency** related to **100 B aggregated data traffic**[3] are evaluated for different **examples of experiments**:

- On the **left** side, the **number of experiment** is fixed in **1**, whereas the **total throughput** is **modified**, using the theoretical input message rate as upper limit (*i.e.* 102,4 *Mbps*) and dividing it by values between 1 and 6.

[3]This size is used in the rest of the analysis because it presents a lower value of latency with a tighter 95 % confidence interval, according to Figure 9.

- On the **right** side, the **number of experiments** is **variable**, ranging from 1 to 6, but the **total throughput** for all experiments is **conserved**, which is achieved by dividing the message rate aforementioned by the number of experiments deployed.

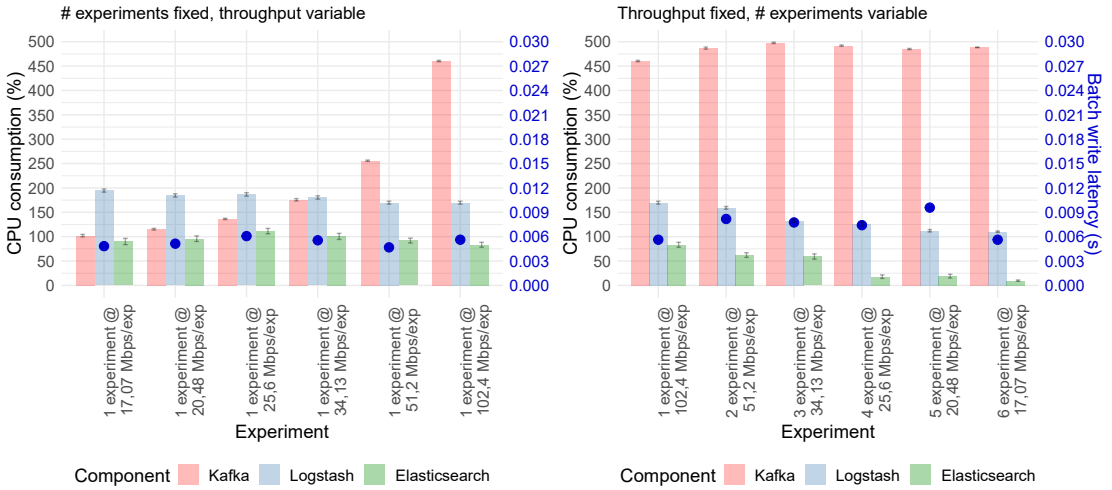

Figure 10: CPU consumption and batch write latency evolution for 100 B data traffic in different experiments, modifying a different design parameter in each case whereas the other one remains fixed.

In both cases, it is observed that the **batch write latency does not vary** when **modifying one of the design parameters**, and it is also true for the **I/O message rate**, which tends to **1**. However, in the **first case**, when the **total throughput becomes higher**, the ***Kafka* CPU consumption increases** with a trend that seems **exponential**, but in the **second case**, the **CPU consumption** also remains **constant** in average.

As a result, while the **total throughput has an effect** in the ***Kafka* CPU consumption** with an **exponential tendency**, the **number of experiments** (*i.e.* the number of topics in the system) **does not seem to influence** the system performance, as long as the total throughput is conserved when there is an increase in the number of topics, taking care of specifying correctly the publication rate in order not to exceed the system limits. However, this is **true** while the **system is not saturated**. When this happens, the effect is similar to the one shown in Figure 11, related to the **part (2)** of the study aforementioned.

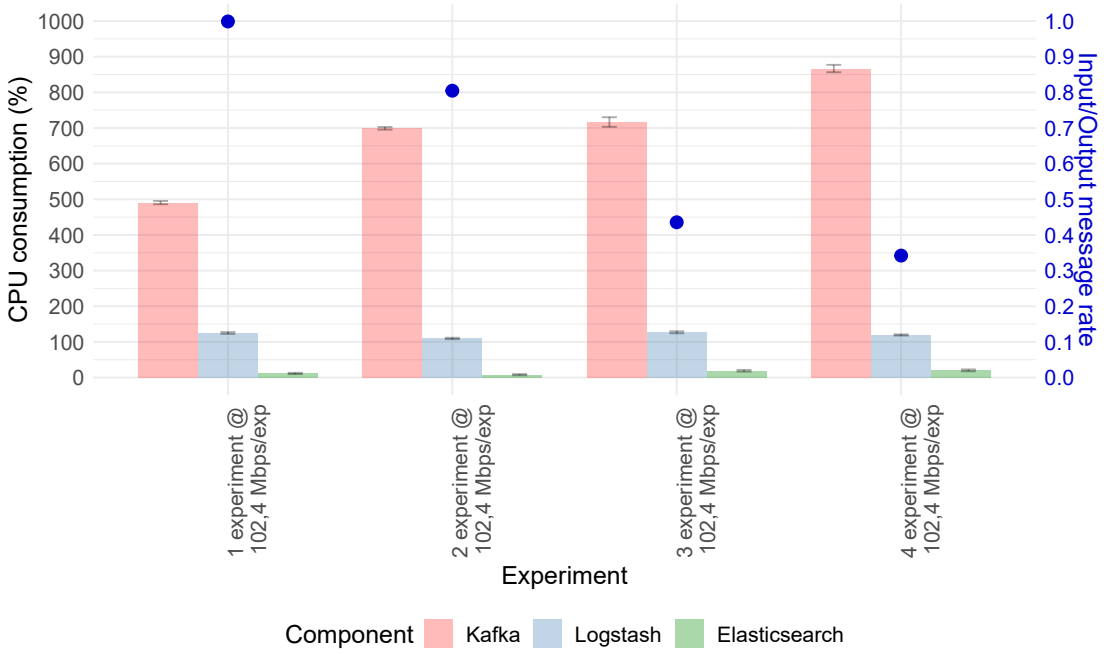

Figure 11: CPU consumption and I/O message rate evolution for 100 B data traffic in different experiments, modifying both number of experiments and total throughput in all cases.

Here, when the **number of experiments increases**, the **total throughput** is also **higher**, and in fact, it can be noticed that **message loss is present from two experiments** deployed, as the I/O message rate is nearly 0,8 (so the 20 % of messages are lost), and falling until less than 0,4 in the case of four experiments deployed simultaneously, value that **remains constant** even if more experiments are deployed (these experiments have not been included in Figure 11 just to present the saturation process with more detail).

The evolution of the **CPU consumption** in *Kafka* is **also stopped** due to this saturation state, as well as the **latency starts to present variations** as it is calculated based on the messages that are eventually received.

In fact, these results are quite aligned with the outcomes from [94], where it was reported that ***Kafka* throughput depends linearly on the number of topics**, reaching a **hard limit** at some specific point. According to this study, when there is only one *Kafka* replica, the limit is reached for around 15.000-20.000 packets per second, value which is close to these results, as one experiment in our testbed means around 16.000 messages per second and the deployment of a second experiment causes a loss of performance, since that limit, which should be between 16.000 and 32.000 messages per second, is exceeded.

3.4. Demonstration of the Platform in a Real Case Scenario

This Section shows how the Monitoring platform behaves in the execution of a **real experiment** in the 5G EVE platform, confirming that it works

properly according to the system specifications commented in Section 3.1.1 and the performance evaluation carried out in Section 3.3.

In particular, all the steps followed to design and execute an experiment are summarized in [5], which presents the testing process of a vertical service composed of a web server and a client in the 5G EVE platform. However, only the **workflow** related to the **Monitoring system** will be presented here, according to the **four major phases** in which the **experiment lifecycle** can be split: (*i*) experiment design and definition, (*ii*) experiment preparation, (*iii*) experiment execution and (*iv*) experiment results analysis [81].

3.4.1. Experiment Design, Definition and Preparation

Prior to the experiment execution, the **experiment design and definition** phase is focused on **planning and formalizing the experiments** for a given **vertical service**, identifying aspects like the components that compose the service, their interactions, the target execution environment(s) and condition(s), the elements to be monitored and the related KPIs, or the detailed steps to run the experiment [81].

The outcome of this stage is the definition of a set of **blueprints** (*i.e.* a sort of *templates*) that define the high-level features of the service. These descriptors also include several variable parameters that can be specified in the **experiment preparation** phase for **tuning** the specific **experiment instance**, obtaining as a result the **descriptors** that are used for the experiment instantiation.

There are different **types of blueprints**, depending on the information related to the experiment they contain: vertical services (*Vertical Service Blueprint*, VSB), contexts (*Context Blueprint*, CtxB), test cases (*Test Case Blueprint*, TCB) and experiments (*Experiment Blueprint*, ExpB) [95]. Regarding the **metrics and KPIs** to be monitored by the Monitoring system, these are defined in the VSB, CtxB and ExpB in the case of metrics (depending on the metric type, which can be infrastructure or application metric), and only in the ExpB in the case of KPIs.

Then, once the experiment is defined and on-boarded in the platform, *i.e.* **ready to be executed**, the ***Experiment Lifecycle Manager*** component of the 5G EVE platform **gathers** the information related to the **metrics and KPIs** from these blueprints and **sends** it to the *Data Collection Manager* through the **signalling topics** [96]. The process of building the **messages** sent to the DCM can be seen in Figure 12.

After receiving all the messages, the DCM performs the **creation of the topics** in the platform, sending also a **notification to the DCS-DV** in order to **enable the consumers** that receives the monitoring data in each topic and to **create listeners** that trigger the **generation of monitoring dashboards** when the monitoring data is received in the DCS-DV. As a result, the Monitoring platform is **ready to start consuming monitoring data** once the experiment starts its **execution**.

3.4.2. Experiment Execution and Results Analysis

During the **experiment execution** phase, the **scripts and applications** to run the experiment are **executed**. In this stage, the **Data Shippers** are in charge

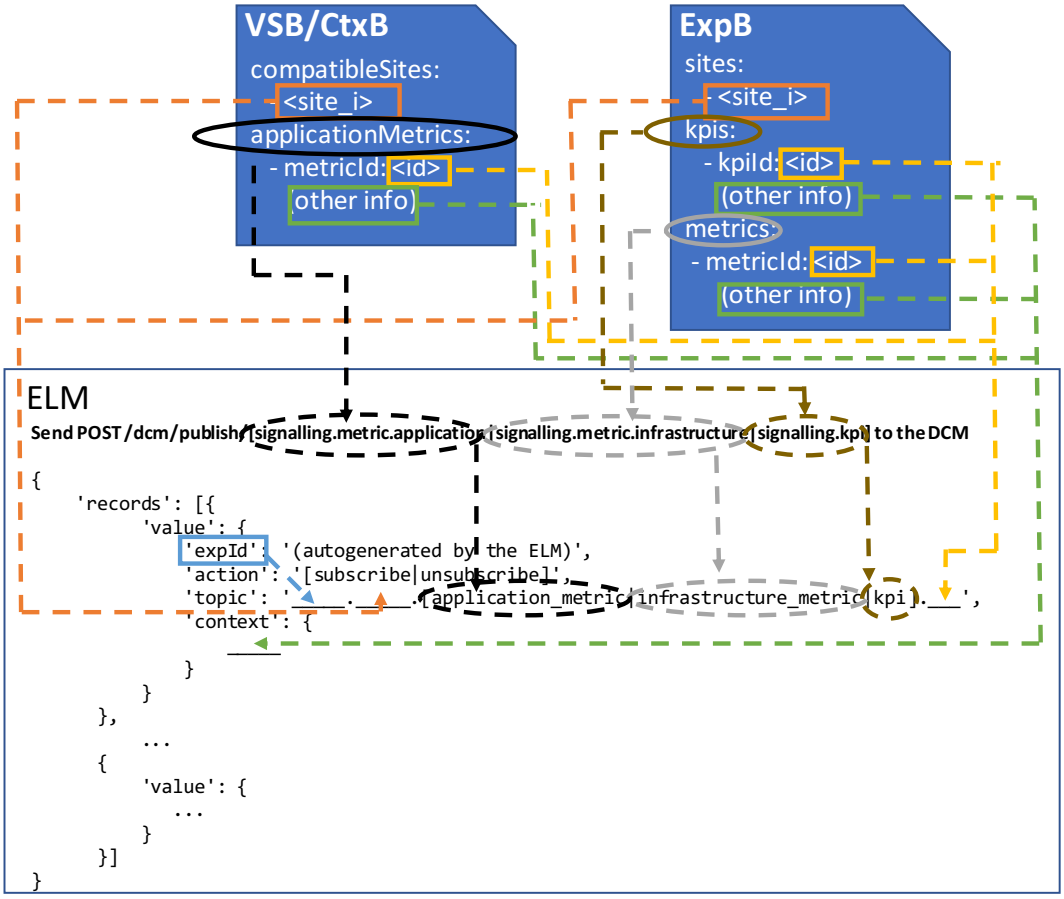

Figure 12: Blueprints used for collecting the information related to monitoring data and generaetion of the messages sent to the DCM.

of **publishing the generated metrics** in the Monitoring system through the corresponding **topics**[4]. Consequently, the **DCM delivers** the **metrics** received to the **RAV** (which **calculates the KPIs** associated to the metrics received and publishes them again in the system) and the **DCS-DV** (which **collects the data** and enables the **dashboards** to visualize them). The full workflow can be reviewed in [69], which contains its latest update.

The **dashboards** offered by the Monitoring system, combined with the **KPI validation process** performed by the 5G EVE platform, are the main outcomes provided for the **experiment result analysis** phase, which is not a stage as is, but it involves the **interpretation and validation of the results** obtained during the experiment execution. As an example, Figure 13 shows the **dashboards** of a **metric** (*request_time_taken*) and a **KPI** (*time_taken_per_request_kpi*) that belongs to one of the experiment executions of the vertical service presented in [5], showing their **evolution over time**.

[4]The Data Shippers must know in advance the topics in which they have to subscribe, information that can be provided as Day-2 configuration by the Runtime Configuration component from the 5G EVE platform [74].

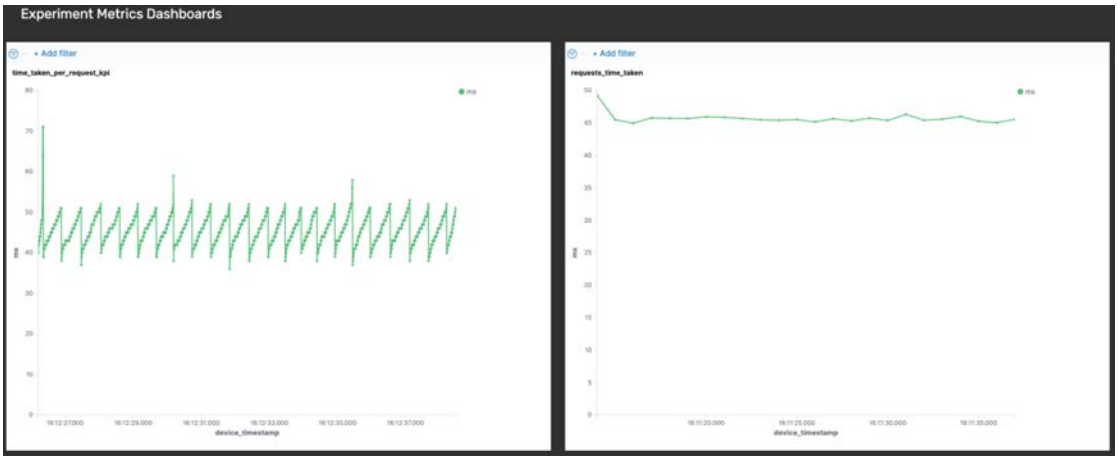

Figure 13: Dashboards that present the evolution over time of a metric and a KPI that belongs to a given experiment.

3.5. Summary

To sum up, this Chapter has presented a **modular Monitoring architecture**, proposing an **implementation** based on the **publish-subscribe paradigm**. Moreover, it has been shown that this architecture is able to monitor a **real experiment** with the results commented in Section 3.4.

Furthermore, the performance evaluation process performed on Section 3.3 has revealed some interesting **insights** related to the Monitoring architecture. The first one is that the **distribution** of the **performance parameter values in topics of the same type** is **uniform** for single-broker experiments, allowing the **aggregation** of the performance values obtained for each topic of the same type and, as a result, simplifying the study of the overall system.

In these single-broker experiments, it has also been detected that the **total throughput** is the parameter that can cause the **greatest impact on system performance**, with **two different possibilities**: while the system has enough **free resources** to work, the **CPU consumption tends to increase exponentially**, keeping **batch write latency and I/O message rate constant**. However, when the system is **saturated**, which seems to happen for a total throughput between 16.000 and 32.000 packets per second, this **exponential growth** is **stopped** and the **I/O message rate fails below 0,4** in the worst case.

4

Towards a Distributed Monitoring Framework for Beyond 5G Networks

The work presented in Section 3, in terms of the application of **monitoring** processes in **5G networks**, has revealed the need for a **flexible and efficient monitoring system** capable of realizing **5G multi-site and multi-stakeholder scenarios**. This kind of platforms, like the Monitoring framework already presented, allows 5G to support the **optimization** of the **network performance** and the **support** for the **requirements and KPIs** defined for each type of **use case** accommodated in the system.

Nevertheless, for the **evolution** towards the so-called **Beyond 5G communications**, focusing on the **next-generation 6G systems**, it has to be noticed that these networks will **not only be Cloud-centric**, which is a paradigm already present in 5G with examples such as Edge Computing or Cloud Radio Access Networks (C-RAN). However, they will likely become fully **Edge-centric, flow-based networks**, with a **reduced relevance of core data centers** in order to move towards more **flexible deployments** based on the **Edge** [97].

Being more precise, in relation to **3GPP standardization** from the current 5G system architecture perspective, the **Release 17** is the phase in which it is expected to see different **improvements** to be adopted for Beyond 5G networks. Among those features, whose availability may arrive in mid-2021 as a first tentative date, we have the enhanced support of **IIoT, NPNs, wireless and wireline convergence, multi-access Edge Computing or network automation** [98]. These enhancements imply, consequently, the support for a **diverse catalog of use cases** in a **flexible, federated, secure and reliable way**, in a network composed by **multiple network environments** [97].

Then, assuming that the **Edge Computing** concept, together with the current specification of **5G networks** and the vastly increased number of **data processing devices** at the Edge, will be the **base of Beyond 5G networks** [99], these next-generation networks could be envisioned as a **user-driven, distributed Cloud Computing system** where the **resource pool** is foreseen to **integrate the participating users**. In this context, current **monitoring**

solutions are **limited**, as they have to be able to maintain 5G performance in a distributed system with heterogeneous resources and still be efficient and sustainable.

This Section **proposes the application** of the **Monitoring framework** presented in Section 3 as a possible **solution** to monitor vertical deployments not only in 5G networks, but also in **Beyond 5G scenarios** based on **multi-site, Edge deployments**, where different stakeholders may share the resource pool in a distributed environment. Taking advantage of the usage of **publish-subscribe** mechanisms adapted to the Edge, this adaptable Monitoring solution should **fit and scale in different network deployments**, **anticipating new requirements** that Beyond 5G networks may impose, such as **efficient resource utilization** or the **processing of large amounts of real-time traffic** generated by the applications located in the resource pool.

Along the above lines, the following **topics** will be discussed in this Chapter:

- Firstly, Section 4.1 identifies the **requirements** that must be taken into account in Beyond 5G scenarios, to be met by the Monitoring architecture already presented in Chapter 3 when integrating it in Beyond 5G deployments.
- In Section 4.2, the general **design** and **implementation** of the Monitoring platform are **reviewed** in order to check whether it fits in Beyond 5G scenarios, and also proposing **extensions** to the architecture to **integrate** new technologies and paradigms also present in these next-generation environemnts, such as **data analytics**.
- Then, Section 4.3 presents the **performance evaluation** of an **evolved** version of the **testbed** that holds the Monitoring platform, testing both **single-broker** and **multi-broker** configurations in a scenario where different **constraints** related to **Edge deployments** are imposed.
- Finally, Section 4.4 **summarizes and concludes** all the work presented in this Chapter.

4.1. New System Requirements

To tackle the integration of the Monitoring platform in Beyond 5G scenarios, it is **not enough** to fulfil the **initial requirements** imposed to the platform, already presented in Section 3.1.1, but it is also necessary to accomplish the **requirements** related to these **next-generation networks**.

As Beyond 5G and 6G communications are currently **under research** from both the academia and the industry, being mostly a declaration of intentions on how new networks should be built, these requirements have to be considered in a **higher level of abstraction**, rather than providing specific KPI figures to be achieved by the platform.

According to the current **literature** related to Beyond 5G and 6G networks from the state of the art [98]–[97][100][99][101], where the analysis of Beyond 5G and 6G system requirements, challenges and possible deployments is introduced,

the new **requirements** for this next-generation network architecture are the following:

1. **Alternative compute architecture:** the current **Cloud-based architectures** are **not enough** in this type of networks. As stated in the introduction of this Chapter, the move to **Edge deployments** must be a reality, achieving the building of **distributed computing and communication** resources through **federated network control** and **orchestration solutions** in these environments. The **flexibility** and **adaptability** to a wider range of **use cases** must take center stage.

2. **Service-based design:** the network should also provide **common service discovery** and **monitoring functions** in order to allow **network functions** to **easily interact** between them, also **facilitating** the introduction of **new features** in the future by utilizing existing services.

3. **Self-operation with less human involvement:** this feature is related to the **enhancement** of **network automation**, which can be enabled with the introduction of **AI and ML technologies** in order to allow the network to **scale on demand** and enable the **self-evolving capability** of network functions based on **data analytics** and **predictive models**. This may drive the need for a new architecture that is **AI-native** and **data-driven**.

4. **Global access:** this not only involves the **integration** of **multiple and different use cases**, but also it requires the **access** to the network **anywhere and anytime**, transforming the network in order to **simplify and unify** this access.

5. **Ultra-scalability:** evidently, scalability must be present, taking into account the **exponentially growing** of the **number of devices** connected to the network. In this way, the amount of data and information managed by the network will explode, becoming impossible to use traditional data collection, storage and analysis technologies to deal with this issue. Consequently, the network should be designed to be **distributed and flat**, **extending network functions** to the **Edge** to **improve the scalability** of the system. For example, the AI/ML algorithms aforementioned can be used in the Edge to enable the evolution of the MEC towards an AI-enabled platform that is able to offer intelligent services delivered over the fixed or mobile access network to the Edge devices.

6. **Sustainability:** this topic aims at **reducing the footprint** of current deployments in terms of **power consumption** or **computing resources' allocation**, among others. The **limitation of the resources** used in **Beyond 5G systems** is, consequently, a fact that must be considered in the design and implementation of platforms to be integrated in this ecosystem.

7. **Respect for user privacy:** the guarantee of **data confidentiality** is also becoming a topic of interest in mobile networks, which are currently subject to **regulation** in order to allow the user to decide how to handle their

personal data. This must be an instrument to put the **user** in **full control** of their **data**, selecting to whom their activity data is given or to which statistics it can be incorporated, among other actions. These **security issues** must be also present in the design of the Beyond 5G and 6G systems.

4.2. Revision of the System Design and Implementation

As already stated in Chapter 3, the 5G EVE complete platform, based on **multiple sites** with **heterogeneous components** generating useful data that is likely to be monitored, relies on a flexible and distributed **Monitoring service** in charge of **collecting** that **monitoring data** and **distributing** it to specific entities that obtain **insight**s about the behaviour of these components.

In this sense, a **general-purpose** Monitoring architecture like the one proposed in Chapter 3 is desired, so that it can also **fit** in **other similar scenarios** to the 5G EVE one, with regard to **multi-stakeholder environments**; for example, by introducing it in **Beyond 5G** scenarios.

To achieve this, the Monitoring architecture and its implementation based on the publish-subscribe architecture must be **reviewed** in order to confirm that it is able to meet the requirements and constraints of the **Beyond 5G** deployments, already presented in Section 4.1, also proposing possible **extensions** of the architecture that may be useful for fulfilling specific **Beyond 5G** requirements, mainly related to the achievement of **network automation** by introducing **AI-based data analytics techniques**.

4.2.1. Adaptation of the Architecture

The **architecture** presented in Section 3.1.2 is suitable for being integrated in **Edge environments**, meeting the **requirement** (1) proposed in Section 4.1. In Figure 14, the Monitoring architecture is presented again, but **emphasizing** the **components** that can fit in either the **Edge** or the **Cloud**.

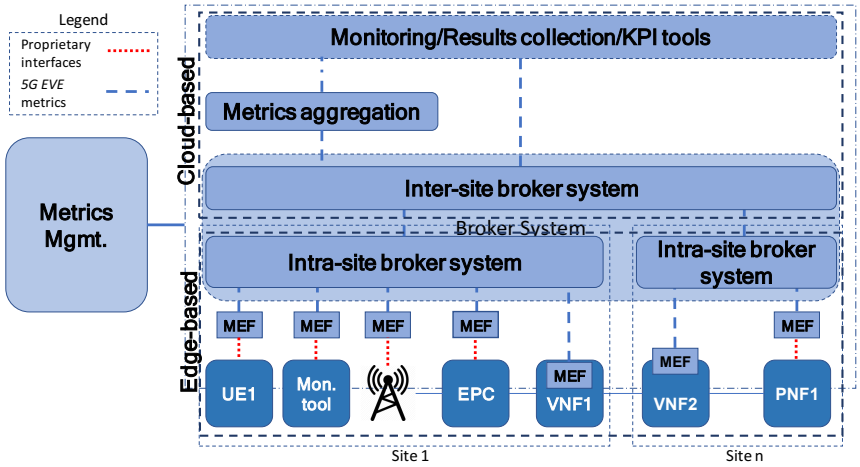

Figure 14: Monitoring architecture, highlighting the components to be deployed in the Cloud and in the Edge.

With this approach, it is clear that the **sites** would be **Edge-based**, making the **process** of **integrating new site facilities** (*i.e.* new stakeholders) in the platform **easier** due to this deployment mode, facilitating the **global access** to the platform to meet **requirement** (4).

This implies the **adaptation** of the ***Broker System***, which is the core of the Monitoring platform, in order to **fully integrate** the ***Intra-site broker system*** within **Edge environments**. For doing this, the **resource consumption** must be limited for the sake of **sustainability**, related to **requirement** (6). Moreover, as each *Intra-site broker* only manages the **data** related to the **Edge location** in which it is deployed, the **security policies** to be applied to the monitoring data can be **different** for each site, **adapting** them to the **user needs** to achieve **requirement** (7).

This adaptation is also required for the ***MEFs***; but this module, from a conceptual point of view, is **flexible enough** to be integrated in different environments, from on-premises deployments to more agile facilities such as Edge environments. Its **lightweight philosophy** is a potential enabler for **embedding lots of devices** in the platform, while maintaining the **scalability** of the system, according to **requirement** (5).

Furthermore, the **other components** of the architecture can still be deployed in a **Cloud-based mode**, but this does not prevent them from being **also integrated in the Edge**. In any case, it is clear that it is a **service-based platform (requirement** (2)), defining **high-level building blocks** that interact between them and that can be **extended** with new functions, without requiring the modification of the other blocks. One of the **potential extensions** that may be added to this platform is the inclusion of technologies and paradigms oriented to the **network automation**, that would allow the platform to achieve the **requirement** (3). This topic will be addressed in the next Subsection 4.2.2.

4.2.2. Extensions to the Architecture

As commented in the last section, and also according to the definition of the *Monitoring/Results collection/KPI tools* entity that belongs to the Monitoring architecture, presented in Section 3.1.2, this block may encompass some other components that take the monitoring data received from the *Broker system* as input and provide **value-added capabilities** to the Monitoring platform.

In particular, this Section is focused on the **integration of the Monitoring platform** in the **AI-driven Data Analytics framework** proposed in [3], whose block diagram is presented in Figure 15:

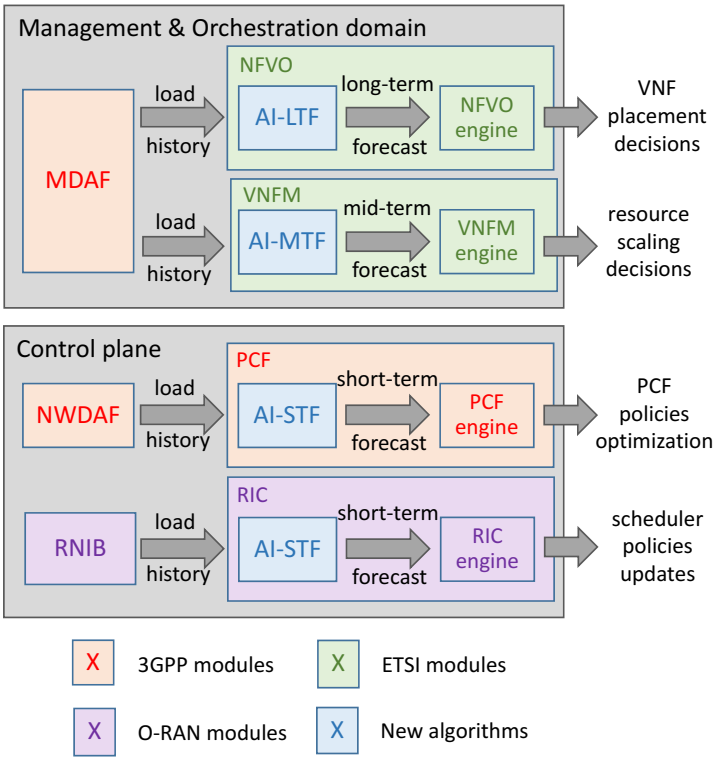

Figure 15: A network data analytics framework proposal with AI-driven capabilities.

This framework supports an **autonomous and efficient control, management and orchestration** of **mobile networks**. To achieve this, it relies on an **efficient collection** of **data** in the **network infrastructure**, together with the **knowledge inference** from these data. While the **second point** is performed by the **AI-based algorithms** proposed in the framework (*i.e.* AI-LTF, AI-MTF and AI-STF), there is **no specific proposal** to cover the **data collection** process.

A possible solution for this would be the **integration of the Monitoring platform** in this system, as proposed at the beginning of this Section, as it perfectly suits the problem of **gathering monitoring data from different sources** (in this case, the MDAF [15] module from the Management and Orchestration plane and the NWDAF [102] and the RNIB [16] modules from the Control plane) and **delivering it to the interested entities** (*i.e.* the AI-driven algorithms).

The **joint architecture** of both Monitoring and Data Analytics framework is presented in Figure 16, where there is **one particular *MEF* module for each data source**, in charge of **providing the network metrics** to the ***Broker system***, which consequently **delivers the metrics to the proper algorithm engine**.

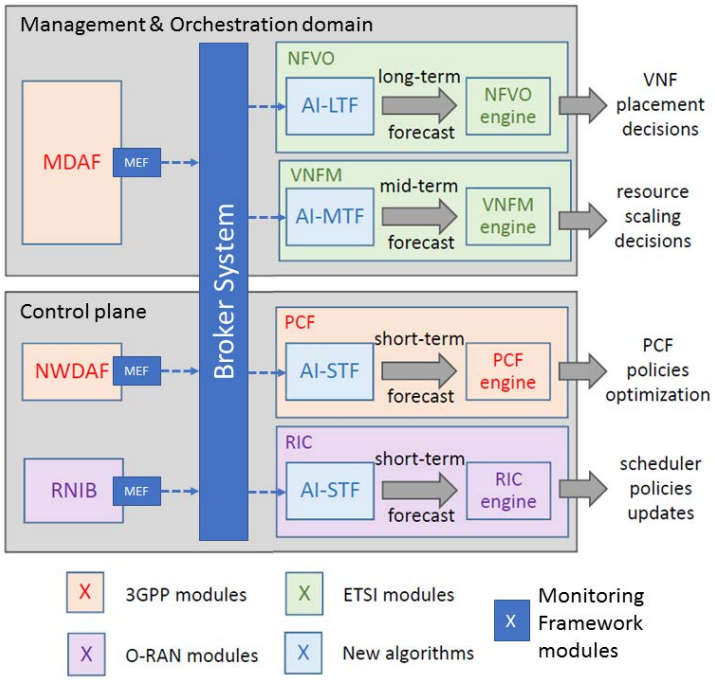

Figure 16: Enhanced network analytics framework with the integration of the Monitoring framework for data collection purposes.

In this way, the Monitoring framework could help to provide **a consistent way** to **obtain and transport network metrics**, **regardless** of their **origin and destination** (*i.e.* 3GPP, ETSI or O-RAN modules, among others), leaning on the broker-based architecture proposed. This allows the Monitoring system to deal with the different **granularity** of the network metrics handled, in terms of **traffic volume** (at global, slice or flow levels) and **timescale** (intervals of hours, tens of minutes, minutes or shorter), due to its flexible nature.

4.2.3. Implementation Update

As stated in Section 2.1, the **publish-subscribe mechanism**, which is the paradigm selected by 5G EVE to implement the *Broker System* from the general Monitoring architecture, is currently used in **different solutions** related to the **distribution of data** in **Edge environments**, so that it is a **good alternative** to be used in this kind of environments.

As a result, this feature allows to **deploy small processes** in the sites that only **gather monitoring data** and **forward** it to **upper layers** of the platform, being then aligned with Edge's philosophy. This can be achieved by the correct **coordination** between the ***Site Brokers*** and the ***Data Shippers***.

In the case of these ***Data Shippers***, and in the same way that happened with the ***MEFs*** in the general architecture, they can be also deployed in a **wide variety of environments**, from Edge to Cloud, according to their **flexible nature**, making the **transition** to an **Edge** environment **smooth**, consequently.

4.3. Performance Evaluation

To **assess** and **validate** the **suitability** of the Monitoring framework implementation for **Edge environments**, the **testing process** described below has been followed, based on the application of a **top-down approach**, starting with **single-broker experiments** to characterize the platform in terms of several **performance parameters**, and finishing with **multi-broker experiments** to check the **impact** of having the **two brokering levels** described in Section 3.1.

4.3.1. Testbed Setup

The **testbed** used for the evaluation of the architecture is an **evolution** of the testbed presented in Section 3.3.2. It consists on a set of *Ubuntu Server 16.04* **virtual machines (VMs)** [90] deployed in a server located in the **5G EVE Spanish site facility, 5TONIC** [88], using *Proxmox* [89] as virtualization environment, and *K3s* (a lightweight *Kubernetes* distribution) [103] to orchestrate the containerized components[5] deployed in each VM. This server is equipped with 40 Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20 GHz and 128 GB RAM. The distribution of components in each VM can be seen in Figure 17.

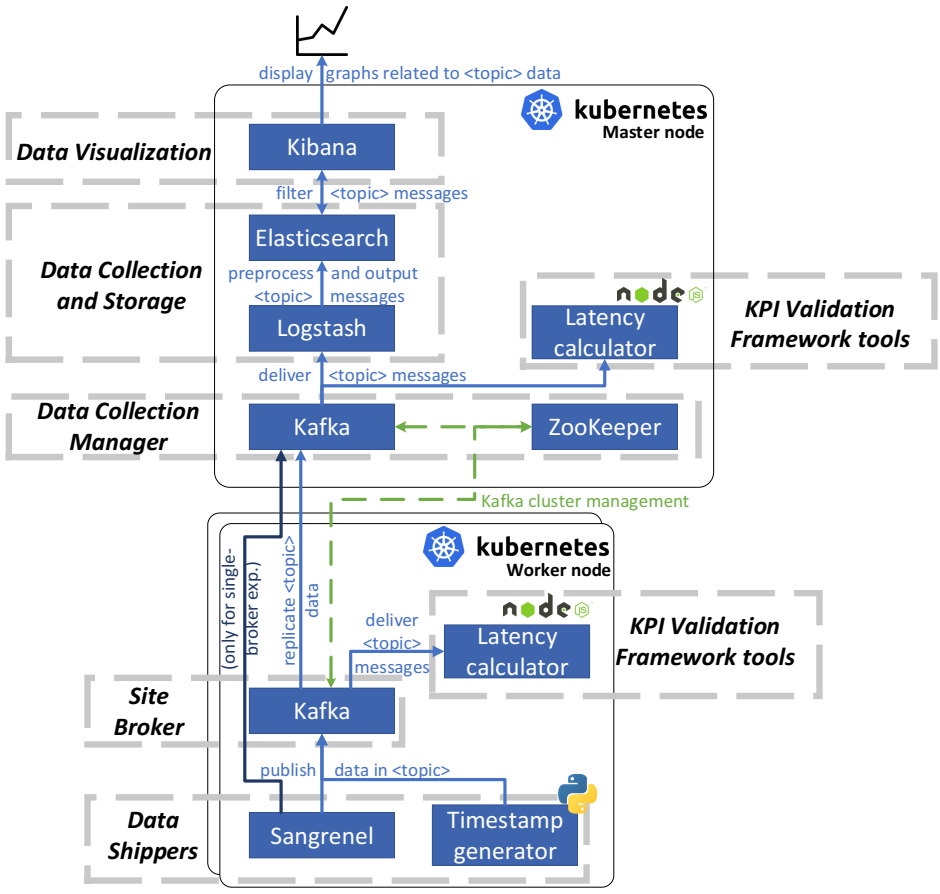

Figure 17: Testbed architecture.

[5]The images of these components can be found in [104].

The proposed scenario simulates the **monitoring** and **data collection** process of the **metrics** and **KPIs** related to a set of **experiments**. The components deployed in each VM are the following:

- ***Kubernetes* Worker node VMs: each *Kubernetes* worker** emulates a **site**, including ***Data Shippers*** for **publishing monitoring data** in a ***Site Broker***, based on *Apache Kafka*, that **replicates** the data towards the ***Data Collection Manager***, placed in the *Kubernetes* Master node. Regarding the ***Data Shippers***, this role is played by two components:
  - ***Sangrenel*** [91], the *Kafka* cluster load testing tool already presented in Section 3.3.2. The **performance parameters** obtained, each second, from this tool, are the **input message rate** (used for calculating the **Input/Output (I/O) message rate**, *i.e.* the received throughput divided by the publication rate) or the **batch write latency** (*i.e.* time spent until receiving an ACK message from the broker).
  - A *Python*-based ***Timestamp generator*** [105], used exclusively in **multi-broker experiments**. It sends **messages with timestamps** embedded that are eventually received by a ***Latency calculator*** component, based on *Node.js*[6] [107], which takes the timestamps and **calculates** the so-called **broker latency**, *i.e.* time spent between the publication of data and its reception in an entity subscribed to the *Site Broker*. In fact, this component can be associated to the ***KPI Validation Framework tools***, as it calculates the latency (KPI) based on timestamps (metric).
- ***Kubernetes* Master node VM:** in this server, the ***Data Collection Manager*, *Data Collection and Storage* and *Data Visualization*** components from Figure 4 have been implemented with a solution based on *Apache Kafka* and the *Elastic Stack*. A ***ZooKeeper*** [76] instance is also running to **coordinate the *Kafka* cluster**, and there is also another instance of the ***Latency calculator*** deployed here to calculate the **end-to-end latency** KPI, this being the time spent between the publication of data in a given site and its reception in an entity subscribed to the *Data Collection Manager* (so that data has been previously replicated from the *Site Broker*).

For monitoring the **resource consumption** of each container, ***Docker*** [92] **native tools** (e.g. *docker stats*) have been used.

4.3.2. Singe-Broker Experiments

For these experiments, only **one *Kafka* broker** is required, so the testbed depicted in Figure 17 can be simplified by only using **one *Kubernetes* Worker node** with just a ***Sangrenel* container** directly connected to that *Kafka* broker,

[6] This programming language has been used in order to make use of *Kafka*'s KIP-392 feature, to receive data from the closest replica [106].

represented with the dark blue line that connects both components in the testbed diagram.

In this way, this analysis **continues** with the **conclusions** extracted from the performance evaluation carried out in Section 3.3.4, related to the **saturation effect** observed in the last experiments performed. This issue must be also taken into account in order to introduce these **CPU-bound components** in **Edge environments**, where the number of physical and virtual **resources** allocated to execute these workloads are quite **limited**. In this way, apart from having a **theoretical limit** imposed by the technology itself, the **amount of resources** can also have an **impact** on **performance** in case of sizing the platform wrongly, provoking a loss of performance even before reaching the hard limit.

To reflect the **impact on performance** caused by the **limitation on computing resources** (*i.e.* vCPU allocation in the *Kafka* container), Figure 18 presents the evaluation of both the **batch write latency** (top subplots) and the **I/O message rate** (bottom subplots), for **100 B data traffic**, in **two situations**:

- First of all, assuming that a **full experiment** is being **executed** in the platform (*i.e.* a total throughput of 102,4 *Mbps* is received by *Kafka*), the **vCPUs** assigned to the ***Kafka* container** was **modified from 1 to 6** (the two graphs on the left in Figure 18); checking that, **from 5 vCPU**, the **values** obtained for the performance parameters become reasonably **good and stable**.

- However, on a scenario where the ***Site Broker*** is placed in the **Edge**, a **high resource allocation cannot be guaranteed**. For this reason, a new set of tests in which the **vCPU** allocation was **fixed to 1 vCPU**, then **varying the throughput** received by *Kafka*, was carried out (the two graphs on the right in Figure 18). The values used for the throughput vary **between the 100 % and the 10 %** of the throughput related to an experiment (*i.e.* 102,4 *Mbps*). The results reflect that, although the **latency does not improve** when a **lower throughput** is received, this is not the case for the **I/O message rate**, which **improves every time that throughput is reduced** until reaching a value of 1 when the throughput is reduced to the 10 %.

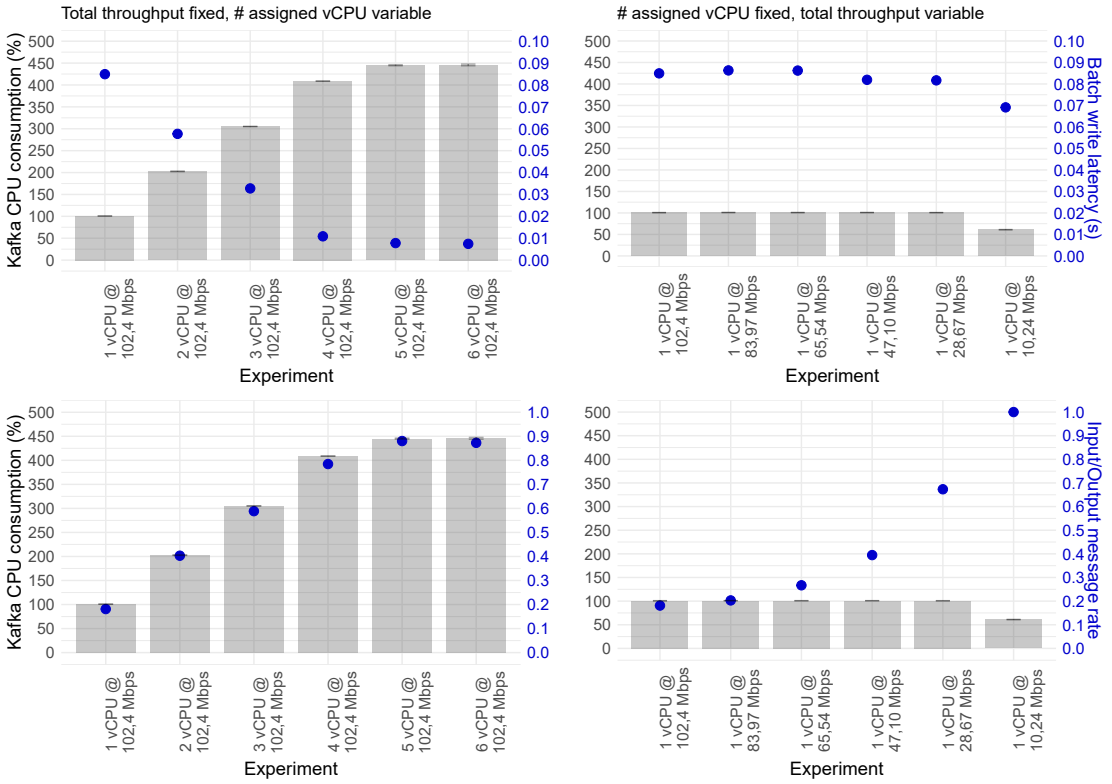

Figure 18: Effect of saturation in performance parameters when limiting *Kafka* vCPU allocated in different experiments.

Consequently, to move to an **Edge environment**, it is crucial to **limit the resource allocation**, but also the **throughput** received by the Monitoring platform, in order to avoid packet loss. This issue should not be a problem in Edge environments, assuming that most use cases deployed in this kind of scenarios will prioritize the ability to support a **large number of connections** rather than guaranteeing a certain value of latency or bandwidth; as happens in IoT, for example. Therefore, the **higher values of latency**, compared to the **ideal scenario** in which there are no problems related to resource consumption (**70** *ms* **vs. 10** *ms*, approximately), **should not be a problem** while the **throughput is kept at a reasonable value**. In this case, this limit can be set to **10** *Mbps*.

4.3.3. System Scalability Validation

To avoid this saturation effect, the direct solution is to build mechanisms and processes that allow **system scalability**, mainly oriented to the application of **horizontal and/or vertical scaling processes** depending on the **current status** of the platform. This kind of systems are useful for Edge environments, in order to **scale** the platform according to the **traffic demand** and the **resources available**.

For this evaluation process, a preliminary **vertical scaling system** for this Monitoring platform is proposed (*i.e.* no new instances are added, but the computing resources attached to the available instance are increased or decreased depending on the workload), based on the **results** obtained in the **previous tests**

as **training data**, used to refine the different cases that can occur in terms of resource consumption (mainly related to CPU) and performance evaluation (mainly based on the batch write latency and the I/O message rate), and the conditions related to each case that trigger the system scale process.

Figure 19 presents an example of **vertical scaling** for **one experiment** deployed in the platform. In this case, the *Kafka* container is scaled by **increasing its vCPU assignment** until the system is able to **handle the workload** received **without saturating**, decision that depends on different parameters, such as, *e.g.* the current CPU consumption, the delay to compute a KPI or some other performance variable.

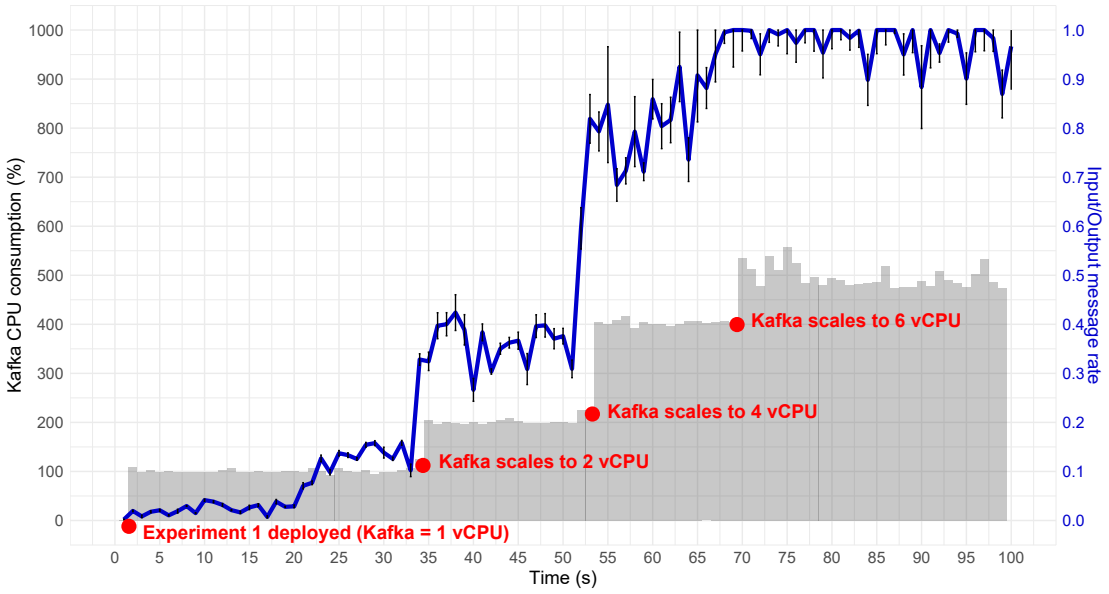

Figure 19: Evolution of the I/O message rate related to 100 B data traffic in one experiment when vertical scaling mechanisms are enabled.

Note that, in this case, for illustrative purposes, an **upscale** is only triggered when a **CPU is fully occupied** for relatively **long periods of times**, this resulting in a relatively high convergence time (around one minute) of the I/O message rate, but more "agile" schemes could be easily implemented if needed.

4.3.4. Multi-Broker Experiments

Finally, the **full multi-broker platform**, as built in the testbed already presented in Figure 17, will be evaluated in terms of the **performance parameters** already presented in Section 4.3.1 and the **CPU consumption** of the ***Data Collection Manager*'s *Kafka* broker**, whose **computing resources** will **not** be **limited**. On the other hand, the ***Site Brokers*** will be **limited to 1 vCPU**, taking the value already tested in the tests presented in Figure 18.

In this case, the **meaning of experiment** will be a bit **different**. This way, each experiment deployed in multi-broker experiments will be **executed in a particular *Kubernetes* Worker node** (so, for six experiments, six *Kubernetes* Worker nodes will be required), sending **monitoring data** to the corresponding

Site Broker **at 10 %** of the **total throughput** calculated in Section 3.3.1 (*i.e.* 10,24 *Mbps*), which is the **throughput hard limit to avoid saturation**, as stated in Figure 18.

The first performance parameter to be evaluated is the **latency**, in the **different acceptations** that were defined in Section 4.3.1: the **batch write latency**, the **broker latency** and the **end-to-end latency**. The **values** obtained during the execution of experiments, from one to six, for 100 B data traffic, can be seen in Figure 20. Here, a similar effect than the one obtained in Figure 9 can be observed: the results obtained in each site are **similar** for each case, so that performance data can be also **aggregated** in future analysis.

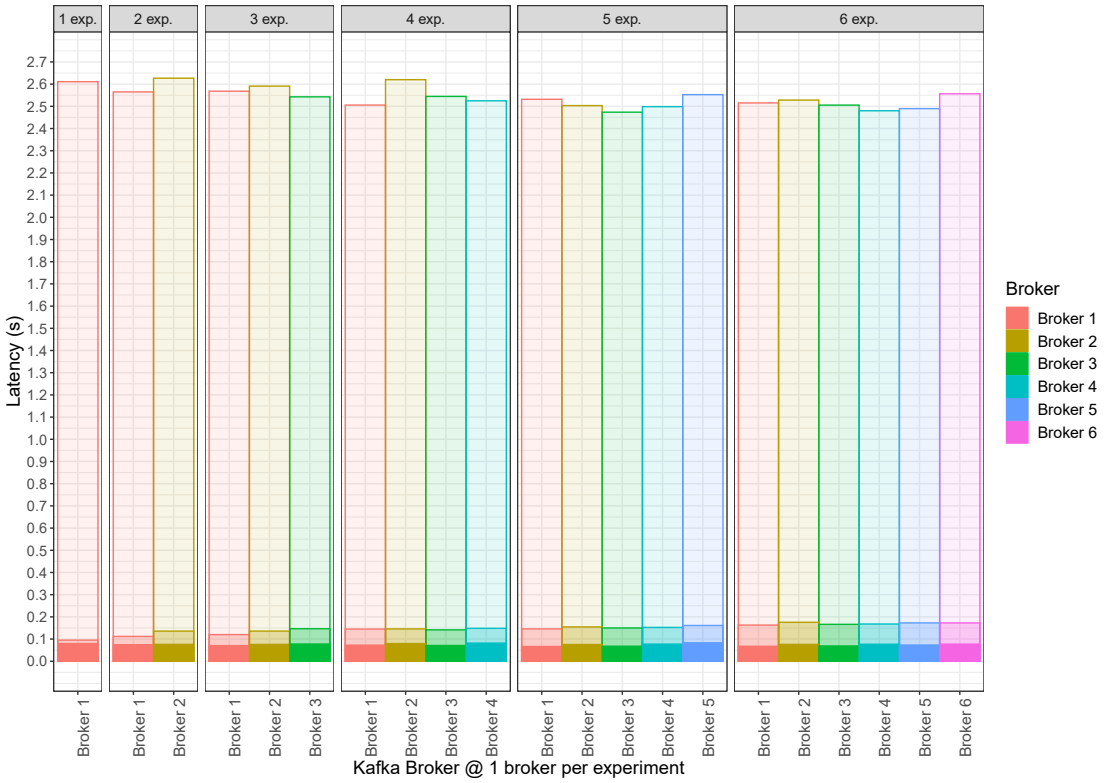

Figure 20: Evolution of the three types of latency in multi-broker experiments.

Moreover, the **same tendency** in latency values than observed in Figure 18 can be also seen here: the latency **does not vary** even though the **total throughput** received by the Monitoring platform **increases** due to the deployment of new experiments.

Furthermore, the **results**[7] obtained for each type of latency are **consistent** with the **definition** of each of them: it is expected that the **batch write latency** (the darker colour for each case) would give the **lowest value** (approx. **70–80** *ms*), as it only implies the **reception of the ACK** from the *Site Broker*. The next one would be the **broker latency** (the colour of "intermediate" darkness in the graph), in which the ***Site Broker*** has also to deliver the data to a **subscriber**,

[7]Note that these results have been obtained in a virtualized scenario, in which the latency between virtual machines and containers is negligible. In a real scenario, the delay introduced by each of the path components must be also take into account.

but it can be checked that this does not cause a great impact on latency, as it is increased to nearly **150** *ms* in the worst case. And finally, the **highest value** on latency (approx. **2,5–2,6 seconds**) is obtained for the **end-to-end latency** (the lighter colour in the graph), due to the **replication** operation performed between each *Site Broker* and the *Data Collection Manager* and also the **delivery** to the corresponding **subscriber**. This value can be **assumed in Edge environments** for the reasons aforementioned.

Finally, the **impact on the I/O message rate** in the **multi-broker experiments** is the **same** than experienced in **single-broker experiments with CPU limitation** (reflected in Figure 18), where the **packet loss increases** with the **increase** of the **total throughput** received in the platform. This effect can be seen in Figure 21, where the **performance results** from **different brokers** have been **aggregated** due to the results obtained in Figure 20.

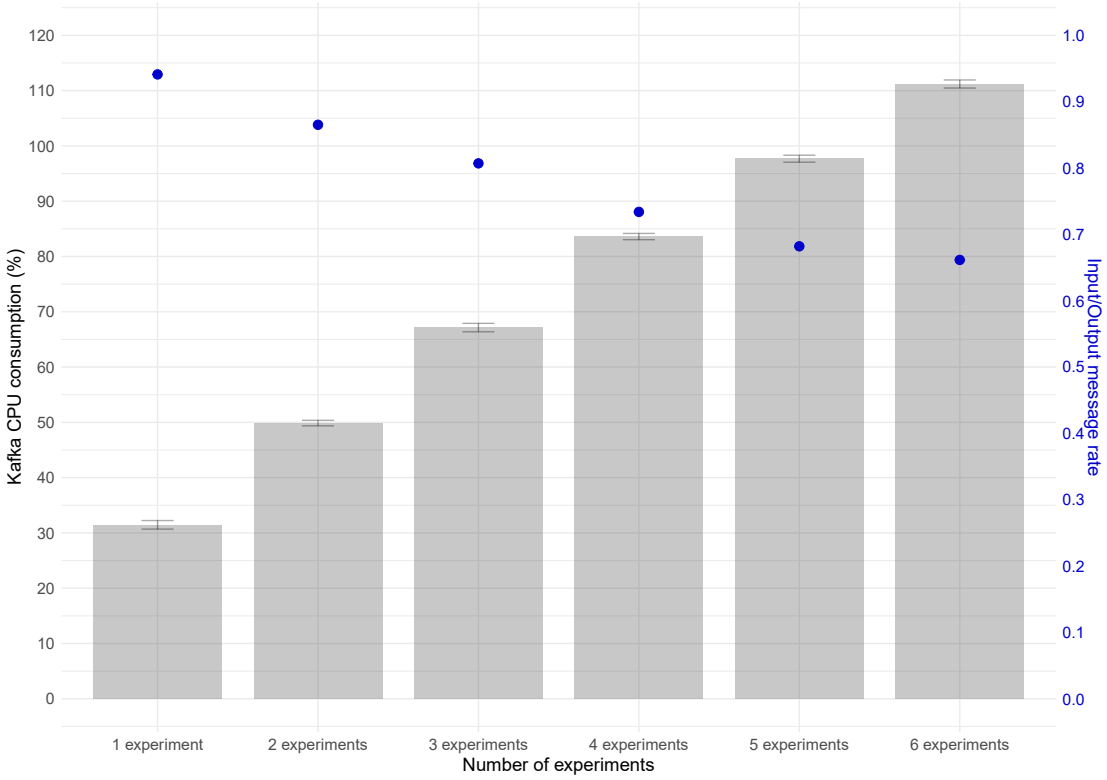

Figure 21: Evolution of Data Collection Manager CPU consumption and I/O message rate in multi-broker experiments.

It can be observed that **I/O message rate falls** to nearly **0.65** when the **six experiments** are being **executed concurrently**, meaning a **total throughput** received of **around 60** *Mbps*. This result, compared to the case observed in 18 with a single broker, with 1 vCPU, consuming 65,54 *Mbps* (the I/O message rate was less than 0.3), implies that the **distribution of the total throughput between several *Site Brokers* improves the results**.

Moreover, the **CPU consumption** in *Data Collection Manager*'s *Kafka* broker also **increases with each experiment**, but in a **less rate**, reaching the **110 % of vCPU consumption for six experiments**. Consequently, although the core

of the Monitoring platform is expected to be executed in environments without limit of computing resources, this final result may allow the **deployment** of some **components** of this **core** (*e.g.* the *Data Collection Manager*) on the **Edge**; as long as the total throughput, again, does not exceed a specific limit that causes saturation (60 *Mbps* in this case).

4.4. Summary

In this Chapter, the modular **Monitoring architecture** already presented in Chapter 3 has been **extended**, in terms of **design and implementation**, to meet the requirements and expectations related to **Beyond 5G scenarios**, with the specific example of **Edge environments**. In fact, it has been showed the platform's ability to be easily adapted when integrating new, **advanced features**, such as the interaction with AI/ML techniques for data analytics purposes, as commented in Section 4.2.2.

Regarding the **performance evaluation** of the platform, focused on these new kind of network deployments, it has been shown that the platform is able to manage real, complex experiments in both **single-broker** and **multi-broker configurations**.

After detecting the possibility of **saturating the platform** in specific conditions, the analysis of the **performance parameters** when the **computing resources allocated** (*i.e.* the vCPU) are **limited** revealed that the system can **reach the saturation state even before** that the **theoretical limit** provided in Section 3.3.4.

This constraint can be regulated with the **modification of the total throughput** injected in the platform, allowing to **increase the I/O message rate by reducing the throughput**, while **maintaining lower resource's usage** and a practically **constant latency**. This is particularly important in the **transition** towards **more flexible deployment** such as **Edge-based environments**, in which resource's consumption is a crucial issue to be tackled. Furthermore, these results were used to build a **preliminary vertical scaling mechanism**, which calculates how many resources are needed for a given workload.

Finally, in **multi-broker experiments**, the impact of **deploying several experiments**, consequently involving the **joint activity** of **different *Kafka* brokers**, was **evaluated**, checking that the **latency**, in its different variants, remains also **constant**, being then the **I/O message rate** the **performance parameter to be optimized** by **adjusting** again the **total throughput** received by the platform, issue that should be **easy to solve in Edge environments**, where latency and bandwidth are not as important as a **flexible deployment** of solutions to ensure a **lower consumption**, allowing the connectivity of a huge set of devices to a given platform.

Summing up, based on the results obtained after this performance evaluation process, it has been confirmed that this Monitoring platform is able to **scale** in **multi-site scenarios**, enabling also **lightweight deployments** oriented to **Edge and Beyond 5G deployments**.

Part 2. Orchestration of 5G Transport Networks

5

Novel Network Orchestration Techniques Based on SDN and Intent-Based Capabilities

Since its conception, the **5G System architecture** has been designed to **enable** the use of techniques such as **Network Function Virtualization (NFV) and Software Defined Networking (SDN)** [55], in order to provide scalability, flexibility, agility and programming capabilities to the multi-tiered 5G mobile networks [108]. These technologies are expected to tackle some key **challenges** in terms of the **5G network infrastructure and traffic management**, where the application of **network slicing** is crucial to achieve a real multi-tenant architecture. Some of them are related, mostly, to the **management of** physical and virtual **resources**, the **end-to-end service orchestration** of services and the **end-to-end connectivity** services.

There are also **other technologies** that can help to cope with these challenges. For example, in terms of **orchestration capabilities**, **embedded monitoring systems** in **orchestration solutions**, like the proposal made in Section 4.2.2, can allow the usage of **measurement-based orchestration algorithms** to **improve system's performance**.

Regarding the application of **SDN** in 5G networks, it can enable the **programmability** of these **connectivity services** provided by 5G networks, thanks to the **dynamic** configuration and management of **traffic flows** [108]. Moreover, SDN also allows to **move from** a networking infrastructure mostly based on **proprietary and specialized hardware**, to a **SDN-friendly switching and routing equipment**, thus **reducing costs** while **gaining** the **flexibility** provided by the development and deployment of network applications regardless of the underlying infrastructure.

The **model** of the **5G transport network** to be managed and controlled by **SDN** can resemble the **Wide Area Networks (WAN) environments** in which SDN have also been introduced, with the so-called **SD-WAN solutions**, for its operation, administration and management. Consequently, the 5G transport

network can be seen as a set of **separate network domains**, which are defined depending on the scope of each of them, and whose **connectivity** is managed by an **SDN-based solution**. Furthermore, there may even be **domains** that are **external** to the **SDN control** (*i.e.* based on legacy routing protocols such as BGP or OSPF) that may **need to communicate with these SDN domains**.

Then, a desirable SDN-based solution to this kind of environments should be able to **understand and learn** from these **legacy routing protocols** used in domains that are external to the SDN domains, in order to correctly **forward the traffic** between the SDN domains (which acts as a transit network) and the legacy domains that are present on the SDN network's edge. Furthermore, taking advantage of the benefits of SDN and network virtualization, a **flexible management** can also be applied to the traffic to meet particular **quality of service (QoS) and service level agreement (SLA) requirements**, introducing the concept of **network slicing** for this purpose. Nevertheless, this niche market, in both WAN and mobile networks, is fully dominated by **proprietary solutions**, with hardly any competitive open-source solutions, causing considerable cost increases that are, in fact, contrary to the SDN philosophy.

To fill this gap, this Chapter presents **Alviu** [9], developed as a **flexible, resilient and Cloud-native SD-WAN orchestration solution** for **enterprise and academic networks** purely based on **open-source tools**, using **ONOS** as **SDN controller** [36], but that can also be a potential solution for **SDN-based 5G transport networks** due to the similarity between the two scenarios, as discussed above.

Alviu's **open-source nature**[8], together with its **compatibility with standard protocols** in the **SDN southbound API** (*e.g.* OpenFlow) and the application of the **intent-based networking paradigm** to manage the **lifecycle** of the **managed network** (as stated in Section 2.2.2), contribute to **reducing the cost** of network equipment and operational expenses while **abstracting the complexity of the underlying** physical and Cloud **infrastructure**, avoiding the risk of vendor lock-in through the support of commodity networking equipment (white-box switches) and x86 servers.

To describe and evaluate this platform, the following **topics** are proposed:

- Firstly, Section 5.1 presents how the **network infrastructure** is **modeled** from Alviu's perspective to fit in the **intent-based networking** approach proposed by this orchestration solution.
- Section 5.2 describes the **Alviu's modular architecture**, designed to offer a secure, easy-to-manage and centralized **control** over the managed **SDN-based domains** based on an **intent-based** operation in the orchestration part.
- Then, Section 5.3 details how **Alviu manages the intents**, including the specification of the **intent states** that are handled by it and the **transition between states**, which are involved in the **workflow** followed by the orchestrator.

[8] Alviu's code is not currently free, as it is a commercial product.

- Section 5.4 shows a **performance evaluation** to check the impact of this solution in terms of the **intent deployment time**, depending on the number of **SDN domains** to be managed, verifying that the system is able to handle a higher number of SDN domains without saturating.
- Finally, Section 5.5 **summarizes and concludes** this chapter.

5.1. Network Infrastructure Model

Alviu enables the integration of Cloud and network services through a **centralized and dynamic administration**, overseeing the 3 Cs in the network (Control, Communication and Computation) by following an **intent-based configuration at run-time**. In particular, this novel paradigm tries to **simplify the network management** for network administrators, providing an **easy-to-use REST API** to declare the **intents** that are used to specify, in a comfortable way, the **scenario** to be managed and the **configuration** to be achieved.

To achieve this, the **network infrastructure** managed and orchestrated by Alviu must be modeled in terms of **smaller, logical building blocks**, representing the entities that play a different role in the infrastructure. This abstraction exercise is depicted in Figure 22:

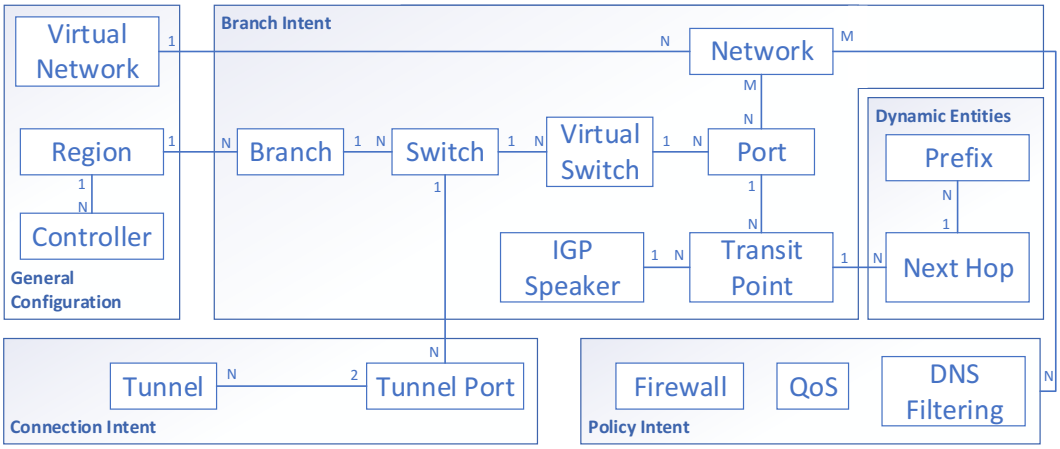

Figure 22: Network infrastructure model implemented by Alviu.

Firstly, there is a distinction between the **entities** managed by **intents** or **independent entities** that belong to the **general configuration** of the system. In the first case, the **Branch Intent** gathers the entities related to the **network equipment** and the **IP networks**, having the following **particularities**:

- The highest level component is the **Branch**, which represents a **SDN domain** composed by a set of SDN **Switches**. Each SDN switch can be composed by several **Virtual Switches** connected logically between them; which are, eventually, the components that **interacts with the SDN Controller** by using the OpenFlow protocol.

- The technology used to implement the virtual switches in the SDN switches compatible with Alviu is **Open vSwitch** (OVS) [109], the *de facto* standard used in switching solutions for virtualized environments.
- Each virtual switch can define a list of **Ports** that belongs to it, which can be **physical ports** of the SDN switch or specific-purpose **virtual ports** (*e.g.* to have a loopback interface for a service that may need it).
- The ports can manage **traffic** from two sources: (*i*) **SDN domains**, modeled with the **Network** entity, and (*ii*) **external domains**, characterized by a pair of entities: the **IGP Speaker**, which is the module, based on Quagga [47], used for the **communication with external domains**, and the **Transit Point**, which specifies all the **information** needed by the **IGP speaker** to establish that communication in a specific **port** of a given **virtual switch**, so that an IGP speaker can handle several transit points at the same time. This data is eventually translated into **Quagga configuration** to enable this exchange of information.
- Finally, from **each Transit Point**, all the **data** related to the **edge routers** of the **external domains** connected to the Transit Point is saved in the so-called **Next Hop** entity, from which all the **Prefixes learned** from these external domains are also attached while they are **announced**. Note that these two entities are **dynamic** and do not really belong to the Branch Intent itself, as they only appear if there is communication with an external domain (Next Hop entity) and if there are prefixes learned from them (Prefix entity). For this reason, they appear in a different block called **Dynamic Entities**.

The other two groups related to intents are: (*i*) the **Connection Intent**, only represented by the **Tunnel** entity, which represents the **WAN interconnection** between **two switches** of the same or different branch, identified by a pair of **Tunnel Port** entities, by using **tunneling protocols** like GRE or IPsec, and (*ii*) the **Policy Intent**, which groups a set of entities (*i.e.* **Firewall**, **QoS** or **DNS Filtering**) related to **policies** to be applied to a specific network.

On the other hand, there are three main **General Configuration** entities managed by Alviu: (*i*) the **Region**, which logically **groups a set of branches** that will be controlled in an homogeneous way by (*ii*) one or a cluster of **Controllers**, which are the **SDN Controllers** present in the network scenario, and finally, (*iii*) the **Virtual Networks**, which are used in order to **group network prefixes** that must be **interconnected**, even though they are used in different branches.

5.2. Alviu Orchestrator's Architecture

Considering the network model commented in Section 5.1, Figure 23 presents the **Alviu high-level architecture** operating in a typical network configuration, in which it manages a set of **SDN domains** (or **branches**) that may interact with a set of **external, legacy domains** or with **Internet** connections.

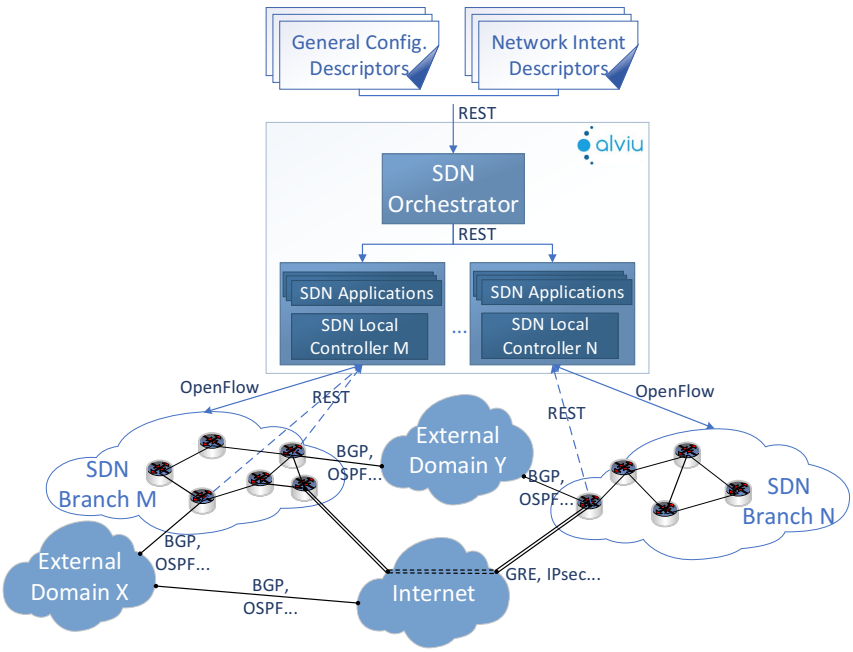

Figure 23: Alviu's high-level architecture.

Furthermore, Figure 23 allows to distinguish between the most **relevant parts** that compose the final solution, which are the following:

- **Descriptors:** input information, provided by the users of the platform (mainly network administrators), that serves to **fully define** the **managed network** and the **desired configuration**. There are two main types, aligned with the network model already presented: the **Intent Network Descriptors**, used for configuring the entities related to the intents, and the **General Configuration Descriptors**, related to the general configuration itself. For doing that, this data is provided by following a specific **data model**, codified in **JSON format** and provided to the platform through the **REST paradigm**. More details about the descriptors and their content can be found in Chapter 5.2.1.

- **Alviu:** the core of the solution, which can be divided in **two main components**, depending on the **level of abstraction** that is applied in each block:

  - **SDN Orchestrator:** this component is in charge of handling the **lifecycle of the intents and configuration** declared in the platform through the descriptors, performing all the required **actions and operations** (including calls to the SDN Controllers, the other block that defines Alviu) to meet the **requirements** defined in the intents.
  - **SDN Controller:** on the other hand, this module receives the **calls** from the SDN Orchestrator through specific **REST endpoints** enabled by a set of **SDN Applications**, which interact with the **SDN Local**

Controller to create, update or delete the **flow rules** that are installed in the **SDN switches**.

The full description of Alviu can be found in Section 5.2.2.

- **Network infrastructure:** by applying the network model described in Section 5.1, Alviu is able to control a set of **SDN switches**, distributed in **domains or branches**. Each domain can be controlled by a set of SDN Controllers (depending on the clustering configuration), using **OpenFlow** for the **switch-controller communication**. Moreover, the SDN branches can interact with other domains, as presented in Figure 23; for instance, different **SDN branches** can be **connected** to each other by using **tunneling protocols** (*e.g.* GRE or IPsec) through the Internet, or there can be configurations where a **SDN branch** is **connected** to one or more **external domains** (*e.g.* SDN Branch M connects External Domains X and Y, or SDN Branch N is connected to External Domain Y), in which case the **SDN switch directly connected to the external domain** must handle the **legacy protocols** used by the external domain (*e.g.* BGP or OSPF) to extract the **reachable networks** (and even routes to reach the Internet[9]) from that domain, providing it upwards to Alviu through REST[10] in order to properly install the **flow rules** in the SDN switches to correctly **route the traffic to/from external domains**. For this purpose, these edge SDN switches are equipped with Quagga, an open-source routing software suite that implements such legacy protocols in Unix systems [47]. This flexible configuration allows the definition of scenarios in which the SDN branches are more than isolated domains, but they can even act as **transit networks** that connects several external domains (*e.g.* SDN Branch M, which interconnects External Domains X and Y).

5.2.1. Intent-Based Networking Characterization

Network characterization is mainly declared by using the **Network Intent Descriptors**. In particular, these descriptors can be divided in **three main classes**, depending on the elements of the network to be defined, and which also match with the **intent's types** presented in Section 5.1:

1. **Branch Intent Descriptor:** it declares the **SDN domains or branches**, in terms of SDN switches and networks managed for each branch.

2. **Connection Intent Descriptor:** it states the **logical connections** between SDN branches and/or switches.

3. **Policy Intent Descriptor:** it serves to establish **corporate policies** to be applied in specific parts of the networks, such as the specification of distributed firewalls or QoS rules.

[9] For example, in the case of the interconnection between SDN Branch M and External Domain X, the external domain is connected to the Internet and can propagate a default network to reach the Internet to the edge SDN switch from the SDN branch.

[10] This interaction is presented in Figure 23 with blue, dotted lines that connect each edge SDN switch with its corresponding SDN Controller.

Moreover, there is also another type of descriptor apart from the intent ones: the **General Configuration Descriptor**, related to the declaration of the **regions** (with their corresponding **controllers**) and the **virtual networks** that are present in all the scenario managed by Alviu.

In any case, each type of descriptor implies the creation of the corresponding **entities**, already presented in Figure 22, in the SDN Orchestrator, to have the **vision of the whole network** to be managed. This also results in the **operation** of the **SDN Controllers** to invoke the proper modules and applications that triggers the **deployment of configuration** in the switches (*e.g.* the activation of Quagga modules if required to communicate with external domains) and the **instantiation** of the corresponding **flow rules** in the switches, among others.

The **relationship** between the three classes of **descriptors/intents** and the **configuration** applied over the network infrastructure is depicted in Figure 24, which takes the infrastructure presented in Figure 23 as example. In Figure 24, different colours are used to differentiate each case: **red** colour represents the definition of the **SDN branches** and their **switches**, **green** colour shows the **connections between switches** (excepting the connections to external domains, which is managed by the previous class) and **orange** colour presents possible **corporate policies** that may appear in this kind of scenarios (*e.g.* the usage of DNS filtering to block traffic to a particular web page in a branch, or the data rate limitation in a specific link).

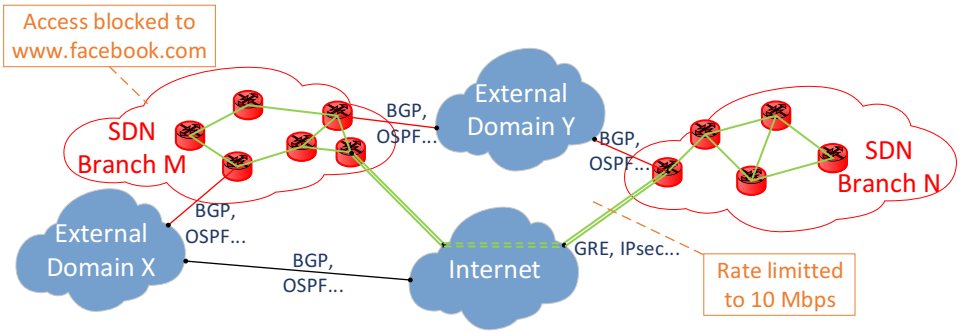

Figure 24: Mapping between the type of descriptors that can be defined and the configuration applied by Alviu in the managed network infrastructure.

As commented in the introduction of this Section, the descriptors must follow a **data model** to specify the **fields** that must be declared in each type and the possible **values** to be included. The **descriptors** are codified in **JSON**, a well-known text format to wrap information exchanged through **RESTful** applications, as in this case.

For better readability of this subsection, the **data model** that composes each class of intent descriptor, followed by the corresponding **fields**, their **meaning** and an **example**, can be found in Annex A.

5.2.2. Alviu Specification

Alviu has been designed as a **modular system**, decomposing the SDN network orchestration and control platform into smaller **building blocks** with a specific **function** in the system. These blocks can be grouped into **two main categories**, which matches with the two main components that define Alviu: **orchestration** modules and **control** modules.

The **orchestration part**, role played by the **SDN Orchestrator**, is in a **higher** level of **abstraction**, as it works with the **network model** presented in Section 5.1, without interacting directly with the network equipment to be managed. Based on that model, the intents and configuration of the SDN domains received through the **descriptors** are translated into specific **operations** that can be internal or directed towards the **control part**, which has **direct connection** to the infrastructure, so that it is able to **apply the configuration** needed on the **network infrastructure** in order to **meet the requirements** declared in the descriptors.

This interaction, which was summarized in Figure 23, is now detailed in Figure 25, presenting the **internal modules** of both the **SDN Orchestrator** and **SDN Controller**.

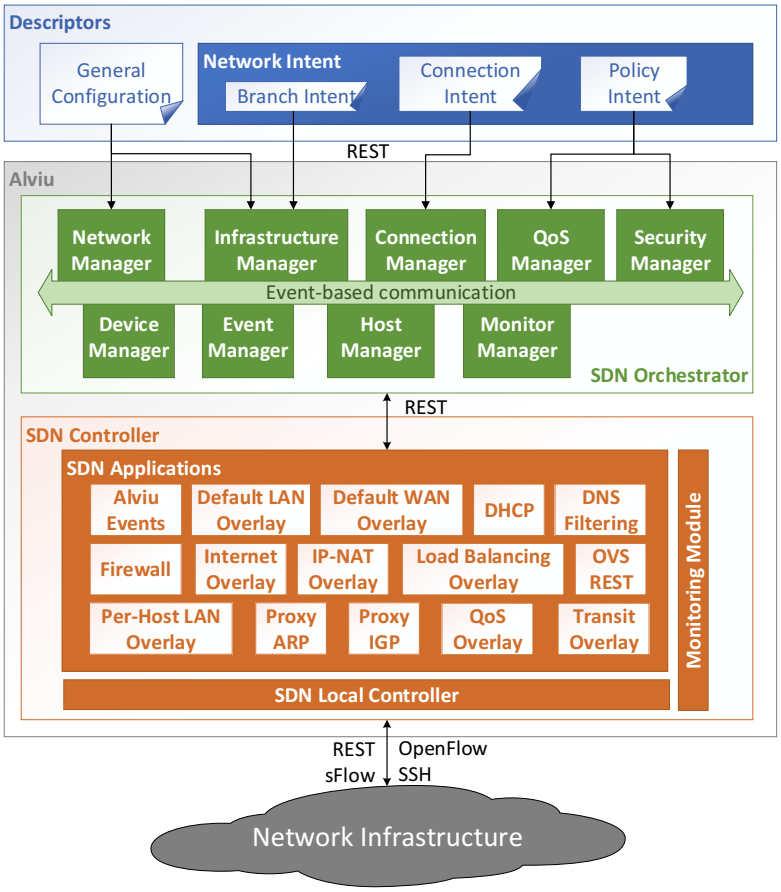

Figure 25: Alviu architecture, including the building blocks that compose the SDN Orchestrator and the SDN Controller.

Starting with the **SDN Orchestrator**, all the **modules** running on it have a similar **design** from an implementation point of view, being oriented to a **microservice** architecture. For this purpose, each module defines **REST endpoints** to be **accessed** by **external components** (*e.g.* for the provision of the descriptors), and, at the other end, it has **REST clients** to connect to the **applications** of the **SDN Controller**.

Between both REST connectors, the orchestrator implements the **main functionalities** carried out by the module, handling the **lifecycle** of the module: **receive data** from the REST endpoints, **perform operations** (including the **event-based communication** with other modules) and **send instructions** to the SDN Controller.

The **modules** that are present in the SDN Orchestrator are the following:

- **Infrastructure Manager:** this module is in charge of **receiving and handling** the data contained in the **Branch Intent Descriptors**, managing all the details related to the **branches** and **SDN switches** of the scenario. Moreover, it also receives, from the **General Configuration Descriptor**, the **configuration** of the **regions** and the **controllers** related to the different branches. For the other elements that belongs to the branch intent (*e.g.* virtual switches or networks), the Infrastructure Manager **interacts with other modules** in order to achieve the configuration proposed in the descriptors.

- **Network Manager:** it manages the **lifecycle** of all the **networking aspects** of the SDN domains, including the **virtual networks** declared (information received from the **General Configuration Descriptor**), the **networks** defined for each branch, the presence of **IGP speakers** and **transit points** if there are interactions with external domains, or even utilities like the configuration of the **DHCP** engine of the SDN Controller and the installation of flow rules to achieve **LAN connectivity** between the switches of the same LAN segment.

- **Device Manager:** in this module, the **management** of the **virtual switches** and **ports** is performed, controlling at all times their status in order to act consequently (for example, Alviu is able to detect if a virtual switch or port is down and reconfigure outdated settings as a result).

- **Connection Manager:** it is in charge of **managing the connection intents** declared in the **Connection Intent Descriptors**. To do this, this module translates the network infrastructure into a **weighted graph**, so that it calculates the **best path** between the two endpoints of each connection intent by applying the **Dijkstra algorithm**, and then it triggers the creation of the corresponding **tunnels** to connect both endpoints. In case of receiving events related to an **incident** on the network that may cause its reconfiguration (*e.g.* physical link down), the Connection Manager is able to **recalculate the installed paths** and change them if needed, guaranteeing the **fulfillment of the connection intents** declared.

- **QoS Manager:** this module is completely dedicated to the **management of the QoS rules** received from the **Policy Intent Descriptors**.
- **Security Manager:** in the same way that the previous module, the Security Manager handles the **firewall and DNS filtering rules** received from the **Policy Intent Descriptors**.
- **Event Manager:** this special module allows to aggregate **events** related to the **infrastructure** (*e.g.* link up or down, switch up or down, etc.), so that the other modules can act consequently in case of happening a specific infrastructure event.
- **Host Manager:** this module pays attention to the **hosts** that can appear in each SDN domain, obtaining information of them from the infrastructure (*e.g.* the MAC addresses from the ARP messages exchanged over the network) or triggering the **networking configuration** of each host through **DHCP** if required.
- **Monitor Manager:** finally, this module manages the **monitoring** of the different **connections established**, contacting with the **Monitoring Module** of the **SDN Controller** for that purpose. This functionality can be used, for example, to establish a **load balancing** service between switches connected by more than one logical connection.

In the case of the **SDN Controller**, it is based on ONOS [36] and it has been built by following the classical **SDN approach** [22], **separating** the **control plane** from the **data plane** with the definition of a well-defined **programming interface** between the **Network Infrastructure** and the **SDN Local Controller** (*i.e.* the southbound API), where the controller platform is able to configure and control the network equipment by using different protocols (mainly OpenFlow).

In addition to this, all the standard functionalities provided by the SDN Local Controller (control of the topology, link status, etc.) are complemented by **specific-purpose SDN Applications**, which interact with the SDN Local Controller through another **well-defined API** (*i.e.* the northbound API) to enhance the control of the network. In this particular case, the SDN Appliactions, together with the **SDN Orchestrator** (which is also in charge of orchestrating the operation of each SDN Application), form the **management plane** of the platform.

Each **SDN Application** takes care of a particular **aspect of the network**, being also related to a set of **modules** of the **SDN Orchestrator** to coordinate their joint action. The **main features** implemented by each SDN Application are the following [110]:

- **Alviu Events:** this application gathers all the **network events** that may happen during the operation of the system (*e.g.* a link or switch down event), **informing** through a REST interface to the **Event Manager** module from the **SDN Orchestrator**, which will act consequently.

- **Default LAN Overlay:** it is in charge of installing the flow rules to achieve **LAN connectivity** between all the switches that belong to the same LAN network, configuration triggered by the **Network Manager** module from the **SDN Orchestrator**. For achieving this, the LAN network is logically transformed into a **tree topology**, where the root node is a WAN switch, the installing **two kinds of flows**: (*i*) a first set of flows related to **upstream traffic**, oriented to reach the root node from any switch in the tree, and (*ii*) another set of flows for the **downstream traffic**, in order to reach any switch in the tree from the root node.

- **Default WAN Overlay:** it allows to achieve **connectivity between WAN switches**, both in the same or different SDN branches, by establishing **logical tunnels** (*e.g.* GRE tunnels) to achieve the level-2 connectivity through ISPs, so that this application must **manipulate the packets** (*i.e.* changing the source and destination IP and MAC addresses) in order to make use of these tunnels . The installation of the corresponding flow rules are triggered by the **Connection Manager** module from the **SDN Orchestrator**.

- **DHCP:** the DHCP application **manages** all the **DHCP requests** received from the **hosts** connected to the SDN network, which are forwarded from the switches to the controller automatically. This way, this application, based on the **DHCP configuration** present in the **Network Manager** module from the **SDN Orchestrator**, **responds** to the requests to provide the **networking configuration** to the hosts.

- **DNS Filtering:** this application, based on the configuration provided by the **Security Manager** module from the **SDN Orchestrator**, is able to **block domains** by examining the **DNS requests** sent by the hosts of the SDN network.

- **Firewall:** in the same way, the Firewall application uses the **firewall rules** provided by the **Security Manager** module from the **SDN Orchestrator** to **filter the traffic** that matches the rules, based on the traffic information between OSI levels 2 and 4.

- **Internet Overlay:** this application is an **extension** of the **Default WAN Overlay** application, also managed by the **Connection Manager** module from the **SDN Orchestrator**. However, in this case, this application install the flow rules needed to **connect the WAN switches** of a given SDN domain to the **Internet**, establishing **tunnels** that forward the traffic to the **edge SDN switch** (*i.e.* the switch connected to the external domain or to the ISP router that provides Internet access), in which another flow rule is used to **forward** the traffic **outwards**. For not colliding with the Default WAN Overlay flow rules, the Internet Overlay flows have **less priority** than the first ones, so that they act as **default rules** in the SDN domain (if the traffic does not match with any Default WAN Overlay flow and there is any Internet Overlay flow, the traffic will match with the Internet Overlay one).

- **IP-NAT Overlay:** it applies **NAT rules** to specific **hosts** of the SDN network that may need this functionality, **translating** the real **IP and port** addresses to others that are **reachable** in the SDN network. This can be used, for example, in case of having legacy hosts that need to be reconfigured, but they are located in a hard-to-reach physical location. The information needed to build the flow rules is provided by the **Network Manager** module from the **SDN Orchestrator**.

- **Load Balancing Overlay:** this application installs the **flow rules** needed to perform **load balancing** when there is **more than one logical connection** between **two switches**. The **Monitor Manager** module from the **SDN Orchestrator** is in charge of **triggering the installation** of these flows, based on the monitoring information gathered from the**Monitoring Module** in the **SDN Controller** using protocols like sFlow.

- **OVS REST:** it is one of the **most used applications** of the catalogue, as it allows to **configure the virtual switches** deployed in the Network Infrastructure. To do this, this application receives **REST requests** from the **SDN Orchestrator's modules** that need to perform an **operation** related to the **virtual switches' lifecycle** (*e.g.* the Device Manager module can trigger the installation or removal of a virtual switch, together with the management of the ports, or the Connection Manager can request the creation or removal of specific ports to handle the traffic to be sent through the tunnel between WAN switches). This way, this application **translates these requests** into specific **commands** that are sent to the switches through a **reverse SSH tunnel**.

- **Per-Host LAN Overlay:** this application complements the operation of the Default LAN Overlay one by installing the flows that **connects** all the **hosts** present in the SDN network with their **corresponding LAN switch**. For this purpose, it uses the **DHCP application** to obtain the **information** needed to **build the flow rules**.

- **Proxy ARP:** it **manages** all the **ARP requests and responses** that are sent through the SDN network, in order to **avoid loops** in the topology that may cause an inefficient behaviour of the network. Moreover, this information is also **used by applications and orchestration modules** for specific purposes (*e.g.* to feed the **DHCP** with the MAC addresses of the hosts present in the network, so that the DHCP can apply the correct configuration for each host).

- **Proxy IGP:** in the same way that the Proxy ARP application, the Proxy IGP is able to **manage the IGP traffic** received from external domains (*e.g.* OSPF or BGP packets), so that it can **extract the useful information** from these packets (for example, the networking data related to the node of the external domain that has established the IGP connection with the SDN domain) to trigger the corresponding **configuration** related to the connection with **external domains**, controlled by the **Network Manager** module in the **SDN Orchestrator**. This information is complemented with

the data extracted from the **Quagga modules** running on the edge SDN switches connected to external domains, which provide the **routes learned from the IGP protocols** to this application through **REST**.

- **QoS Overlay:** this application is in charge of managing the installation of the **QoS policies** in the corresponding switches, being controlled by the **QoS Manager** module from the **SDN Orchestrator**.
- **Transit Overlay:** this application, based on the operation of the **Proxy IGP** application and the control of the **Network Manager** module from the **SDN Orchestrator**, triggers the installation of the flow rules that **connects the edge SDN switches** with the corresponding **node** of the **external domain**, enabling the **interconnection with external domains**.

5.3. Intent States Management

In Alviu, the **intent operation** is based on the **transition** between the possible **states** in which the **intent** could be in a given moment. These **states** are applied, in turn, to all the **entities** that belong to a given **intent** (for example, the Branch Intent relies on the Branch, Switch or Virtual Switch entities, among others), summarizing their current **situation** in terms of the **achievement of the intent**.

5.3.1. States Specification

All the **entities** involved in a given intent share a common set of **states** to define their **status** during the intent operation. These possible states are depicted in Figure 26, also including the possible **transitions between states**:

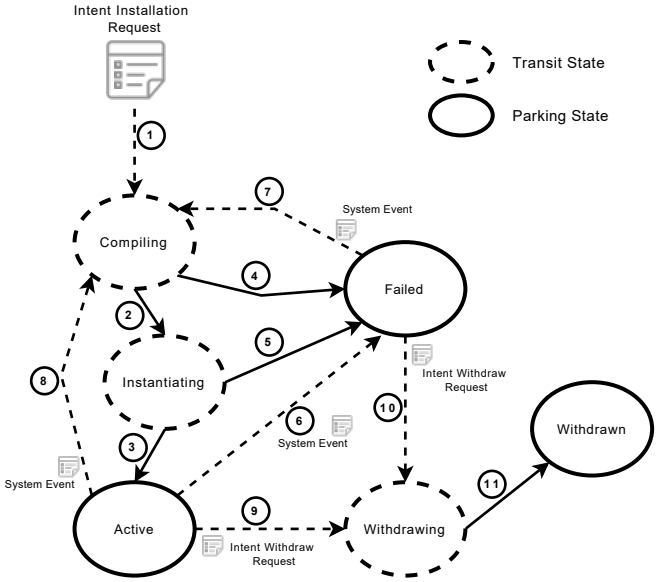

Figure 26: General intent states present during the intent operation.

First of all, there is a distinction between **Parking and Transit States**, depending on whether the **state is definitive or not**. Starting with the **Transit**

States, which are the first ones that appear when an intent is requested, there are three possible options: (*i*) **Compiling**, which refers to the **validation** of the **consistency** of the request, also verifying if the **requirements** exposed by the intent **can be met** based on the input information, (*ii*) **Instantiating**, where the **intent installation** is performed, and (*iii*) **Withdrawing**, which **withdraws the intent**.

Regarding the **Parking States**, there are also three possible states that can fit in this category: (*i*) **Active**, which means that the intent has been **correctly installed** and all the **requirements** have been **satisfied**, (*ii*) **Failed**, which is the opposite to Active: the intent **cannot be installed** in the current network status, and (*iii*) **Withdrawn**, where the intent is **no longer held**.

In the end, the **objective** of all intents is to have all their dependent **entities in Active states**, and in case of **discarding** the intent, to correctly change all the states to **Withdrawn**. Consequently, the **configuration** provided in the definition of the intent is only **completely installed** when the **Active state** is reached, and similarly, it is only **completely deleted** when the intent is in the **Withdrawn state**.

In Figure 26, the **transition between states** is also represented, differentiating between **transitions** that depend on **external events or requests** and **others** that **do not require them**:

- In the **first case**, for example, transition **(1)** is triggered by the **creation of the intent**, transitions **(6), (7) and (8)** are due to **changes in the environment** related to the intent, and transitions **(9) and (10)** are related to the **intent withdrawal**.
- In the **second case**, there are transitions related to **successful operations**, like transitions **(2), (3) and (11)**, and there are others connected with **operations that have failed**, such as transitions **(4) and (5)**.

5.3.2. States Workflow

As stated in the introduction of this Section, the **status of an intent** depends on the **individual states** of each **entity** related to the intent. This way, although the state diagram depicted in Figure 26 is followed by all the intent types, there are **particularities in each intent type**, as the related entities have their own complexity, introducing **dependencies between states of different entities**.

To analyze the impact of these constraints, the **Branch Intent** and **Connection Intent** workflows for both **intent installation and withdrawal operations** will be reviewed and explained in this subsection, checking how the general intent states framework can be applied to each type.

In Figure 27, the **Branch Intent installation operation** is presented, with all the possible **states** that the different **entities** related to the Branch Intent (already presented in Figure 22) can have:

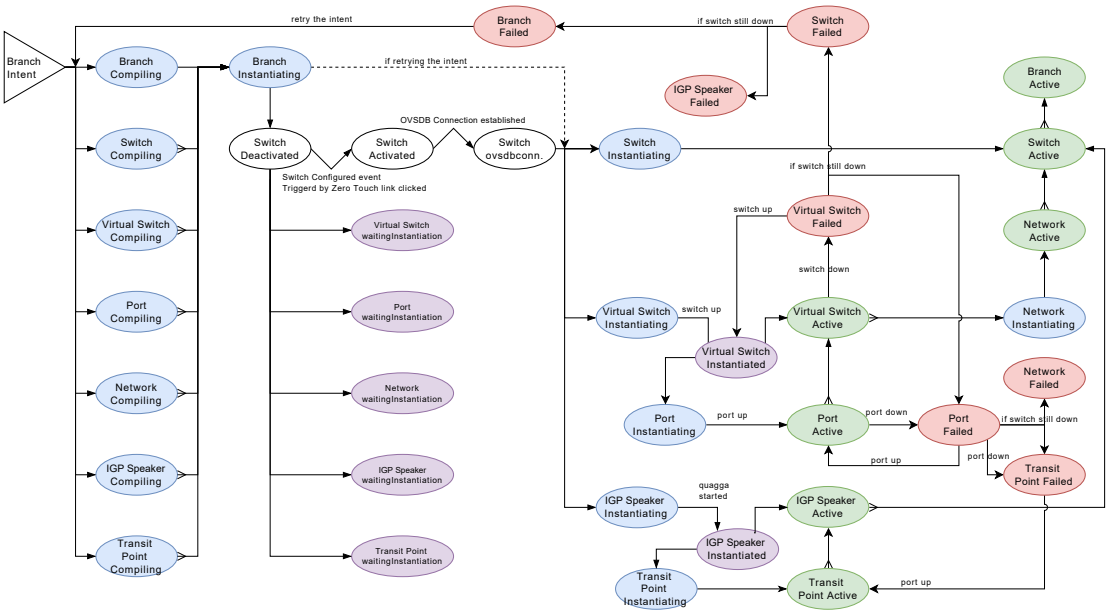

Figure 27: Branch Intent installation operation workflow.

When the **Branch Intent is declared**, all the **entities** enter in the **Configuring state**, and if all requirements are fulfilled, the **Branch** entity changes its state to **Instantiating**. From this moment, the **rest of entities** stay in an **intermediate state**, called **WaitingInstantiation**, because they are not instantiated at the same time. In the case of the **Switch** entity, it passes through different states related to the **activation** of the physical switch, using Zero Touch procedures, before passing to the **Instantiating state**.

From this point, the **instantiation** of the **Virtual Switches** and **IGP Speakers** are triggered. In the first case, the **Virtual Switch** is considered **Active** when it is correctly **installed** and all the **Ports** that depend on it are also **installed**. To achieve this, the corresponding **switch up and port up events** have to be captured by Alviu to perform the **transition between states**. A similar process happens with the IGP Speakers and their Transit Points: the **IGP Speaker** is only **Active** when all the **Transit Points** are in **Active** state and when the **Quagga module is running**.

The activation of a Virtual Switch triggers the **instantiation and activation** of the **Networks** attached to the Ports of that Virtual Switch, and finally, when all **Networks and IGP Speakers** are in **Active** state, the corresponding **Switch** is marked as **Active**, and when **all the switches** of the branch are in **Active** status, then the **Branch** is marked as **Active**, meaning that the **requirements** declared in the Branch Intent **have been met**.

The diagram also presents some **casuistries** in which some entities may pass to a **Failed** status. For example, if a **switch down event** is detected, the corresponding **Virtual Switch** will be marked as **Failed**, and will **wait** some time for receiving a **switch up event**. If it is **not received**, a ***cascade update*** of all entities' state is applied, changing all to the **Failed state** and **restarting the intent workflow** to do all the **changes** needed **meet the requirements again**. In addition to that case, the **port down event** is also represented, which causes the **Port Failed** state and, consequently, the **Transit Point Failed** state

for the transit points attached to that port, only returning to the **Active** state again when the **port up event** is received.

The same process is followed in the **Branch Intent withdrawal operation**, depicted in Figure 28. When the request is received, the different entities **change their state in order**, reaching the **Withdrawn state** if all the **operations finished correctly**. No Failed states have been represented in this diagram for better readability.

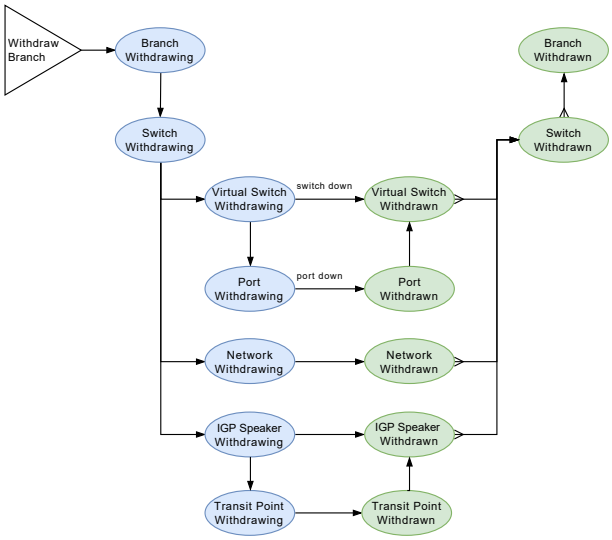

Figure 28: Branch Intent withdrawal operation workflow.

In the case of the **Connection Intent**, as it has **less entities** that depend on it, the **workflow** is also **simpler**. This is presented in Figure 29, where there are two main entities that are used to establish a connection between two switches, as depicted in Figure 22: the **Tunnel** entity (*i.e.* the **connection** itself), and a pair of **Tunnel Port** entities, each of them associated to the **switches** that are connected through the tunnel.

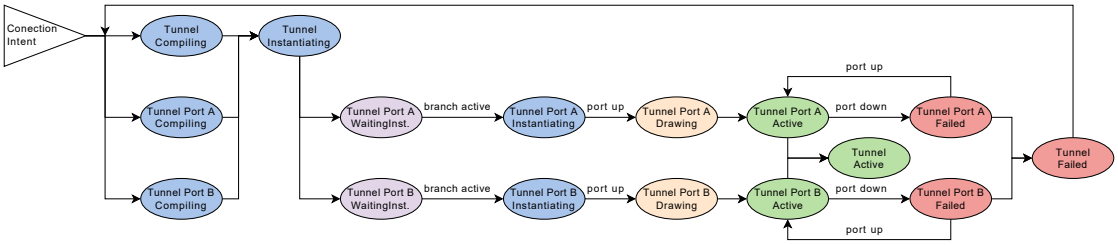

Figure 29: Connection Intent installation operation workflow.

After passing the **Compiling** state, the **Tunnel** passes to the **Instantiating state**, and both **Tunnel Ports** stay in the intermediate **WaitingInstantiation state** until each **Branch** that has each Switch becomes **Active**. Then, the **Tunnel Ports** start to be **instantiated**, and after being deployed, the **port up event** must appear. In that moment, the Tunnel Ports passes to the **Drawing state**, a special state which means that the **weighted graph** is being **updated** with the

new connection, **calculating the best path** to achieve the connectivity between the two endpoints. After achieving this, the **Tunnel Ports** are marked as **Active**, and consequently the **Tunnel is activated**.

Of course, there may happen **problems** during the intent operation; for example, in case of receiving a **port down event**, the corresponding **Tunnel Port** would be marked as **Failed**, then **waiting** some time for receiving a **port up event**. In case of **not receiving it**, in the same way that happened with the Branch Intent, the upper entity (*i.e.* the **Tunnel**) will be also marked as **Failed** and the intent would be **restarted** in order to meet again the requirements.

Finally, the **Connection Intent withdrawal operation** also follows the same procedure than the Branch Intent withdrawal operation, with a **transition** between the **Withdrawing to Withdrawn** event when the **port down event** related to the Tunnel Ports is received. This workflow can be seen in Figure 30:

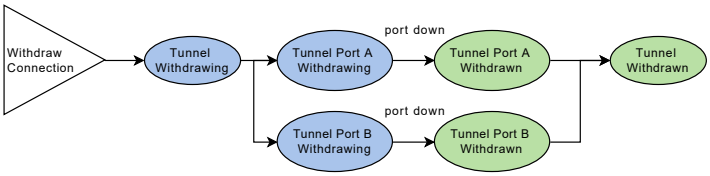

Figure 30: Connection Intent withdrawal operation workflow.

5.4. Performance Evaluation

Apart from building a complete and functional service to orchestrate and control SDN domains, the solution must also fulfil some specific **performance requirements**, ensuring the **convergence** of the **intent deployment** in a **reasonably predictable time**, avoiding deployment times with an exponential evolution as the number of SDN branches increases.

In this Section, a full **deployment** of a network controlled by Alviu will be **evaluated** in terms of the **deployment time**, which is the time elapsed from the **launch of the intent** until the **infrastructure is properly deployed** (*i.e.* all the branches are in active state) and there is **connectivity** between **all the switches** on the network in a given **topology**.

5.4.1. Testbed Setup

The **testbed** used for the evaluation of this deployment consists of an **Ubuntu Server 16.04 LTS virtual machine** [90], with 12 vCPU and 12 GB of RAM, deployed in a server virtualized with *Proxmox* [89], which is equipped with 40 Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20 GHz and 128 GB RAM. In the virtual machine, **Alviu** has been deployed in a ***Dockerized* format**, deploying both the SDN Orchestrator and the SDN Controller with *Docker* containers [92].

To deploy the different **scenarios** to test, a **Continuous Integration** platform has been used. This environment consist of a set of **microservices** that allows to check the correct **definition** of the **Network Intent Descriptors**, deploy **network topologies** using **lightweight virtualization technologies** (*e.g.* containers), or perform **network unit tests** (*e.g.* connectivity between nodes) over that topology, among other functions.

The architecture of this CI environment is presented in Figure 31, where **three microservices** can be distinguished: (*i*) ***dockerTopo***, in charge of **deploying** the **network topology** with **containers** according to the data from the **Topology Descriptors**, which contain the **network elements** (*e.g.* hosts, SDN switches, routers from external domains, etc.) that compose the desired topology, (*ii*) ***deploymentService***, used to provision of the **Scenario Descriptors**, which contain the **definition** of the **topology** and the **intent descriptors**, and (*iii*) ***proxyClient***, which connects the ***deploymentService*** with both the **SDN Orchestrator**, providing the **Network Intent Descriptors**, and the ***dockerTopo*** service, to supply the **Topology Descriptors** to it, so that it performs the **translation** of the **Scenario Descriptors** to the corresponding **Topology and Network Intent Descriptors**.

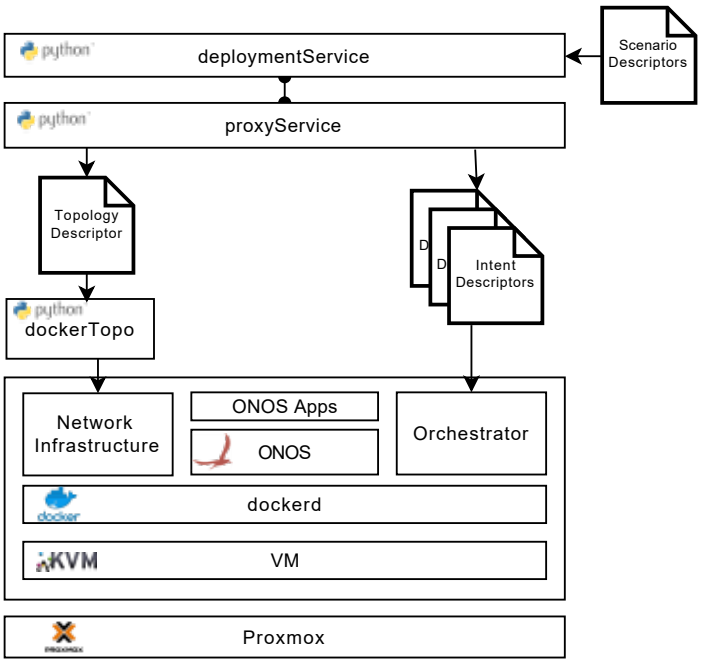

Figure 31: CI environment architecture, including the technologies used in each component.

The **topology type** chosen for the performance evaluation process is the **star topology**, as it is the most common option followed by enterprise and academic networks to connect their domains, having a **central SDN branch** acting as headquarters and the **rest of branches connected** to the **central** one.

In particular, to simplify the scenario evaluated, each **SDN branch** deployed in the CI environment will be **composed** by a **single SDN switch** with OVS and Quagga installed, with a **host** connected to it through a **LAN port**, and with a **connection** to a **router**, based on Quagga, from an **external domain** through a **TRUNK port**, using **OSPF** to exchange routes with it. This router will also have a **host** connected to it, in order to check the **connectivity between hosts from different domains.** Finally, **one branch** will act as **headquarters**, and the **rest of branches** will be **connected to it** through a **tunnel connection** established **between SDN switches**.

This particular deployment is depicted in Figure 32, presenting the **HQ Branch** as the **root branch** of the **star topology**, and **several branches**, from 1 to N, **attached** to it through a **GRE tunnel** between SDN switches, which are also secured with IPsec.

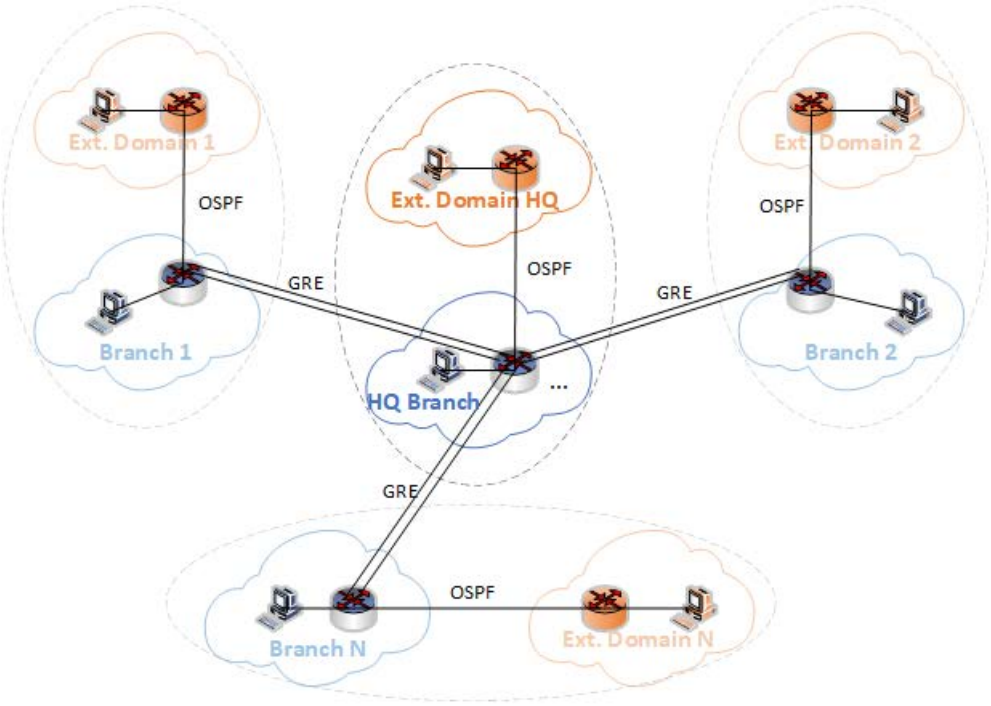

Figure 32: Star topology evaluated in the performance evaluation analysis.

5.4.2. Deployment Time Evaluation

The **evaluation** of the **deployment time** has been done by measuring the **time** spent by Alviu to **meet the requirements** defined in the **intents**, which are mainly two: (*i*) **deploy** an **active branch** with **one SDN switch** connected to a **host** and to an **external domain** (*i.e.* Branch Intent), and (*ii*) **connect** the **SDN switch** to the **endpoint** located in the **HQ Branch** through a tunnel (*i.e.* Connection Intent). However, the **convergence time** to **learn** the **prefixes** from **external domains** has been **excluded** from the analysis, as it is **difficult to predict beforehand**, directly impacting in the accuracy measurements during the evaluation.

The **evolution** of this performance metric, **changing** the **number of branches** deployed in the topology from one (only the HQ branch) to ten, can be seen in Figure 33. Several conclusions can be extracted from this graph, which shows the **linear trend** of the **deployment time**, confirming that the time spent to deploy a given topology is quite **predictable and scalable**, which is a **desired behavior** against exponential trends that can saturate the system. This has been possible due to the **concurrent management** of **multiple intents** provided by Alviu, which is able to process intent requests in parallel to improve system's scalability.

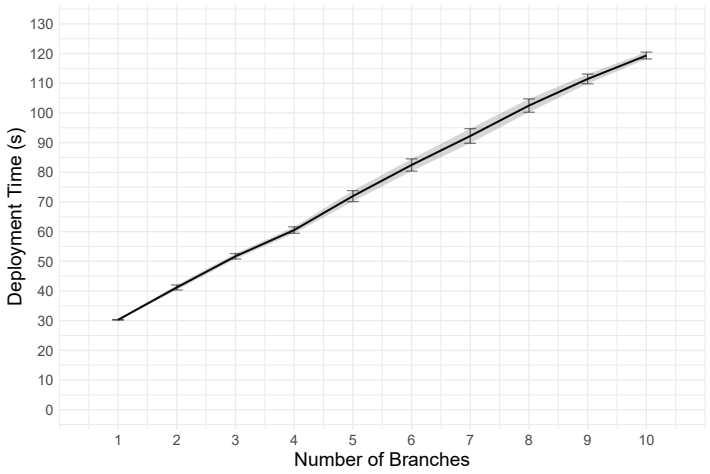

Figure 33: Evolution of the deployment time, in seconds, varying the number of branches deployed in the star topology.

Furthermore, the **values** obtained are also **coherent** for a typical system operation, having a minimum deployment time of around **30 seconds** for **one branch**, which **increases** at a **rate** of **10 seconds** per **new deployed branch**, approximately. This confirms Alviu's ability to **manage multiple branches simultaneously**. This time can even be **lower** because Alviu uses some **small guard times** to **stabilize the system** that may be avoided (*e.g.* it waits a few seconds after activating a branch and before starting the connection of the branch with other branches), but they have been **maintained** as they **ensure the system scalability without** having a great **impact on performance**.

5.5. Summary

In summary, this Chapter has presented **Alviu**, a **SD-WAN Orchestrator** that can be applied to **enterprise and academic networks**, as well as to **5G transport networks**.

The main contributions that have been reviewed in this Chapter are **Alviu's abstraction capabilities** used for **modeling** the **network infrastructure**, as presented in Section 5.1, which is managed and orchestrated with a system based on a **modular architecture**, as discussed in Section 5.2, implementing **intent-based networking** capabilities.

The **intents** have also had great relevance in this Chapter, including their **high-level definition** with the usage of the **Network Intent Descriptors**, and also the full specification of the **different states** that a particular **intent** and their **related entities** may **have** during the operation of the system, as stated in Section 5.3.

Finally, in Section 5.4, the **deployment time** spent by Alviu to achieve **intent's fulfillment** has been **evaluated** in a **CI environment**, which allows to **easily deploy and test** a **star topology** with a **variable number of branches**. In this way, the **system scalability** has been confirmed, as the measured **deployment time depends linearly** on the **number of branches** present in the scenario, not following an exponential trend.

6

Evaluation and Demonstration of Intent-Based Orchestration Capabilities in Real Scenarios

In Chapter 5, **Alviu** was presented and evaluated as an alternative to **control and orchestrate 5G** networks with the application of **Software-Defined Networking** and the **intent-based networking approach**. Its **modularity**, together with the capability of **enhancing** the platform with the inclusion of new features thanks to the **network programmability** of the platform, following a simple deployment model, allows Alviu to achieve a **customized management** of the SDN domains.

One of the most ambitious objectives of Alviu is to **integrate different network domains**, which may be **related to SDN or not**. In this context, the **management** of the **information** contained in the traditional **IGP protocols** is **crucial** to accomplish this goal, extracting the **network prefixes learned** from external domains to **install the proper flow rules** in the SDN network to ensure the connectivity between domains.

Other value-added capability already integrated in Alviu is the ability to perform **load balancing** between several **logical connections** between SDN switches, thus guaranteeing a fault-tolerant service while using adequately the network resources. These two **examples**, together with other ones like the provision of **distributed policies** related to firewall or QoS services, justify Alviu's ability to be **adapted** to **different scenarios and casuistries**.

This Chapter focuses on the performance evaluation of Alviu, in the **two** first **use cases** aforementioned: the **interconnection with external domains** and the **load balancing** between switches connected with more than one link, detailing the **solution** implemented in Alviu to integrate them in the system workflow, and also **testing** them in a **real scenario** to confirm their correct implementation.

To to this, the following **structure** to describe both cases is proposed:

- In Section 6.1, the first use case, related to the **interconnection** between **SDN domains** and **external domains** is presented, detailing the way **Alviu** handles the **information** received from **IGP protocols** to achieve this connection.
- Secondly, Section 6.2 goes in depth in the **load balancing** use case, explaining the way in which **multiple links** are introduced in **Alviu's network graph** and how and when the load balancing between different links that connect the same SDN switches is **triggered**.
- After explaining the two main use cases studied in this Chapter, Section 6.3 describes the **Proof of Concept** performed to check the correct **behaviour** of both use cases in the same **testbed**, and also introducing **other value-added capabilities** like the provision of **firewall and QoS rules** related to **network slicing capabilities**.
- To conclude, Section 6.4 **summarizes** the lessons learned in this Chapter and **concludes** this chapter.

6.1. Interconnection with External IGP Domains

6.1.1. Use Case Overview

As it has already been commented on several occasions, the problem of the interconnection between SDN branches and external domains using legacy IGP protocols is a matter of being able to **understand** the **messages** related to **IGP protocols** that are sent from the **external domains**, **manipulate** them in order to **extract** the useful **information** and **use them** to **answer** the external domain **back**. This process is summarized in Figure 34:

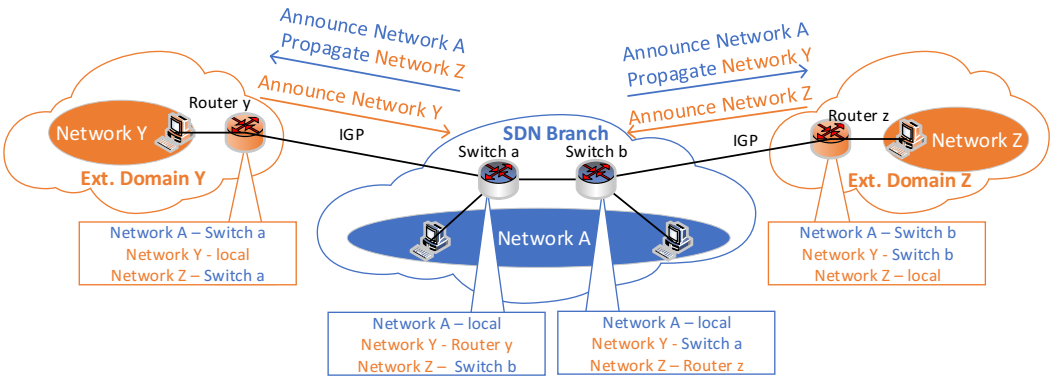

Figure 34: Example of topology with interconnection with external domains.

In the previous Figure, it can be seen that the ***external domains*** *Y and Z* **announce** their **networks** (*networks Y* and *Z*, respectively) to the **edge SDN switch** with which they have an IGP session established (*switches a* and

b, respectively). This exchange of information has been possible because, during the connection establishment phase, the edge SDN switches have been able to **understand** the data encapsulated in the IGP messages, **manipulate** them to obtain that data and **use** the IGP protocol messages to answer back with the network information related to the SDN branch.

In this way, when a **network prefix** is **received** from an **external domain**, it is immediately **forwarded to Alviu**, which decides the proper **flow rules** to be installed in each SDN switch to correctly **reach all the networks** from external domains, according to the best paths calculated in the network graph, and also **instructs** the **SDN domains** to **reply** to the **external domains** in order to provide to them the **networks** related to the **SDN branch** and to other **external domains** to allow the **connectivity** between all the networks present in the topology. As a result, each **SDN switch learns** how to **reach all the networks** of the scenario, and the same for each **router** from the **external domains**.

In the previous example, the *switch a*, for instance, learns from the *router y* that the *network Y* can be accessed from it. Regarding *network Z*, as the *switch b* has already learned that it is reachable from *router z* and this information is also known by Alviu, the orchestrator instructs switch a to learn that the *network Z* is reachable from *switch b* (according to the network graph), propagating this information to the *router y*, apart from announcing *network A*, so that *router y* will learn that *networks A* and *Z* are reachable from *switch a*, despite there are networks that are not handled by the SDN branch (*e.g. network Z* is not managed by *switch a*, but it knows how to forward the traffic to reach it).

In the topology presented in Figure 34, the **SDN branch** is not only able to **connect** with other **external domains**, but it also **connects external domains between them**, acting as a **transit network**.

To achieve this, the integration of **Quagga** in the **edge SDN switches** is **compulsory**, in order to **handle** the typical **IGP protocols** that are used in external domains (*e.g.* BGP or OSPF). However, this integration is **not trivial**, as the SDN switches are based on **OVS**, needing a **particular configuration** to **allow the communication** of Quagga through the switch ports that connect it to the edge routers from external domains. This issue will be discussed in depth in Section 6.1.2.

Moreover, apart from Quagga, some **lightweight processes** are also needed in order to provide to **Alviu** all the **information** learned from **external domains**, having then all the **knowledge** to make the proper **decisions** in terms of interconnection with external domains (*e.g.* propagate prefixes to other external domains, install the flow rules to achieve the connectivity, etc.). The specification of these processes, together with the workflow followed by Alviu when receiving IGP traffic, will be presented in Section 6.1.3.

6.1.2. Integration of Quagga in OVS-based Switches

As a reminder, **Alviu** deploys and configures **OVS** in the managed SDN switches, interacting with it by using **OpenFlow** in order to **fully instruct** the

virtual switch with the proper **flow rules** to handle the traffic in such a way that the **intent's requirements** are **met**.

To do this, the **physical ports** of the **switch** are **connected** to the corresponding **OVS** deployed on it, delegating to OVS the control and management of the ports. This is true for the LAN (to be used in LAN networks) and TRUNK (to connect the switch with external domains) ports, but **WAN ports** (which serve to connect SDN branches through logical tunnels) need a **specific deployment** to achieve their purpose, as an **IP address** must be specified for each endpoint to **create the tunnel** and the **ports directly managed by OVS cannot have IP addresses** attached.

The solution to achieve this goal is depicted in Figure 35, showing an example of a typical SDN switch with four ports, using the first one as WAN port, and a separate port for management purposes (*e.g.* to establish the OpenFlow session between switch and SDN Controller). In this case, this switch has an intermediate **Linux Bridge** to connect the WAN port (ge-1-1-1) and the OVS. As the **Linux Bridges** can have **IP addresses** assigned to them, this would **solve the issue** for WAN interconnection, using this bridge to create and terminate logical tunnels. However, a bridge must connect interfaces between them, so the Linux Bridge cannot be directly attached to the OVS. For this reason, a **pair of Virtual Ethernet** (veth) interfaces (*i.e.* ge-w-1-1 and ge-w-2-1) are created, connecting in this way the Linux bridge with the OVS.

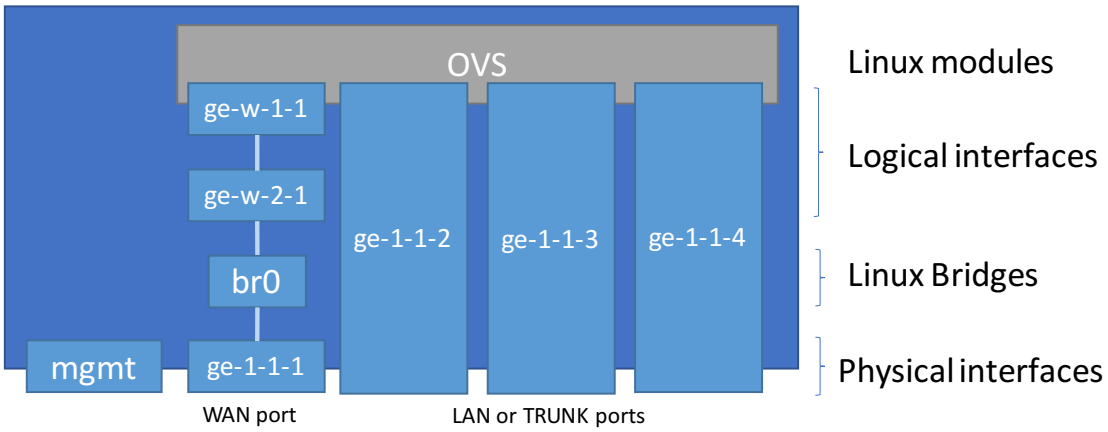

Figure 35: Typical internal architecture of a WAN-type SDN switch.

The **edge SDN switches** are **particular cases** of **WAN-type SDN switches**, as they are utilized to **interconnect domains**, so they must have at least **one WAN port** defined on it, with the corresponding Linux Bridge and a pair of veth interfaces to connect the Linux Bridge to the OVS, as already explained. In this architecture, **Quagga must fit** in to establish the logical connection with external domains.

This integration can be easily achieved with the architecture proposed in Figure 35 by **handling the messages** exchanged by **Quagga** from the **Linux Bridge** of the **WAN port**, which is the only interface interacting with the OVS that can have an IP address attached. Then, the **IP address** used by the switch for

the **interconnection** with the **edge router** of an **external domain** would be defined in the **Linux Bridge**.

As a result, the Linux Bridge would have **multiple IP addresses attached**: **one** for establishing **tunnels with other SDN domains**, and **one** for **each external domain**. This feature is possible in the Linux kernel, enabling the capability of handling IGP sessions from multiple external domains by just **connecting** the **switch** to these **domains**, using **TRUNK ports**[11], and then **configuring** the **Linux Bridge** and **Quagga** with the proper **IP address** and **IGP information** to enable the IGP session with other external domains.

The only issue remaining to be solved would be how to **forward** the **IGP traffic** in the **Linux Bridge** to reach the proper **TRUNK port**, as it acts as **level-2 switch**: if a packet destination is unknown, it will send all the traffic through all the ports excepting the one from which it has received the traffic. With this behaviour, the **Linux Bridge** will also **send all the IGP traffic** managed by Quagga through the **WAN physical port**, which **must not happen**.

The **solution** for this problem is presented in Figure 36, which completes Figure 35 with the introduction of **Quagga**, exchanging the IGP traffic through the Linux Bridge defined for the WAN port (br0), and also specifying that the ge-1-1-2 port acts as **TRUNK port**, so that it would be used for the connection with external domains. To **avoid flooding the WAN port** with **IGP traffic**, a set of ***ebtables* rules**[12] would be created in order to **filter the traffic**. This way, the Linux Bridge will only **send/receive IGP traffic to/from the OVS**, **instructed by Alviu** to correctly **forward the traffic from/to the corresponding TRUNK port**. The details of this behaviour will be better explained in Section 6.1.3.

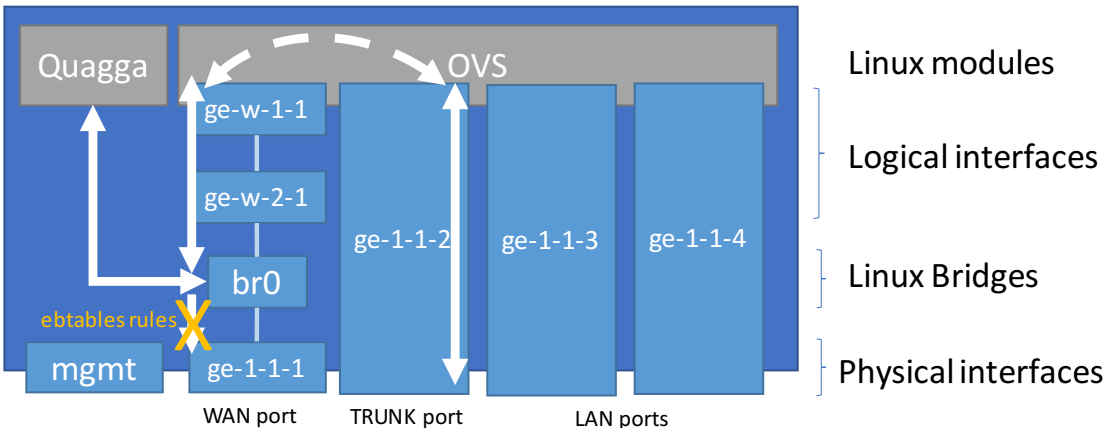

Figure 36: Connection of Quagga to OVS to allow the exchange of information with external domains.

[11]It is possible that **multiple external domains** could be **attached** to the **same TRUNK port** (*e.g.* if the TRUNK port is connected to a switch or router to which the external domains are connected), so it would not be completely true that a TRUNK port was needed for each external domain, fact that would help to save the use of physical ports as TRUNK ports if necessary.

[12]They are similar to *iptables* rules, but they are specifically designed to act in Linux bridges, which is the case here.

6.1.3. System Workflow

The **workflow** related to the interconnection with external domains consists of different **stages**, but mostly involving **the same components** of the **architecture** in all of them. That way, the **entities** which participate in this process are the **edge router** from each **external domain**, the **edge SDN switches** from the **SDN domains** connected to the different external domains, and **Alviu**, with the usage of the **Proxy IGP application** in the **SDN Controller** and the **Network and Connection Manager modules** from the **SDN Orchestrator**.

The **first phase** in the workflow is known as the **IGP connection establishment phase**, which is detailed in Figure 37 with the **high-level messages** exchanged between the different entities.

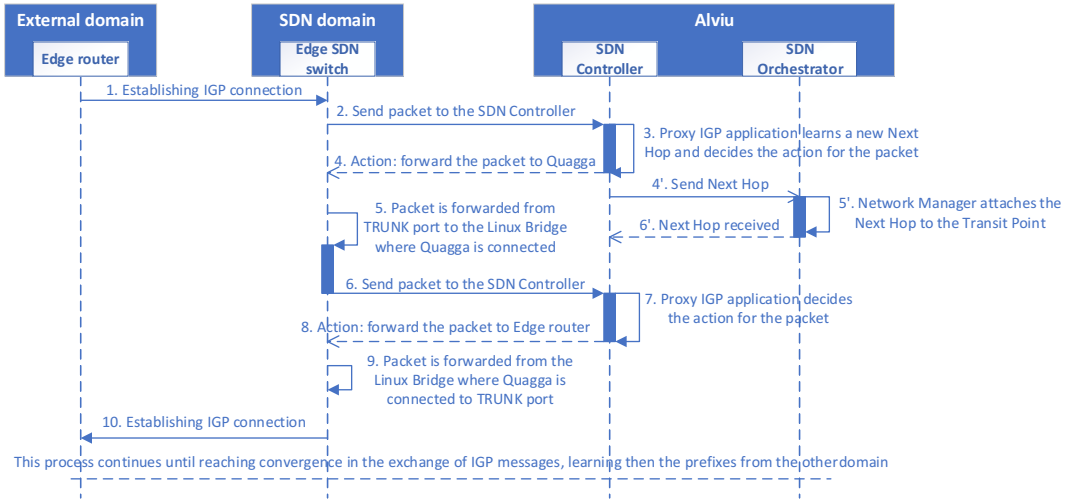

Figure 37: IGP connection establishment workflow.

The full description of the messages is the following:

1. A first message is received in a **TRUNK port** of the edge SDN switch from the corresponding **edge router**, indicating the intention to **establish a IGP connection** from the edge router. To make this message possible, the **edge router** have to be **configured beforehand** with the proper **network information** of the **edge SDN switch** (*i.e.* the IP address attached to the Linux Bridge related to this connection) to be able to establish a **IGP neighbor relationship** with the edge SDN switch.

2. The packet reaches the **OVS**, which has been instructed beforehand by the SDN Controller to have a **reactive behaviour** with the **IGP traffic**, **sending** it always to the **SDN Controller**.

3. Then, the **SDN Controller** will be the responsible for **deciding** what **action** must be done to the **packet**. In addition to this, as a **new IGP connection request** has been received from an **edge router** in an external

domain, this **endpoint** will be considered as a new **Next Hop**, being **saved internally** in the Proxy IGP application with all its related **network information**: IP address, device and TRUNK port where it is connected, MAC address (obtained by the Proxy ARP application afterwards), etc. This last action triggers a **parallel workflow** with the **SDN Orchestrator**, with the following messages:

4'. All the data related to this new **Next Hop** is **sent upwards** to the **Network Manager module** of the **SDN Orchestrator**.

5'. The orchestrator **attaches** the **Next Hop** learned to the **Transit Point** that corresponds to this IGP interconnection. As the device and port in which the Next Hop is connected are provided in the data received from the SDN Controller, the Transit Point can be easily inferred from that information. The impact of the inclusion of this new entity in the data model will be reviewed after explaining all the messages of this stage.

6'. Finally, the orchestrator sends a **reply** to the SDN Controller to **confirm the operation**.

4. As stated in message (3), the SDN Controller replies back to the OVS with the **action** to do with the **packet** received. As it must be received by Quagga, the action is to **send the packet** to the **Linux Bridge** to which **Quagga** is **attached**.

5. This internal interaction represents the process of **forwarding** the packet from the **TRUNK port** to the **Linux Bridge**, being then **received by Quagga**.

6. **Quagga** generates the **reply** to the message and **sends** it to the **Linux Bridge**. In this way, the bridge **forwards** the traffic to the **OVS**, and then, the same interaction presented in message (2) is performed: as the OVS is configured in reactive mode for IGP traffic, it **sends** the packet to the **SDN Controller** in order to know what to do with it.

7. In this case, the message comes from Quagga and has to be sent to a Next Hop which is already known, so the **Proxy IGP** application will only generate the **reply to the OVS** with the **action** to do.

8. The action, in this case, is the opposite to the one instructed in message (4): now, the **message** must be **forwarded** to the **TRUNK port**, in order to reach the Next Hop.

9. In the same way that in the interaction number (5), the **packet** is **forwarded** from the **Linux Bridge** to the **TRUNK port**.

10. Finally, the packet is **received** in the **edge router**, and from this point, this process **continues** until the **prefixes** from the external domains are **announced** due to a **convergence** in the IGP negotiation.

To see the relationship between the entities involved in this workflow, Figure 38 presents an example of the **mapping** between the **entities** and the **real components** of the network infrastructure. Firstly, the **IGP Speaker** corresponds to **Quagga**, as already known. Each **Transit Point** is related to a specific **TRUNK Port** of the switch, but it is **possible** to have **several transit points** in a **port**, as it happens with the ge-1-1-N port in the example shown. Then, **each Transit Point** can have **at least one Next Hop behind**, as there can be an **intermediate device** (*e.g.* switch or router) connecting the TRUNK port with the different edge routers, as happens with the interconnection between the TP TRansit Point1, in the ge-1-1-2 port, and the edge routers A and B. Finally, a **Next Hop** may have also attached to it **several Prefixes**, as stated in the external domain Y.

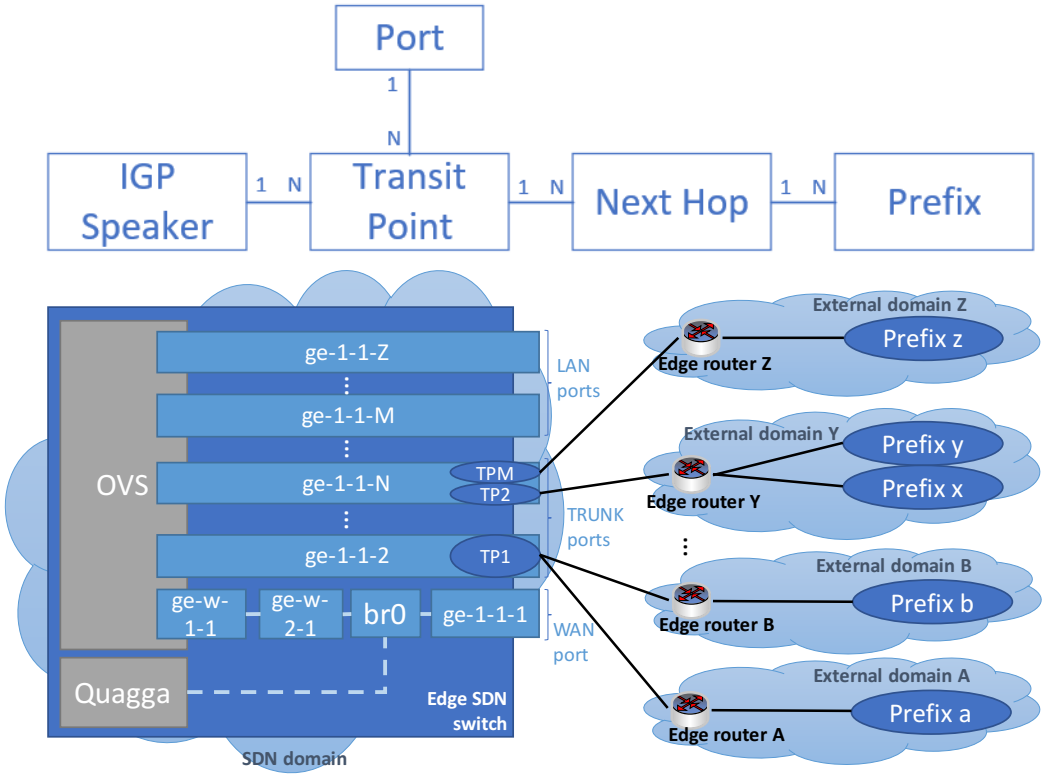

Figure 38: Introduction of Next Hop and Prefix entities in the data model, showing how all the entities related to IGP match in a real scenario.

With this clarification, and continuing with the stages of the workflow, the next phase corresponds to the process of **learning** a specific **prefix** from an **external domain**, which can be seen in Figure 39:

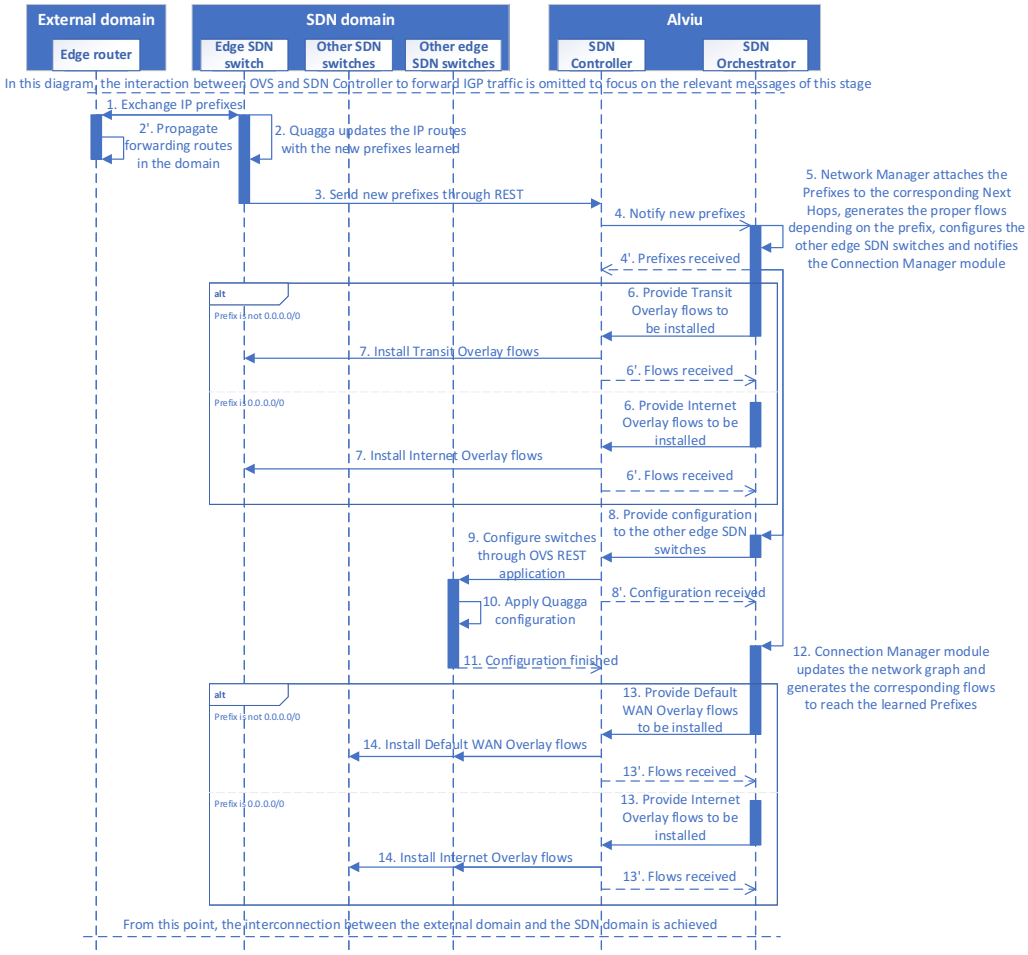

Figure 39: Prefix learning workflow.

The full description of the messages is the following:

1. When there is **convergence** between **IGP neighbors**, the **IP prefixes** managed by each endpoint **are exchanged**. Of course, in the case of the prefixes received by the edge SDN switch, the IGP packets will follow the same workflow than presented in the Figure 37, but it is not represented here to simplify the diagram.

2. This way, the **prefixes** are received by **Quagga**, then **updating the IP routes** with the new prefixes learned. In parallel, the **edge router** from the external domain also **propagates** the **forwarding routes** to the other devices from its domain, according to the prefixes learned from the SDN domain, which is represented with the message (2′).

3. Due to this, a **script** running in the **edge SDN switch**, **monitoring** the **Quagga routing table**, detects that **new prefixes** have been **learned**, so that it triggers a **notification** to the **SDN Controller**, which has a REST endpoint enabled in the Proxy IGP application to handle this kind of notifications.

4. The Proxy IGP application in the SDN Controller **forwards** the **prefixes** to the **SDN Orchestrator**, answering back with the message (4′).

5. After receiving the prefixes, the orchestrator, in the **Network Manager module**, performs several operations: (*i*) attach the **prefixes** learned to the corresponding **Next Hop** entities, (*ii*) generate the **flows** to be installed in the **edge SDN switch** (message (6)), (*iii*) **configure** the **other edge SDN switches** to **propagate the prefixes** in **other external domains** (message (8)) and (*iv*) send a notification to the **Connection Manager module** to **update the graph** and **install** the corresponding **flows** (message (12)).

6. Depending on the network prefix learned, a specific flow is installed in the edge SDN switch. If the **prefix** is **different** than the **default one** (*i.e.* 0.0.0.0/0), then the **Transit Overlay flow** is requested. **In other case**, it would be the **Internet Overlay flow**, as the default prefix represents an **Internet connection**. This message is **replied back** from the **SDN Controller** notifying that the flows have been received correctly, as shown in message (6′).

7. This message is related to the installation of the corresponding flows (**Transit Overlay** or **Internet Overlay**) in the edge SDN switch. This flow specifies that **all the traffic** whose **destination IP address** is the **prefix learned** must be sent through the corresponding **TRUNK port**. In the case of the **Internet Overlay flow**, as the default prefix matches with all IP addresses, its **priority** would be **lower**, so that a traffic flow would only match that rule if no other flow with more priority is matched.

8. Here, it is reflected the **notification** from the **Network Manager module** to the **OVS REST application** in the SDN Controller to **configure** the **rest of edge SDN switches** with the prefixes learned, so that they can **propagate** them through their corresponding **external domains**. This request is **replied back** by the SDN Controller in the message (8′).

9. As a result, a **reverse SSH tunnel** is established from the **OVS REST application** to the **switches** in order to apply the **configuration**.

10. In this point, the **configuration** is **applied** in the **Quagga** modules, and the prefixes learned start to be **propagated** in the other external domains.

11. Finally, the switches send a **notification** to OVS REST to confirm that the operation has been done correctly, **closing** then the **SSH tunnel**.

12. Finally, the **Connection Manager module** of the orchestrator receives the **prefixes** and calculates the **best path** to reach them **from the rest of SDN switches**, **updating** the **network graph** consequently and triggering the **installation** of the **corresponding flows** in these switches, depending on the prefix again (whether it is the default prefix or not).

13. Again, depending of the prefix, the orchestrator asks the SDN Controller to install a specific flow: if it is **not the default one**, the **Default WAN Overlay** flow will be selected, managing the connectivity to the prefixes as it were a WAN connectivity, so a **tunnel** will encapsulate the traffic. **In other case**, the **Internet Overlay** flow will be selected again. This request is **answered back** with message (13′).

14. Finally, in the same way that in message (7), the corresponding **flows** are **installed** on **each SDN switch different** than the **edge SDN switch** from which the **prefixes** have been **received**. However, in this case, the **output port** will be the **next hop** to reach the prefix learned, as calculated by the network graph.

In the same way, in a specific point of time, a given **prefix** may be **no longer propagated** by the edge router of an external domain, thus triggering the **prefix deletion workflow** presented in Figure 40. This workflow will not be fully explained because it follows exactly the **same process** than explained in Figure 39, but doing the deletion instead of the installation.

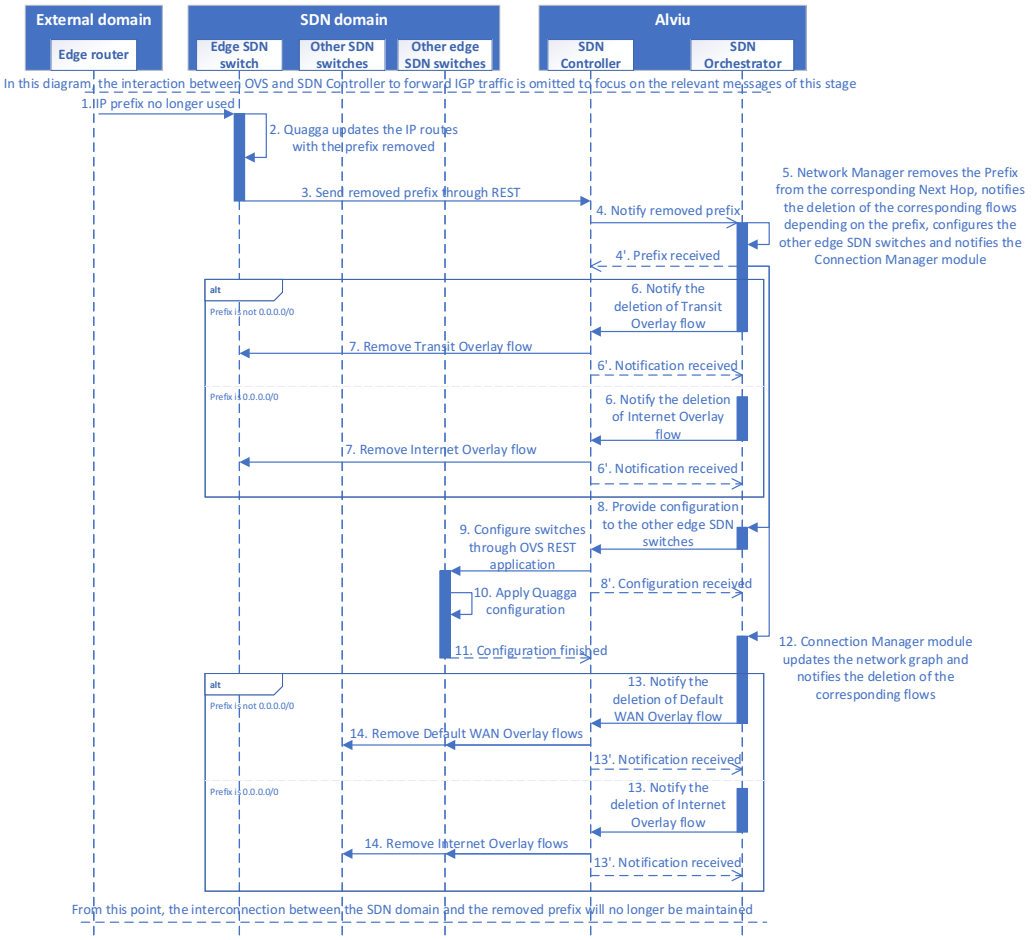

Figure 40: Prefix deletion workflow.

And finally, it may happen that a particular **prefix** could be **learned** from **different external domains**, being then received in Alviu more than once. To handle this, a **status** attribute is added to the **Prefix** entities, which could be either **active** for the **first prefix received**, to which the prefix learning workflow would be applied, or **backup** for the same prefixes learned afterwards, which would **not** be **reachable until** the **active prefix** is **deleted**.

In that moment, the **backup prefix activation workflow** would be applied, being described in Figure 41. Again, this workflow will not be explained in detail, as it is practically equal to the prefix learning workflow, but in this case the process will be **triggered** after **removing a prefix** that is **repeated** in Alviu.

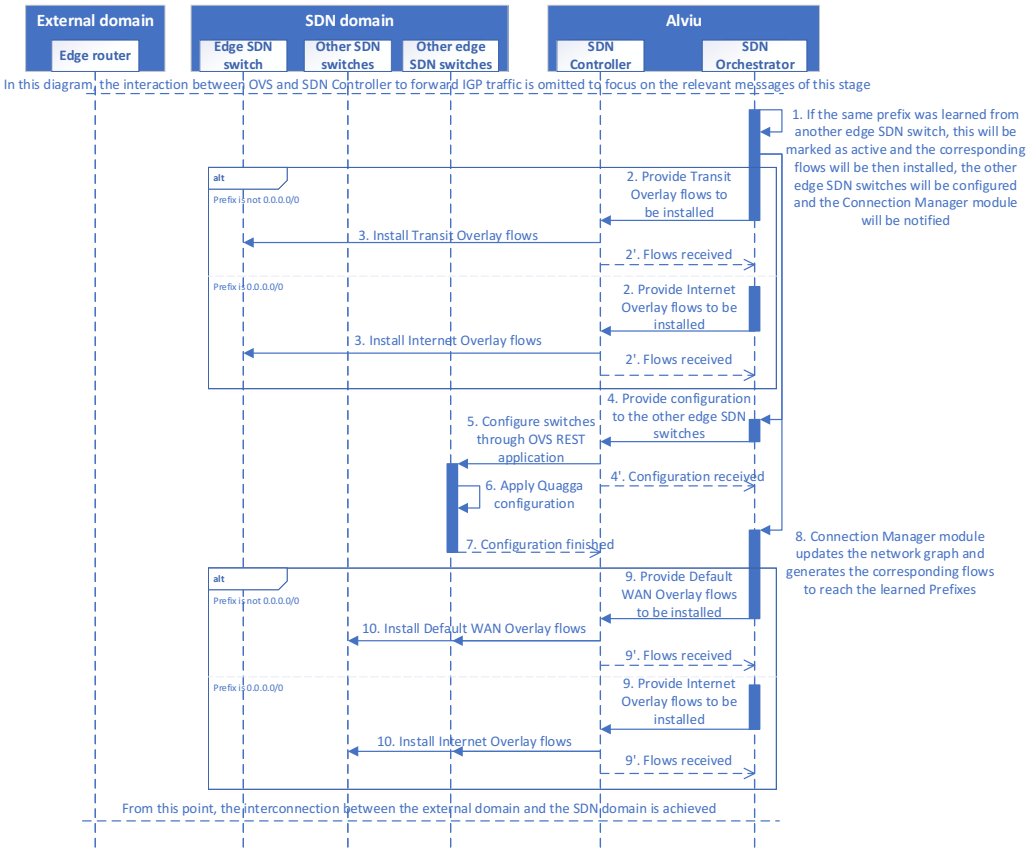

Figure 41: Backup prefix activation workflow.

6.2. Load Balancing with Dual Link Between SDN Switches

6.2.1. Use Case Overview

This particular scenario is presented when there are **more than one logical connections** between **WAN-type SDN switches** (*i.e.* switches that interconnects SDN domains), having then the opportunity to **balance the traffic** between these links according to a given **algorithm**. Note that these logical

connections can go through **separate physical links** for each case or **using the same physical link** for a set of logical tunnels, depending on the case.

A practical **example** of this can be seen in Figure 42, where there are two SDN branches, A and B, interconnected with a double-link connection between a pair of WAN-type SDN switches.

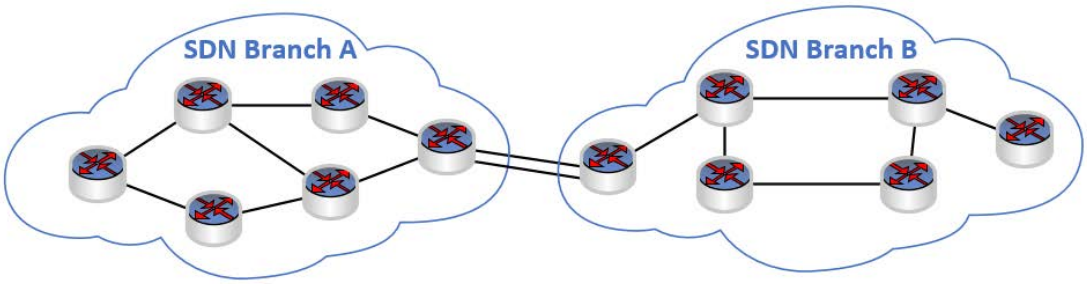

Figure 42: Practical scenario with switches connected by more than one link.

This configuration has some **impacts** in different aspects of the platform, such as the **design** of the **SDN switches** or the introduction of a new, parallel **workflow** to handle the traffic that will trigger the creation of the proper flows to manage the load balancing between switches. These issues will be discussed with more depth in Sections 6.2.2 and 6.2.3, respectively.

6.2.2. Updates Needed in OVS-based Switches

Although the changes needed in the SDN switches to **enable** this use case are less compared to the use case related to the interconnection with external domains, some new modules and configurations are needed.

Obligatorily, a **Monitoring agent** must be deployed in the switches, being connected to the **Monitoring Module** present in the **SDN Controller**, as depicted in Figure 25, to exchange information about the **traffic flows** handled by the switch.

Moreover, in case of using **several physical links** to perform the load balancing, the corresponding **physical ports** must be tagged as **WAN ports**, also requiring the same configuration based on a **Linux Bridge** and a **pair of veth interfaces** as commented in Section 6.1.2. However, as explained in Section 6.2.1, it is possible to have a load balancing based on **logical connections** established through a **single physical link**, thus only needing **one WAN port** in that case.

In Figure 43, it is presented an example of a **4-port SDN switch** based on OVS, including Quagga for the interconnection with external domains with the configuration already explained in Section 6.1.2, and also integrating the **Monitoring agent** as a new Linux module to be activated. This **agent** is logically **connected** to the **Linux Bridges** used in the WAN ports; in this case, as there are two WAN ports, it is connected to the two Linux Bridges depicted.

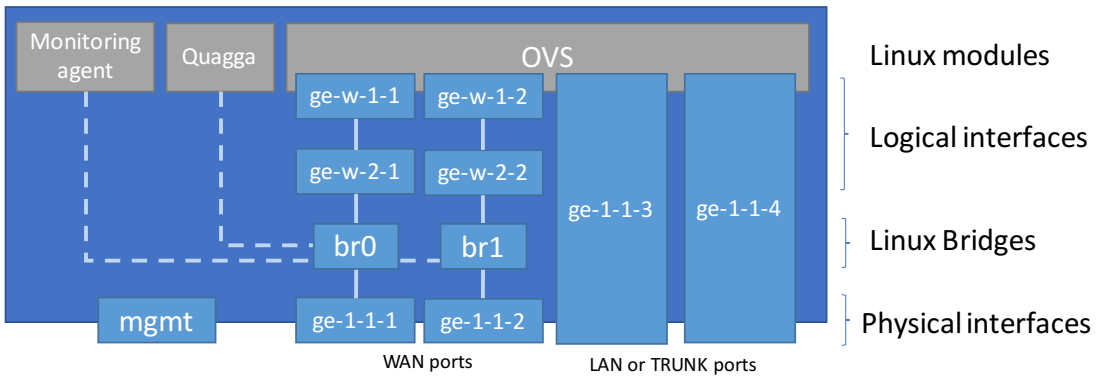

Figure 43: Update of the OVS-based switches to include the components needed for the load balancing use case.

Finally, a modification needed due to a problem found during the development phase of this feature was the fact that **dual-link SDN switches** whose **WAN ports** belongs to the **same network** forward traffic based on the **forwarding table** installed in the **Linux kernel**, even though the SDN Controller installs the flows correctly. And, as both WAN ports belongs to the same network, it is used **only one of the interfaces** for sending the traffic to the network. This was solved with the use of ***iptables*** in the **PREROUTING** and **OUTPUT chains** for **marking** the **tunneling protocols** used (GRE, ESP for IPsec, etc.) with a specific **key value**, which is **different** for each **link** of the dual-link path. so the traffic is **forwarded** for one link or another depending on the **key value** afterwards. The key value can be included when defining a **Connection Intent**.

6.2.3. System Workflow

Before starting with the explanation of the workflow of this use case, an **example** of a simple scenario will be firstly presented in order to clarify some aspects related to the workflow itself. This can be seen in Figure 44, where there are three branches, with one SDN switch per branch, and where there are **two branches interconnected** with a **dual link** between switches.

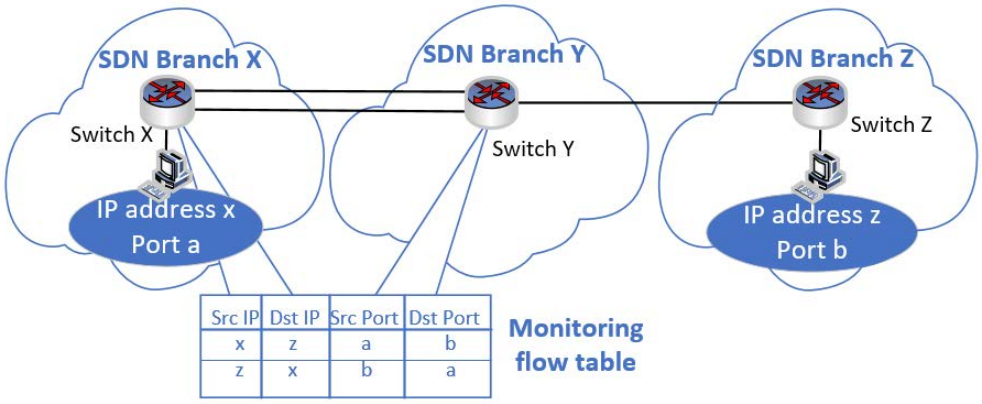

Figure 44: Example of traffic flows captured in a scenario with dual link between switches.

Note that a **Monitoring flow table** is presented at the bottom of the picture, describing an already established **traffic flow** between the hosts from branches X and Z in both senses of communication. The first thing to have in mind is that this table is **only managed** by **switches with dual links**, so the SDN switch Z will not require the Monitoring agent on it.

This information is gathered by **Alviu** by doing a **continuous polling**, every so often, to the **SDN switches** with **Monitoring agents** installed. From this point, the load balancing workflow is started, as presented in Figure 45:

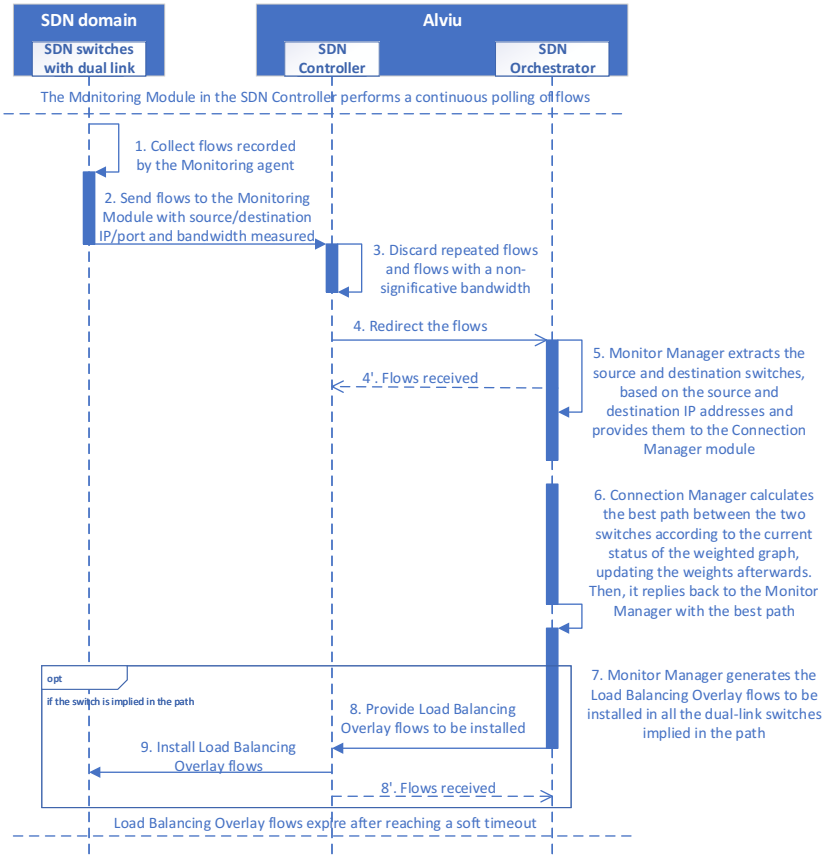

Figure 45: Load balancing workflow.

The full description of the **messages** implied in this workflow is the following:

1. The **Monitoring agent** of the SDN switches with dual links are continuously **collecting traffic flows** if detected, registering **networking information** such as the source and destination **IP addresses** and **ports** or the **bandwidth** measured, which can be used afterwards by Alviu to compute the flows related to load balancing.

2. As commented before, **Alviu polls** the **Monitoring agent** to obtain **new traffic flows**. In that case, they are **sent** to the **Monitoring Module** of the **SDN Controller**, using specific protocols related to monitoring purposes (*e.g.* sFlow).

3. In the **Monitoring Module**, there is a **process** running that **detects** if each **flow** captured is **new or old**, checking if it has been received previously or not (for example, the same traffic flow could be detected in the two switches connected by a dual link), only **conserving the new ones**. Also, it **discards flows** whose **bandwidth** is **not meaningful**, as it could be punctual traffic exchanged between the switches that may introduce useless flows in the SDN switches. A threshold of 1 Kbps is usually set, as it could be considered the "limit.ºf a significant value in this kind of networks.

4. The **filtered flows** are then sent to the **Monitor Manager** module of the **SDN Orchestrator**, which will act consequently. This message is **replied** by the orchestrator with message (4′).

5. According to the networking information received, the **Monitor Manager** module is able to **extract** the **source and destination switches** implied in each flow traffic, sending them to the **Connection Manager** module to perform the calculations over the graph.

6. In this point, after calculating the **best path** between the two endpoints, **two cases** may happen: (*i*) if the **traffic flow** involves a path in which **no dual links** are present, it will be directly **rejected**[13] and the workflow would finish in this point, or (*ii*) in the **best path**, there are **dual-link switches** involved, so it is forwarded to the **Monitor Manager**. First of all, note that **only one best path** will be obtained because it is a **weighted graph**, where the **weights** of each link are **updated** with every **new load balancing flow** installed, according to a given **algorithm**.

   In the current implementation, a **Round Robin algorithm** is used, so that the **first flow** will use the **first link** between switches (this being the default behaviour when the weights in the dual links are the same), **incrementing** the value of **its weight** in one unit. As a result, the **second flow** will use the **second link**, as its weigth value is less than the weigth value of the first link, **updating** again its **weight** afterwards, and so on and so forth.

7. The **Monitor Manager**, after receiving the best path, **generates** each **Load Balancing Overlay flow** to be installed in the corresponding **SDN switches with dual links** present in the **path**, achieving that the traffic is sent through the **same link** in **both senses of communication**.

8. Then, the **data** needed to generate the **Load Balancing Overlay flows** (*i.e.* source/destination IP/port, switch identification and output port) is sent to the **SDN Controller**, which **replies back** to this request with message (8′) to confirm that it has received the data.

9. Finally, the **Load Balancing Overlay application installs** the **flows** in the **switches** with a **soft timeout** of 30 seconds, so the flows are **deleted** after spending that **time without receiving that traffic flow**. This is done to avoid wasting resources in traffic flows that may be ephemeral.

[13] This interaction has not been included in the workflow diagram so as not to complicate it more.

6.3. Proof of Concept

The features commented in Sections 6.1 and 6.2 will be tested in a particular **testbed**, showing all the **capabilities that Alviu** can offer to **manage** a set of **SDN domains** with **advanced configurations**.

6.3.1. Testbed Setup

The **testbed** used for doing this proof of concept is the same than used in Chapter 5.4.1, using an **Ubuntu Server 16.04 LTS virtual machine** [90], with 12 vCPU and 12 GB of RAM, deployed in a server virtualized with *Proxmox* [89], which is equipped with 40 Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20 GHz and 128 GB RAM. In the virtual machine, **Alviu** has been provisioned in a ***Dockerized* format**, deploying both the SDN Orchestrator and the SDN Controller with *Docker* containers [92].

The **scenario**, deployed with the **Continuous Integration** platform, is fully described in Figure 46, also including the **networking configuration** for each link. The topology is quite similar to the example shown in Figure 44, with **three SDN branches** deployed in a **star topology**, each of them having only **one SDN switch** and **one host**, and with a **dual link** between the switches from **branches X and Y**. There are also some **external domains**: SDN branch X is connected to two external domains, and SDN branch Z to another two. Note that there are **two external domains**, the external domains C active and C backup, whose **IP prefixes** (172.16.200.0/24) are **repeated**, in order to test the active-backup configuration of IP prefixes. Moreover, the external domains A and B have another interconnection with other **external domains** related to the connection to the **Internet**, because the **router** of these domains will propagate the **default prefix**. Again, as there are two domains announcing the default route, that prefix will be received **twice**, acting one as **active** and the other one as **backup**. All the **routers** used in both **external and Internet domains** have **Quagga** installed.

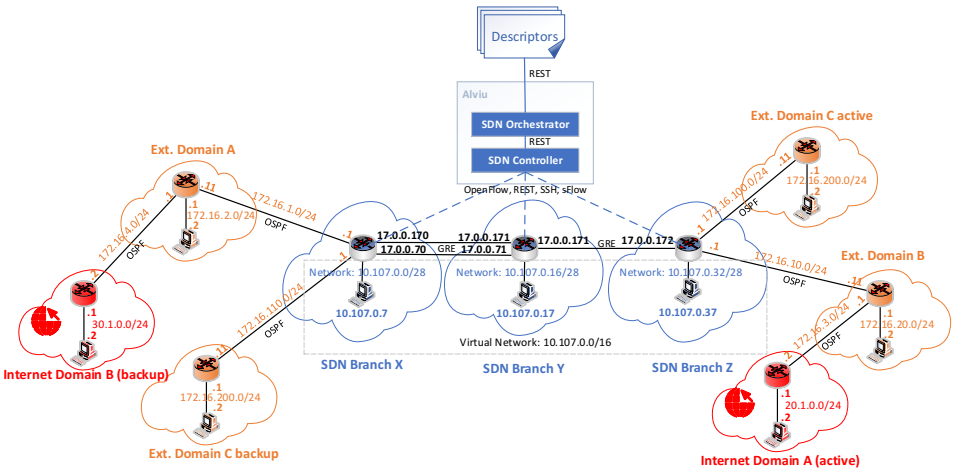

Figure 46: Testbed built with the Continuous Integration environment to do the proof of concept.

After presenting the scenario, the different **stages** of the proof of concept will be briefly described, also showing the **flow rules** installed or updated on each phase.

6.3.2. Basic Connectivity Between SDN Switches

When **Branch and Connection Intents** are **deployed** and **activated** (*i.e.* all the entities related to each intent are marked as active), while the **routers** from external domains do **not establish IGP sessions** with the edge SDN switches to propagate the network prefixes, the **topology** will remain in a **basic connectivity** status, so that the **hosts of the SDN branches** can **exchange traffic** between them.

This connectivity is achieved thanks to **two type of flows**: (*i*) a **Default WAN Overlay flow** installed **for each network** from **other SDN domains**, and a **Per Host LAN Overlay flow** to reach **each host** of the **branch itself**. For instance, in the case of the SDN branch X, the corresponding flows are presented in Figure 47, which is a capture from ONOS (*i.e.* the SDN Controller) GUI. In this case, the traffic related to the Default WAN Overlay flows are sent through the port number 100, which is one of the GRE port which connects branches X and Y (because there is a dual link), and the traffic directed to the host of this branch is managed by the Per Host LAN Ovelay flow, which sends the traffic to the port 2 (which is a LAN port).

SELECTOR	TREATMENT	APP NAME
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.32/28	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.16/28	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.7/32	imm[ETH_SRC:00:00:00:11:11:11, ETH_DST:00:00:00:00:0A:01, OUTPUT:2], cleared:false	telca.alviu.perhostlanoverlay

Figure 47: Flows related to basic connectivity in the SDN switch of branch X.

6.3.3. Interconnection with External Domains

When all the **Quagga** modules of the **routers** from external domains are **activated**, the **workflow** explained in Section 6.1.3 is then triggered, and after reaching a **convergence point**, some **new flows** are added to each SDN switch, as presented in Figure 48 in the case of branch X, and in Figure 49 for branch Y. Starting with **branch X** (the same can be applied to branch Z), the **group of flows** that can be distinguished in that moment are:

1. These group of flows are the same than presented in Figure 47, related to the **basic connectivity between SDN branches**.

2. This second group of flows are the **Default WAN Overlay flows** that allows to **reach the prefixes** from **external domains** learned in **other edge SDN switches** (*i.e.* the branch Z switch). Note that all the traffic is sent to the port 100, so that the traffic is forwarded to branch Y.

3. Then, this group of flows represents the **Transit Overlay flows** to reach the **prefixes** from **external domains directly attached** to this switch, so the **output** is a **TRUNK port** (in this case, the same port, number 4, is used for both cases).

4. Finally, as a **default route** is announced, an **Internet Overlay flow** is installed. Note that the **traffic allowed to reach the Internet** can only come **from** the **SDN domains**, as the source IP address is matched with the virtual network defined in the scenario, so that the **traffic from external domains cannot traverse** the **SDN domains** to go to another external domain which allows them to **reach the Internet**. This is done for **security** purposes, to avoid having a SDN network dedicated exclusively to exchange traffic with the Internet. However, in any case, this behaviour could be **changed** but just **removing the source IP address match** in the **flow rule**, confirming the **ease of changing settings** in this kind of platforms.

	SELECTOR	TREATMENT	APP NAME
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.7/32	imm[ETH_SRC:00:00:00:11:11:11, ETH_DST:00:00:00:00:0A:01, OUTPUT:2], cleared:false	telca.alviu.perhostlanoverlay
1	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.32/28	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.16/28	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.200.0/24	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
2	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.20.0/24	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.3.0/24	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
3	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.4.0/24	imm[ETH_DST:8A:C1:18:93:2B:B5, OUTPUT:4], cleared:false	telca.alviu.transit-overlay
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.2.0/24	imm[ETH_DST:8A:C1:18:93:2B:B5, OUTPUT:4], cleared:false	telca.alviu.transit-overlay
4	IN_PORT:7, ETH_TYPE:ipv4, IPV4_SRC:10.107.0.0/16	imm[OUTPUT:100], cleared:false	telca.alviu.internet-overlay

Figure 48: Flows that includes the interconnection with external domains in the SDN switch of branch X.

In the case of the switch of **branch Y**, the flows are slightly **different**, as it is not directly connected to any external domain:

1. First of all, the flows related to **basic connectivity** with other SDN branches are presented.

2. In this case, to reach all the **prefixes learned from external domains**, as it is not directly connected to the external domains, a **Default WAN Overlay flow** is installed **for each prefix** learned, specifying the GRE port to send the traffic to the next hop in each case.

3. In the same case than in branch X, the **Internet Overlay flow** is defined.

	SELECTOR	TREATMENT	APP NAME
1	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.17/32	imm[ETH_SRC:00:00:00:11:11:11, ETH_DST:00:00:00:00:0A:02, OUTPUT:2], cleared:false	telca.alviu.perhostlanoverlay
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.0/28	imm[OUTPUT:101], cleared:false	telca.alviu.default-wan-overlay
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.32/28	imm[OUTPUT:103], cleared:false	telca.alviu.default-wan-overlay
2	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.4.0/24	imm[OUTPUT:101], cleared:false	telca.alviu.default-wan-overlay
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.2.0/24	imm[OUTPUT:101], cleared:false	telca.alviu.default-wan-overlay
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.200.0/24	imm[OUTPUT:103], cleared:false	telca.alviu.default-wan-overlay
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.3.0/24	imm[OUTPUT:103], cleared:false	telca.alviu.default-wan-overlay
	IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.20.0/24	imm[OUTPUT:103], cleared:false	telca.alviu.default-wan-overlay
3	IN_PORT:7, ETH_TYPE:ipv4, IPV4_SRC:10.107.0.0/16	imm[OUTPUT:103], cleared:false	telca.alviu.internet-overlay

Figure 49: Flows that includes the interconnection with external domains in the SDN switch of branch Y.

Finally, it may happen that a **prefix** is **no longer announced**, **removing** the related **flows** from the switches consequently. And, if these prefixes are **repeated** in the scenario, the workflow presented in Figure 41 will be then started, updating the **status** of these prefixes to **active** and **installing the new flows** in the switches.

For example, if Quagga modules from the external domain C active and the Internet domain A are stopped, then the prefixes learned from the external domain C backup and the Internet domain B, respectively, will be marked as active, and the new flows will be then installed. This change is reflected in Figure 50 for the case of the switch of **branch X**, where the **IP prefix 172.16.200.0/24** and the **default route** are now reached through the **TRUNK port**, so that the first one is modeled with a **Transit Overlay flow**, and the second one is still an **Internet Overlay flow**, but including the **change of the destination MAC address** with the edge router's one.

SELECTOR	TREATMENT	APP NAME
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.7/32	imm[ETH_SRC:00:00:00:11:11:11, ETH_DST:00:00:00:00:0A:01, OUTPUT:2], cleared:false	telca.alviu.perhostlanoverlay
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.32/28	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:10.107.0.16/28	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.200.0/24	imm[ETH_DST:F2:71:5D:D8:07:64, OUTPUT:4], cleared:false	telca.alviu.transit-overlay
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.4.0/24	imm[ETH_DST:8A:C1:18:93:2B:B5, OUTPUT:4], cleared:false	telca.alviu.transit-overlay
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.20.0/24	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.3.0/24	imm[OUTPUT:100], cleared:false	telca.alviu.default-wan-overlay
IN_PORT:7, ETH_TYPE:ipv4, IPV4_DST:172.16.2.0/24	imm[ETH_DST:8A:C1:18:93:2B:B5, OUTPUT:4], cleared:false	telca.alviu.transit-overlay
IN_PORT:7, ETH_TYPE:ipv4, IPV4_SRC:10.107.0.0/16	imm[ETH_DST:8A:C1:18:93:2B:B5, OUTPUT:4], cleared:false	telca.alviu.internet-overlay

Figure 50: Update of the flows related to the interconnection with external domains in the SDN switch of branch X.

6.3.4. Testing Load Balancing Capabilities

In case of establishing **traffic flows** between **hosts** that **traverse the dual link** between switches of branches X and Y (*e.g.* by using iperf to test it), the **load balancing workflow** explained in Section 6.2.3 is then triggered, installing the corresponding ephemeral **Load Balancing Overlay flows** that allows to do the load balancing.

For example, Figures 51 and 52 shows some **Load Balancing Overlay flows installed** in SDN switches from branches X and Y respectively, checking that **they are balanced** by following a Round Robin algorithm (*i.e.* the number of times that an output port is used is equally distributed), and that the **flows are paired off between switches** to always use the same link in both senses of communication (*e.g.* the first flow in the switch from branch X is paired off with the third flow in the switch from branch Y).

SELECTOR	TREATMENT	APP NAME
IN_PORT:7, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.107.0.7/32, IPV4_DST:172.16.200.2/32, TCP_SRC:30000, TCP_DST:56580	imm[OUTPUT:100], cleared:false	telca.alviu.load-balancing-overlay
IN_PORT:7, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.107.0.7/32, IPV4_DST:10.107.0.37/32, TCP_SRC:30000, TCP_DST:40292	imm[OUTPUT:100], cleared:false	telca.alviu.load-balancing-overlay
IN_PORT:7, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.107.0.7/32, IPV4_DST:10.107.0.17/32, TCP_SRC:30000, TCP_DST:37236	imm[OUTPUT:104], cleared:false	telca.alviu.load-balancing-overlay
IN_PORT:7, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.107.0.7/32, IPV4_DST:172.16.20.2/32, TCP_SRC:30000, TCP_DST:35970	imm[OUTPUT:104], cleared:false	telca.alviu.load-balancing-overlay

Figure 51: Load Balancing Overlay flows in the switch from branch X.

SELECTOR	TREATMENT	APP NAME
IN_PORT:7, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:172.16.20.2/32, IPV4_DST:10.107.0.7/32, TCP_SRC:35970, TCP_DST:30000	imm[OUTPUT:105], cleared:false	telca.alviu.load-balancing-overlay
IN_PORT:7, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.107.0.17/32, IPV4_DST:10.107.0.7/32, TCP_SRC:37236, TCP_DST:30000	imm[OUTPUT:105], cleared:false	telca.alviu.load-balancing-overlay
IN_PORT:7, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:172.16.200.2/32, IPV4_DST:10.107.0.7/32, TCP_SRC:56580, TCP_DST:30000	imm[OUTPUT:101], cleared:false	telca.alviu.load-balancing-overlay
IN_PORT:7, ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.107.0.37/32, IPV4_DST:10.107.0.7/32, TCP_SRC:40292, TCP_DST:30000	imm[OUTPUT:101], cleared:false	telca.alviu.load-balancing-overlay

Figure 52: Load Balancing Overlay flows in the switch from branch Y.

6.3.5. Testing Network Slicing Features with Firewall and QoS Policies

Finally, some **value-added capabilities** can be added with the definition of the **Policy Intents**, which can be related to the achievement of **network slicing objectives** (*e.g.* guarantee a given throughput for a traffic flow, isolate traffic flows between them, etc.).

In this case, two examples will be shown. The first one is related to the **firewalling capabilities of Alviu**. In Figure 53, some **Firewall rules** can be seen, meaning that all the traffic exchanged with the network of the SDN branch Y is rejected excepting the traffic directed to the TCP port number 30000.

SELECTOR	TREATMENT	APP NAME
ETH_TYPE:ipv4, IP_PROTO:6, IPV4_DST:10.107.0.16/28, TCP_SRC:30000	imm[ETH_SRC:00:00:00:11:11:11, ETH_DST:00:00:00:22:22:22, OUTPUT:8], cleared:false	telca.alviu.firewall
ETH_TYPE:ipv4, IP_PROTO:6, IPV4_SRC:10.107.0.16/28, TCP_DST:30000	imm[ETH_SRC:00:00:00:11:11:11, ETH_DST:00:00:00:22:22:22, OUTPUT:8], cleared:false	telca.alviu.firewall
ETH_TYPE:ipv4, IPV4_SRC:10.107.0.16/28	imm[NOACTION], cleared:false	telca.alviu.firewall
ETH_TYPE:ipv4, IPV4_DST:10.107.0.16/28	imm[NOACTION], cleared:false	telca.alviu.firewall

Figure 53: Example of firewall rules installed in the switch of branch Y.

And secondly, a **QoS example** is also presented for branch Y. In Figure 54, it can be seen that the traffic exchanged with the network of the SDN branch Y using the TCP port number 30000 is tagged with **DSCP**, which is used afterwards by the queues configured in the OVS to **guarantee a particular bandwidth** to that traffic flow, according to what it has been defined in the **Policy Intents**.

SELECTOR	TREATMENT	APP NAME
ETH_TYPE:ipv4, IP_DSCP:0, IP_PROTO:6, IPV4_SRC:10.107.0.16/28, TCP_DST:30000	imm[IP_DSCP:6, QUEUE{queueId=0}, OUTPUT:6], cleared:false	telca.alviu.qos-overlay:qos
ETH_TYPE:ipv4, IP_DSCP:0, IP_PROTO:6, IPV4_DST:10.107.0.16/28, TCP_SRC:30000	imm[IP_DSCP:6, QUEUE{queueId=0}, OUTPUT:6], cleared:false	telca.alviu.qos-overlay:qos
ETH_TYPE:ipv4, IP_DSCP:0	imm[IP_DSCP:6, OUTPUT:6], cleared:false	telca.alviu.qos-overlay:default

Figure 54: Example of QoS rules installed in the switch of branch Y.

6.4. Summary

This Chapter is understood as an **application** to all the **general topics** presented in Chapter 5 regarding **Alviu**, confirming that this orchestration solution is **flexible** enough to be able to achieve the integration of specific **use cases**, with **different purposes**.

Specifically, **two use cases** have been fully described in this Chapter, from the **need** each of them tries to cover to the **workflow** followed by the platform to achieve the desired **configuration**, also including some **technical specifications** about the **internal design** of the **SDN switches** to fulfill these requirements.

These two cases are, in summary, the **interconnection** with legacy, **external domains**, which is one of the most relevant features that Alviu implements, and being complemented by the **load balancing** service between **switches interconnected** by **more than one connection**.

Finally, to confirm that the **workflow** has been **correctly implemented** in Alviu, a **proof of concept** has been fully described, including both **use cases** and other ones related to **other features** of Alviu (*i.e.* implementation of **policies**).

Part 3. New Virtualization Techniques

7

Integration of the Serverless Paradigm within 5G Networks

The fifth-generation (5G) of mobile networks will tackle the current and **future trends** of **data consumption** while fulfilling stringent **requirements** on delay, reliability, and throughput, among others. In order to provide **customized services** that efficiently meet these requirements, **mobile networking** is adopting two key **trends** from **computer science**, which are ***softwarization*** and ***modularization***. The **first one** is the ability to **operate fully-fledged networks** through **software components**, while the **second one** consists of defining and instantiating **re-usable and highly focused Virtual Network Functions (VNF)**, which can be eventually interconnected with orchestration solutions such as Alviu, already presented in Chapters 5 and 6. Thanks to these, network providers can **move away from highly specialized hardware** solutions and benefit from building **deployments** based on **general-purpose hardware architecture**, running re-usable software components.

The **adoption** of *softwarization* and *modularization* by the **mobile networking community** provides significant **benefits**; such as **flexibility**, **improved resource efficiency** or **commoditization**, eventually **enabling** the **network slicing** paradigm [111]. Furthermore, the **availability of solutions**, both commercial and open-source, implementing these two technologies, are also **increasing**, which is caused by the relative **maturity** of these technologies related to the Cloud Computing success.

Despite this, this transition towards *softwarization* and *modularization* also implies a **non-negligible cost**, *e.g.* the **management overhead** or the **redesign** of certain **functions** that now run as software components instead of as hardware implementations [112]. Moreover, in terms of **projects and implementations available** in the **mobile networking ecosystem**, it is true that its evolution is going **late** compared to the **Cloud Computing ecosystem**, in which **there are already technologies** adapted to this trend.

This is the case of the **serverless architectures**, also known as **Function as a Service (FaaS)**, a novel paradigm that appeared a few years ago supporting

an **extremely liquid approach** to **scalability** and **resource usage** [52][113]. In fact, with this approach, the **software** is **decomposed** into its **minimum building blocks** (*i.e.* **functions**), **maximizing scalability, flexibility and resource efficiency**, being this last topic of paramount importance in the multi-tenant scenarios envisioned in 5G. This way, a tenant would be able to **create "on-demand" calls** to specific platform- and server-independent **functions** that are then executed by an infrastructure provider. This **flexibility** also allows to easily **reallocate** the **functions** along the **infrastructure** if required in order to **optimize** the overall **system's consumption**, using **monitoring solutions** such as the one presented in Section 3 to constantly **check** the **performance** of these **functions** and to **trigger** their **reallocation**.

As a result, this Chapter delves into the **serverless paradigm** applied to **mobile architectures**, **merging** the new **trends** on the **Cloud Computing evolution** into the **mobile networking community**. Apart from extending the main **characteristics** of this new technology, it will be also considered its **integration** into a well-known **platform** related to **5G networks**, which is the **Monitoring platform** already presented in Section 3, then discussing the **lessons learned** from this process.

The main **topics** that will be discussed in this Chapter are the following:

- As a first step, Section 7.1 presents the **need for serverless computing** as a key **candidate technology** for the **next generation of Network Function Virtualization (NFV)**, also discussing the **advantages and challenges** introduced by this approach.
- Then, Section 7.2 introduces the usage of the **serverless paradigm** in the **Monitoring platform** already presented in Section 3, describing the **transition** from the **original implementation** to a **full serverless-based platform**.
- Taking into account the new serverless implementation of the Monitoring platform, Section 7.3 **validates** its correct behaviour in a **testbed** which uses some of the **tools** related to the **serverless paradigm**, presented in Section 2.3.
- And finally, Section 7.4 **summarizes and concludes** the work related to this introduction of the serverless paradigm.

7.1. Serverless Mobile Architectures' Overview

In this Section, the transition towards a **serverless mobile network architecture** will be analyzed, introducing the **concept** firstly and then discussing the **advantages** and the **challenges to address** to achieve this evolution.

7.1.1. Concept

To describe the serverless mobile network architectures, **Radio Access Network (RAN) functions** will be used as examples, as they provide the **most difficult scenario** for **serverless architecture** given their tight execution

constraints. This way, in Figure 55, the **evolution** of the different **architectures** to support a **mobile service** from the RAN perspective is reflected.

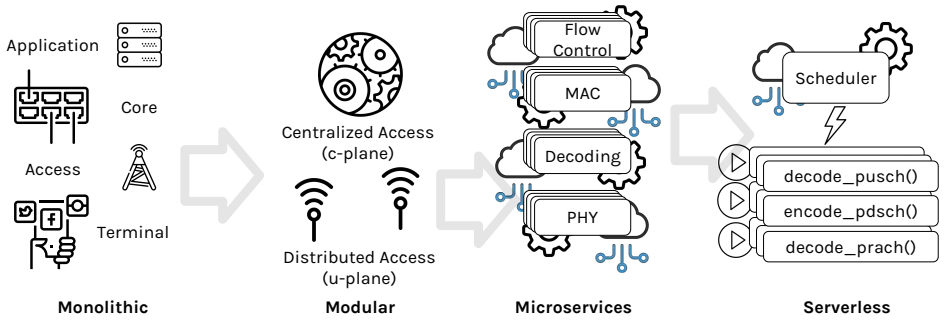

Figure 55: Mobile network architecture evolution.

In this case, **four architectures** can be distinguished:

- Firstly, the leftmost subfigure depicts the traditional **monolithic** paradigm (*e.g.* 4G networks), where **functions** are implemented in **specialized pieces of equipment**. In this case, **software** and **hardware** are **tightly coupled**, and it is not uncommon that different functions are indissolubility associated to the same piece of equipment, *e.g.* the Serving Gateway (S-GW) and Packet data network Gateway (P-GW).

- The next subfigure shows a **modular** network architecture, represented by the **Cloud-RAN (C-RAN) paradigm**, where some **control functionality** traditionally associated with the antenna (*i.e.* the scheduling algorithm) is **re-located** to a **central server**. This change constitutes a shift from the monolithic approach, with some **functions "released"** from their **traditional association** to **monolithic pieces** of **hardware**. These functions are now logically **different pieces of software**, whose **execution** can be placed in **different parts** of the **network**.

- In the case of the **microservices** architecture, it pushes the modular paradigm further by **decomposing** the **building blocks** into **sub-modules**. Note that this is a **logical division** and that the actual **implementation** of the architecture needs to **accommodate** based on specific **use-case requirements**, thus eventually resulting in fewer or more pieces of software. For the case of the RAN, this results in the protocol stack now being logically divided into physical layer processing, decoding, encoding, MAC, flow control, etc., each of them **running** in an **independent execution environment** and **connected** through **synchronization APIs**. This allows an **easier scaling** over a **finer resource assignment strategy**, which eventually leads to **better resource utilization**. Furthermore, some very recent proposals are pushing for **microservice-based core network functions** [114], showing that this **increased modularity** in the VNF design is catching momentum.

- And finally, the desired **serverless** mobile architecture would be composed by **atomic functions** that can **run independently** on a Cloud infrastructure. This independence contrasts with the tight coupling across functions

in the other architectures, with strict timing considerations between modules. In a serverless approach, **functions** are **dis-aggregated** from the **main scheduling logic** and **executed** in the **most appropriate server available**. As Figure 55 illustrates, for the case of User Plane Functions envisioned, for instance, the decoding of different Modulation and Coding Scheme could be made by different functions that could run in different executors, provided that **some "loose synchronization"** is **guaranteed**.

7.1.2. Advantages

Introducing the serverless operation brings several **advantages** to the **network operation**. These are mostly related to the **cost efficiency** of the resources needed to provide a given network service; that is, the **heavy load** caused by tasks such as, for example, baseband processing, can be **pulverized** into **atomic operations** that can, in turn, be **dynamically orchestrated** (*i.e.* scaled in and out) with a **very precise match** to the **real load**.

In particular, based on the reasoning introduced in [52], the following **advantages** of serverless mobile networking are detected:

- **No server management:** in the serverless paradigm, the **functionality** carried out by a VNF is **broken** into **very fine execution environments** (*i.e.* **functions**) that **do not need** to directly undergo into the **classic lifecycle management** (instantiation, run-time and decommissioning), but rather be **scaled** according to the **real load** and with a very **fast pace** in a **"message broker" fashion**. By moving this complexity to the network orchestration, this allows increasing the **commodification** of the **network** with a clear **separation** between the **infrastructure** and the **services** orchestrated therein.

- **No idling: operators usually provision** the network based on the **peak load**. This is very **inefficient** at all network layers; at the access level, needless to say, but also at more centralized levels in which VMs or containers may be **underused** or even **idling** in trough loads. With the serverless paradigm, **execution engines** are **spawned** and **operated** just **when and where** they are **needed**. This is key for **minimizing resource wastage** in the network operation.

- **Liquid scalability:** this is achieved by providing the **highest *modularization* level**. As a result, **specific functions** of a VNF can be **scaled** according to the **real demand**, avoiding the scaling of the full VNF instead, and achieving the liquid scalability depicted in Figure 56.

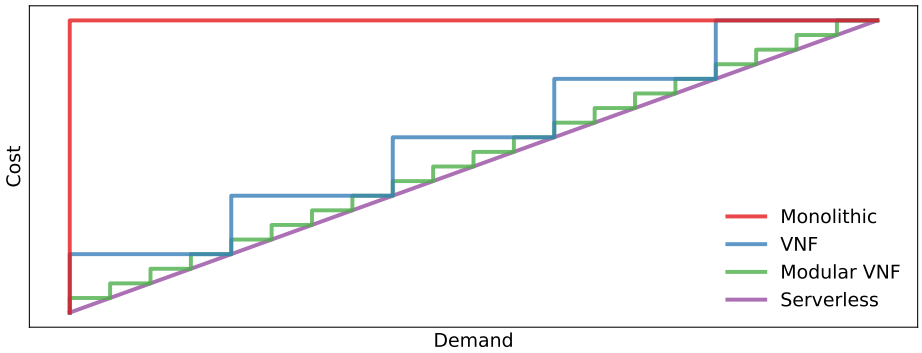

Figure 56: Representation of the liquid scalability.

As a reference for this concept, in [50], the **cost** in terms of **resource overhead** of **deploying** and **operating** the **infrastructure** needed to support multi-service networks was measured. In that study, it was showed that the **efficiency** (*i.e.* the number of resources used by a not multi-tenant network compared to a multi-tenant one) was **very low** (15 %) for **edge resources** (*e.g.* spectrum, antennas) and only **slightly better** (65 %) when considering **core resources** (*e.g.* CPU in a Cloud data center). The study also showed that **only** with a **very dynamic network reconfiguration** it is possible to **improve** these figures up to 60 % and 90 %, respectively. So, achieving the **finest granularity** (the analysis of [50] is performed at byte level), such as the one envisioned by the serverless paradigm, will allow **achieving such extreme gains** in terms of **resource utilization**.

- **Continuous deployment:** with NFV based on serverless, the **DevOps paradigm** (which has been recently proposed for the network operation as well) is brought to an **extreme level**. Developers can **update** just very **specific parts of the code** (*i.e.* the functions) instead of fully-fledged VNFs, thus **reducing** the **time** to **deploy new functionalities**.

- **Pay-per-use network:** although the pricing model behind the **network slicing paradigm** is not clear yet, it is to be expected that, at least for the **software part**, it will follow a **classic approach** in which **tenants** are **charged** on the number of **CPUs**, the amount of **memory** and **bandwidth** used. With a **serverless** approach, instead, **tenants** can be **charged** on a **specific usage basis** (*i.e.* number of **times** and **duration** of each **function**), allowing for a **richer pricing model**.

- **Customization:** current **mobile network** technology provides only **limited customization**. For example, the currently envisioned resource models in 3GPP [115] target the **Network Slice as a Service** (NSaaS) paradigm, which is the telecommunication counterpart of the well-known Software as a Service (SaaS) paradigm employed in the Cloud Computing world. Under this model, **service providers** (or tenants) are allowed to **select**, from an operator Network Slice **portfolio**, some available **templates** (*e.g.* Enhanced Mobile Broadband). However, this provides **limited customizability** to

tenants, as the **network provider** still handles most of the **management part**. This effort may be **released** with the serverless approach, providing a **higher customization** with **new function-based applications** adapted to **different environments**.

- **New markets:** in addition to the advantages in terms of the cost-effectiveness of the system, a new mobile networking paradigm based on a serverless architecture would also introduce and strengthen **new markets**. Currently, the **lack** of a **technical solution** for very high customizability has the side effect of **hindering** the adoption of **new business models**. However, the **adoption** of a **technology** enabling such **customization** would **foster** its **adoption**.

7.1.3. Challenges to Address

To achieve the above advantages, the serverless paradigm needs to deal with the following **challenges**:

- **New VNFs:** the **current way** of **implementing VNFs** is still very **bound** to the **traditional way** of **implementing network functions**. Current solutions do not embrace *modularization*: many commercial products are softwarized but very **bounded** to the **hardware platform**, while **open-source initiatives** are practically mere **translations** of **hardware functionality** into **software modules**. To adopt the **serverless** approach, the way in which VNFs are designed needs to be **changed**, trying to **improve** the **speed execution** while **minimizing** the number of **resources** needed for their operation. For example, in the case of the RAN, as the radio functions are the most resource-consuming ones (considering resources of all kinds: spectrum, transport network, and computational resources [116]), the transition towards high modularity will be especially beneficial for such functions.

- **Scalable interconnections:** for the **execution** of such challenging **VNFs**, a new **environment** with **minimal overhead** is also required. This kind of environments needs to cope with **highly-dynamic deployments** with a **larger number** of **software components** (for each tenant, slice, and service, there might be **multiple software functions**). For example, from the **mobile data consumption point** of view, one of the objectives to be fulfilled is to achieve the **fastest data plane possible**, even though the original virtualization platforms were not designed with this goal in mind. To address this issue, apart from using **platforms** relying on technologies such as **SDN** or **NFV** to **interconnect VNFs** (*e.g.* Alviu, already presented in Chapters 5 and 6), the most common **approach** to achieve high performance has been ***kernel bypassing***, through technologies such as DPDK [117] and SR-IOV, but this makes the **management** of the VNF very **machine-dependent**, so it is only **valid** for **scenarios** with a relatively **small number** of **VNFs**. A possible **solution** to avoid this limitation is, for example, to **integrate** the **data path** back into the **kernel** with tools

like enhanced Berkeley Packet Filters (eBPFs) [118]. These are pieces of code that can be dynamically injected into the kernel at run-time through a programmable interface, which allow the VNFs' management running on top of the kernel holistically, controlling all the aspects such as their CPU, memory, etc. in a unified way.

- **Precise orchestration algorithms:** the serverless paradigm aims at the **most efficient service provisioning**, by accurately **adjusting** the **resources deployed** at any point in time to the **actual demand**. To benefit from this paradigm, it is essential to accurately **estimate** the **demand** required by a **service** and to **forecast** its **envisioned resource consumption**, to boost the multiplexing gains. To support this type of management, **two** main **building blocks** are required: (*i*) **technical solutions** to support **flexible** and **fast resource re-orchestration** at the **finest granularity**, and (*ii*) **Big Data techniques** that operate on **historical data** and **anticipate future trends**. The **former** should be achieved with the **first two challenges** (*i.e.* the use of **functions** instead of VMs, deployed in an **environment** with **minimal overhead** for being able to **scale**), while the **latter** requires the design of **new techniques**. For example, **data-driven techniques**, empowered with **deep learning solutions**[14], can be used to accurately **characterize** the **future demand trends** for a given **service**, this supporting a **proactive**, **efficiency-driven** and **fine-grained orchestration** of the **network** [120]. In this way, solutions like the one presented in Section 4.2.2, which described the integration between the Monitoring platform from Chapter 3 and the Data Analytics Framework proposed in [3], may fit in this approach.

7.2. Introducing Serverless Techniques in the Monitoring Platform

To fully describe the **implications** of **moving** to the **serverless** paradigm, the **5G EVE Monitoring platform** will be used as an example of a 5G-related system that is liable to be **transformed** into **serverless** in some of its main **building blocks**.

In this way, the **challenges** to address by the serverless paradigm proposed in Section 7.1.3 will be **present** in this transition towards a serverless design and implementation, taking into account the necessity of **new VNFs** working in an **scalable environment** with **precise orchestration algorithms** to use the compute resources available in an effective way.

7.2.1. Problems Found in the Legacy Architecture

As a summary, the **5G EVE Monitoring platform**, just as described in Section 3.2, was designed as a **modular architecture**, according to the classification proposed in Section 7.1.1. As a result, **two** main **building blocks**

[14]These technologies are currently being investigated by ETSI ENI [119] from the architectural point of view.

were defined: **the Data Collection Manager** (Figure 5) and the **Data Collection and Storage-Data Visualization** (Figure 6), with the objective of managing the implementation of the **publish-subscribe delivery system** and the **monitoring data collection, indexing and visualization mechanisms**, respectively.

To support these functionalities, a set of **handlers** were defined in both components, denoted as ***Python* logic** in both cases. These pieces of code are in charge of managing the **lifecycle** of the **topics** related to the **metrics** and **KPIs** to be **monitored** in a particular experiment, **triggering action** such as the creation of a topic in *Apache Kafka* and *Apache Kafka*, or the building of a Kibana dashboard for a given topic.

Although these handlers are not hard to understand in terms of workflow, it is true that they are somehow **attached** to the **deployment** in a **single server**, so that the transition towards a serverless approach cannot be directly applied, as there exist some **limitations** in the current implementation that, in case of not being reviewed and redesigned properly, they would prevent the platform from evolving to serverless effectively.

Due to this, the **transition towards serverless** is planned in **two stages**: first of all, **transforming both components** (*i.e.* DCM and DCS-DV) to fit in a **microservices architecture** (third architecture presented in 7.1.1), identifying the **core functions** executed on each ***Python* logic** that interact between them as a **chain**, and implementing them as **REST-based services**. In this way, the identification of the **service function chain** that describes the concatenated operations that are executed sequentially to obtain the desired results will be **fully described** in terms of these **REST services**.

After this, these services will be used as **base** to build the **serverless functions** that would finally **transform** the **Monitoring platform** in a **serverless-based architecture** (fourth and final architecture presented in 7.1.1.

7.2.2. Transformation from Legacy to Microservices Architecture

In this first stage, some **atomic functions** present on each ***Python* logic** from both the DCM and DCS-DV are **extracted** from them, being then **modeled and implemented** as **REST services**, exposing a **REST API** to be **accessed** from the different components involved in the monitoring workflow.

In the case of the **Data Collection Manager**, whose **new architecture** based on a **microservice's approach** is presented in Figure 57, **three new functions** have been modeled as REST-based services: (*i*) ***createKafkaTopic***, which implies the **creation of topics** in *Kafka*, (*ii*) ***deleteKafkaTopic***, related to the **deletion of topics** in *Kafka*, and (*iii*) ***listKafkaTopics***, an auxiliary module that allows to **check** if a given **topic** already **exists or not** in *Kafka*. All these **modules** are **handled** by the so-called ***DCM Topic LCM***, which is the **simplified version** of the ***Python* logic** present in the **legacy architecture**, being reduced in terms of functionality and complexity due to this decoupling exercise.

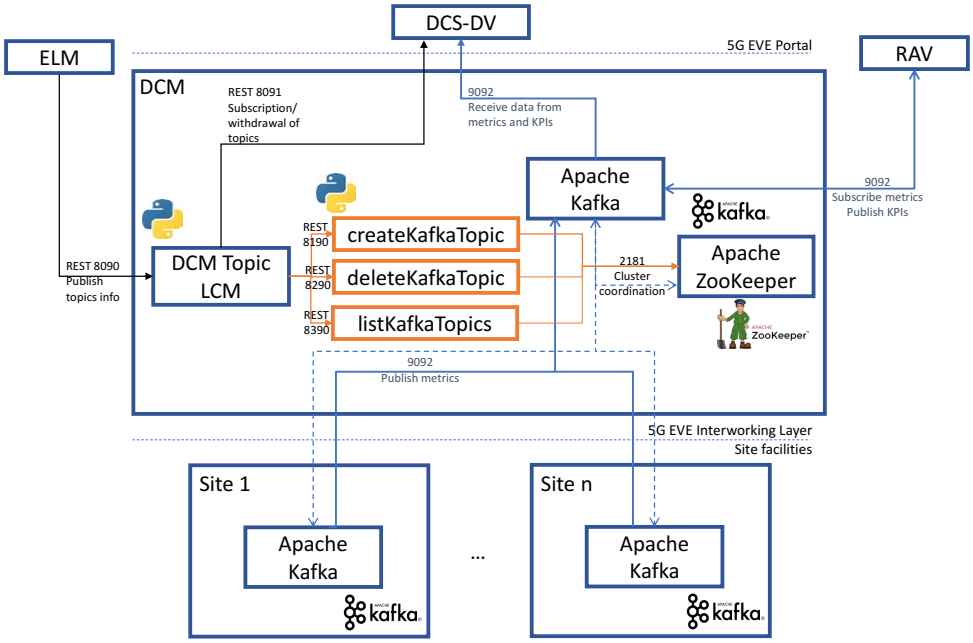

Figure 57: Data Collection Manager microservices architecture.

For the **Data Collection and Storage-Data Visualization** component, however, only **one REST-based module** has been **identified**: the ***kafkaConsumer*** module, which **creates** a ***Python*-based *Kafka* consumer** listening to the topic created in the platform, and whose function is to **trigger** the **creation** of the corresponding **Kibana dashboard** when the **first message** is **received** in the **topic**, so that the dashboard is only created when there is data available in the topic.

Apart from this, the ***Python* logic** has also been transformed into a **simpler component**, identified as ***DCS Topic LCM***, and ***Logstash*** counts now with a specific **module**, called **Pipeline manager**, which **manages** the **creation and deletion** of ***Logstash* pipelines**, function that was executed by the DCS *Python* logic in the legacy architecture. All these **changes and updates** are reflected in Figure 58:

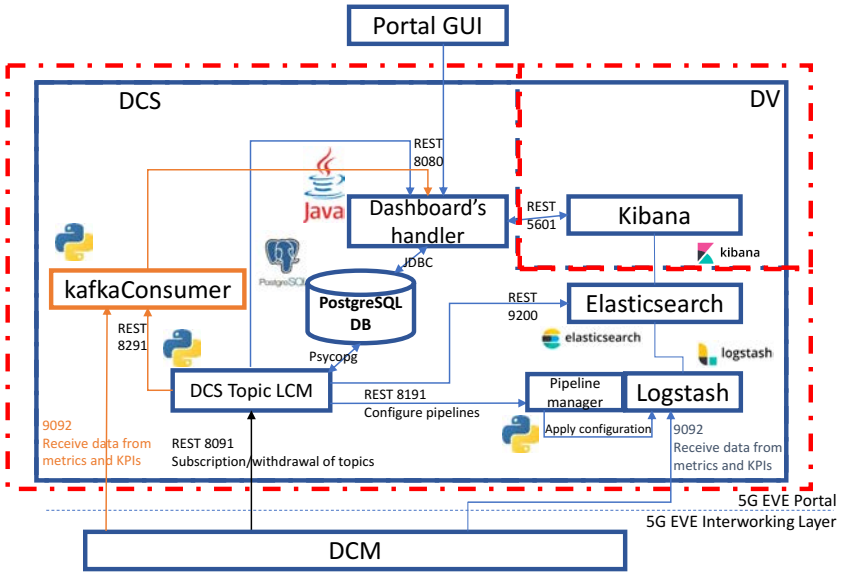

Figure 58: Data Collection and Storage-Data Visualization microservices architecture.

Now that the Monitoring microservices architecture has been presented, it is worth to review **how** the **monitoring workflow changes** to fit in this new **approach**, distinguishing between the **topic creation** and **topic deletion** workflow, which are the **two main operations** that involves the **Monitoring platform** in terms of functions' **automation**.

The first case, related to the **topic creation workflow**, is fully described in Figure 59:

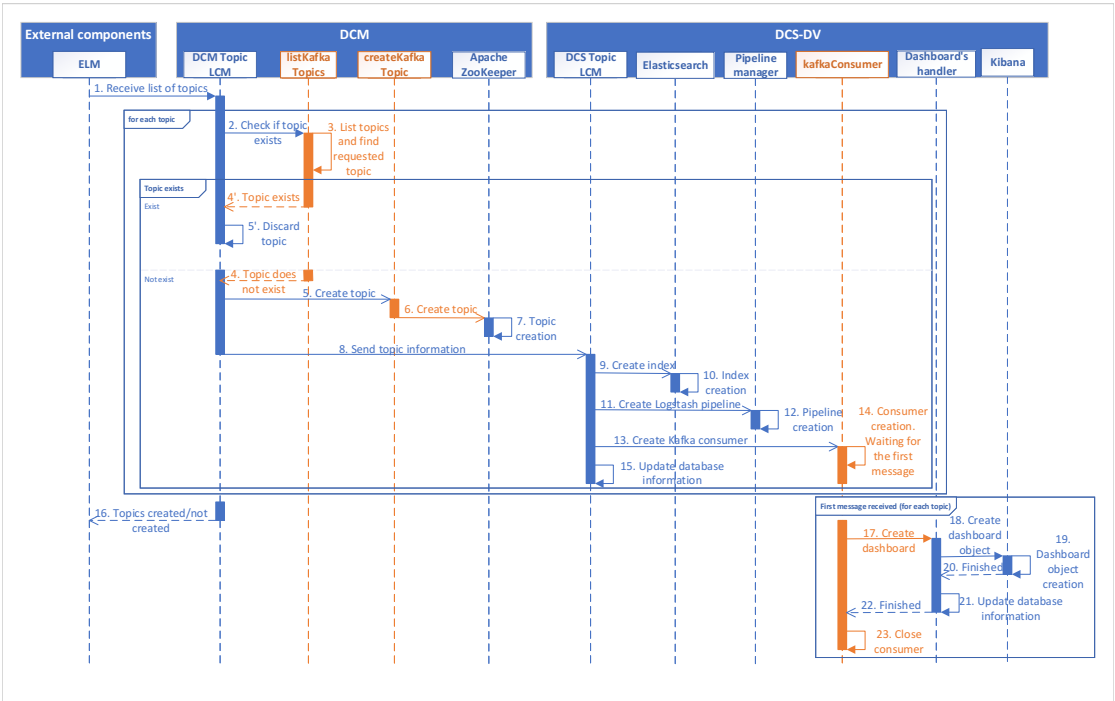

Figure 59: Topic creation workflow for the Monitoring microservices architecture.

The **full description** of the **messages** is the following:

1. The **topics** to be created in the platform are received in the **DCM** from the **ELM** component. This **data** is **handled** by the ***DCM Topic LCM***.

2. The ***DCM Topic LCM*** **check**, for **each topic**, if the topic **already exists** in the platform, checking if it needs to be created or not. This checking is requested to the ***listKafkaTopics*** **module**.

3. The ***listKafkaTopics*** module **list** the **topics present** in the system and **checks** if the requested **topic** is **contained** in that list. It may happen that the topic **already exists**, so that it **replies back** to the ***DCM Topic LCM*** (message 4'), and this component automatically **discards** the topic because it is **already present** in the platform (message 5').

4. The typical case is that the **topic does not exist**, message that is sent from ***listKafkaTopics*** to the ***DCM Topic LCM***.

5. As the topic is not present in the system, the ***DCM Topic LCM*** requests to the ***createKafkaTopic*** module its **creation**.

6. ***createKafkaTopic*** invokes the specific **commands** to **create the topic** in ***Apache Kafka***.

7. **ZooKeeper** receives the instructions sent by ***createKafkaTopic*** and then **creates the topic**.

8. In parallel, the ***DCM Topic LCM*** send the **information** related to the **topic** to the **DCS-DV**, specifically to the ***DCS Topic LCM*** component.

9. The first operation triggered by the ***DCS Topic LCM*** is the **creation** of the related ***Elasticsearch*** **index**, to **filter** the **information** received in ***Logstash*** and to **offer** it to ***Kibana***.

10. ***Elasticsearch*** receives the request and **creates the index**.

11. After this, the ***DCS Topic LCM*** requests the **creation** of the corresponding ***Logstash*** **pipeline**, which enables the process in *Logstash* that will **listen** to the corresponding *Kafka* **topic**, will **pre-process** the **data** received and will **serve** it to the proper ***Elasticsearch*** **index** already configured. This request is sent to the **Pipeline manager**.

12. The **Pipeline manager** present in the *Logstash* component **applies** the **pipeline configuration**, **creating it** as a result.

13. Finally, the ***DCS Topic LCM*** requests the **creation** of the ***Kafka*** **consumer** that will be **listening** to the **topic** until the **first message arrives**, so that the ***Kibana*** **dashboard** is **created**. This is requested to the ***kafkaConsumer*** module.

14. This module **creates** the **consumer** and **waits** for the **reception** of the **first message** in the **topic**.

15. The ***DCS Topic LCM*** **saves** some **internal information** on **database** to maintain the **system status**.

16. After **processing all the topics** received by the ELM, the ***DCM Topic LCM*** informs to the **ELM** the **status** of **each topic** requested (*i.e.* if each of them has been created or not).

17. In parallel, when the **first message** is **received** for a given **topic**, the ***kafkaConsumer*** module triggers the **creation** of the corresponding ***Kibana*** **dashboard**, sending the request to the **Dashboard's handler** component.

18. This handler requests the **creation** of the **dashboard's object** to ***Kibana***.

19. ***Kibana*** **creates** the **object** and **replies back** with the **information** needed to **build** the **URL** to be served to **external components**.

20. ***Kibana*** indicates that it has **finished** the **creation** of the **dashboard's object**.

21. The **Dashboard's handler saves** this data into the **database**.

22. Finally, the **handler** indicates to the ***kafkaConsumer*** that it has **finished** the **creation** of the **dashboard**.

23. Consequently, the ***kafkaConsumer*** **closes** the ***Kafka*** **consumer** created before.

At this point, the system would be ready to **process** the **monitoring data** received in the platform and would **serve** the **values** received with the proper ***Kibana*** **dashboards**.

In the case of the **topic deletion workflow**, triggered when a given **experiment** is **finished** and starts to be **decommissioned** from the **platform**, it is presented in Figure 60:

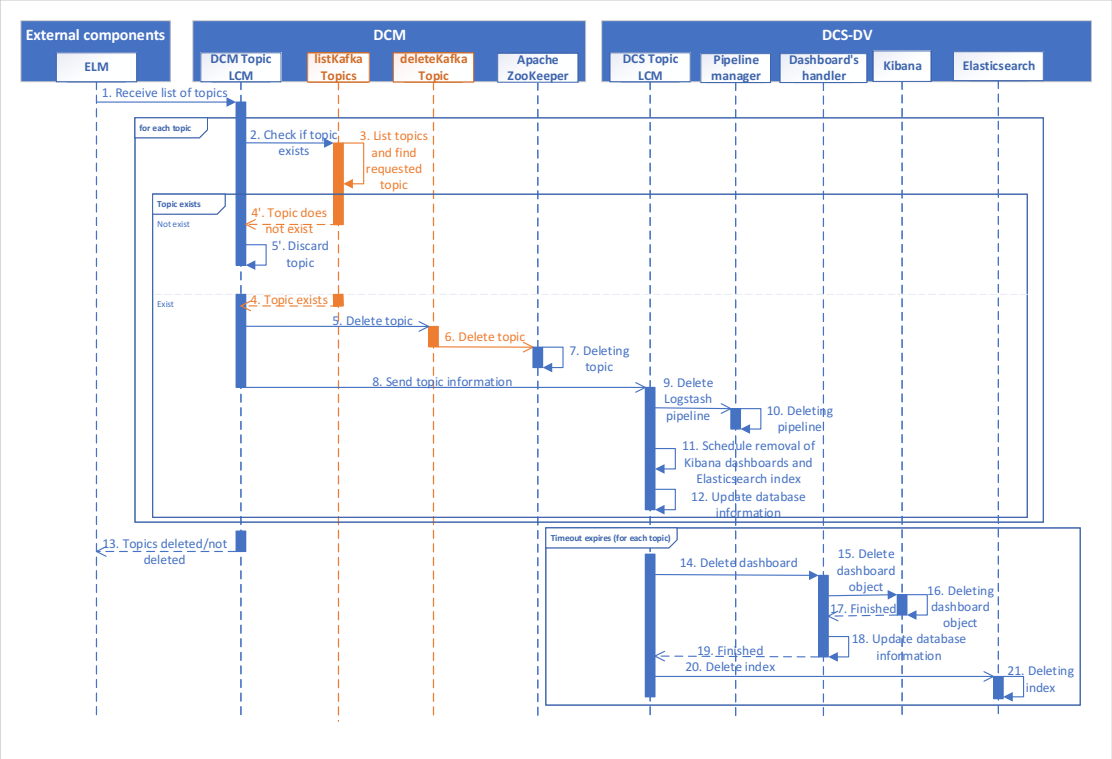

Figure 60: Topic deletion workflow for the Monitoring microservices architecture.

The **full description** of the **messages** is the following:

1. The **topics** to be **deleted** in the platform are received in the **DCM** from the **ELM** component. This **data** is **handled** by the ***DCM Topic LCM***.

2. The ***DCM Topic LCM*** **check**, for **each topic**, if the topic **already exists** in the platform, checking if it needs to be deleted or not. This checking is requested to the ***listKafkaTopics*** module.

3. The ***listKafkaTopics*** module **list** the **topics** present in the system and **checks** if the requested **topic** is **contained** in that list. It may happen that the topic **does not exist**, so that it **replies back** to the ***DCM Topic LCM*** (message 4'), and this component automatically **discards** the topic because it is not present in the platform (message 5').

4. The typical case is that the **topic exists**, message that is sent from ***listKafkaTopics*** to the ***DCM Topic LCM***.

5. As the topic is **present** in the system, the ***DCM Topic LCM*** requests to the ***deleteKafkaTopic*** module its **deletion**.

6. ***deleteKafkaTopic*** invokes the specific **commands** to **delete** the **topic** in ***Apache Kafka***.

7. ***ZooKeeper*** receives the **instructions** sent by ***deleteKafkaTopic*** and then **deletes the topic**.

8. In parallel, the ***DCM Topic LCM*** send the **information** related to the **topic** to the **DCS-DV**, specifically to the ***DCS Topic LCM*** component.

9. Firstly, the ***DCS Topic LCM*** requests the **deletion** of the ***Logstash*** **pipeline** to the **Pipeline manager**.

10. The **Pipeline manager** present in the *Logstash* component **removes** the ***Logstash*** **pipeline** requested.

11. As the **monitoring data** has a **retention time**, it cannot be deleted directly. For this reason, the **removal** of the **data** present in the **platform** (*i.e.* the data saved in the *Elasticsearch* index and the *Kibana* dashboard that presents it through the GUI) is **scheduled** for after the retention time.

12. The ***DCS Topic LCM*** **saves** some **internal information** on **database** to maintain the **system status**.

13. After **processing all the topics** received by the ELM, the ***DCM Topic LCM*** informs to the **ELM** the **status of each topic** requested (*i.e.* if each of them has been deleted or not).

14. In parallel, when the **timeout expires** (*i.e.* the retention time has expired), the ***DCS Topic LCM*** requests the **deletion** of the ***Kibana*** **dashboard** to the **Dashboard's handler**.

15. This **handler** requests the **deletion** of the **dashboard's object** to ***Kibana***.

16. ***Kibana*** **deletes** the **object**.

17. ***Kibana*** indicates that it has **finished** the **deletion** of the **dashboard's object**.

18. The **Dashboard's handler removes** this **data** into the **database**.

19. Finally, the **handler** indicates to the ***DCS Topic LCM*** that it has **finished** the deletion of the dashboard.

20. Then, the ***DCS Topic LCM*** requests the **deletion** of the ***Elasticsearch*** **index** to ***Elasticsearch***.

21. ***Elasticsearch*** manages to **remove** the **index**.

7.2.3. Transformation from Microservices to Serverless Architecture

Taking into consideration the building of the **microservices architecture** of the Monitoring platform already presented in Section 7.2.2, the **second stage** consist on taking the **REST-based services** identified, together with the **two Topic LCM** present in the architecture, and **modeling them** as **serverless functions** managed by *OpenFaaS* [121], a platform already presented in the state of the art in Section 2.3. This **transformation** can be seen in Figure 61, marking the **serverless part** in **orange**:

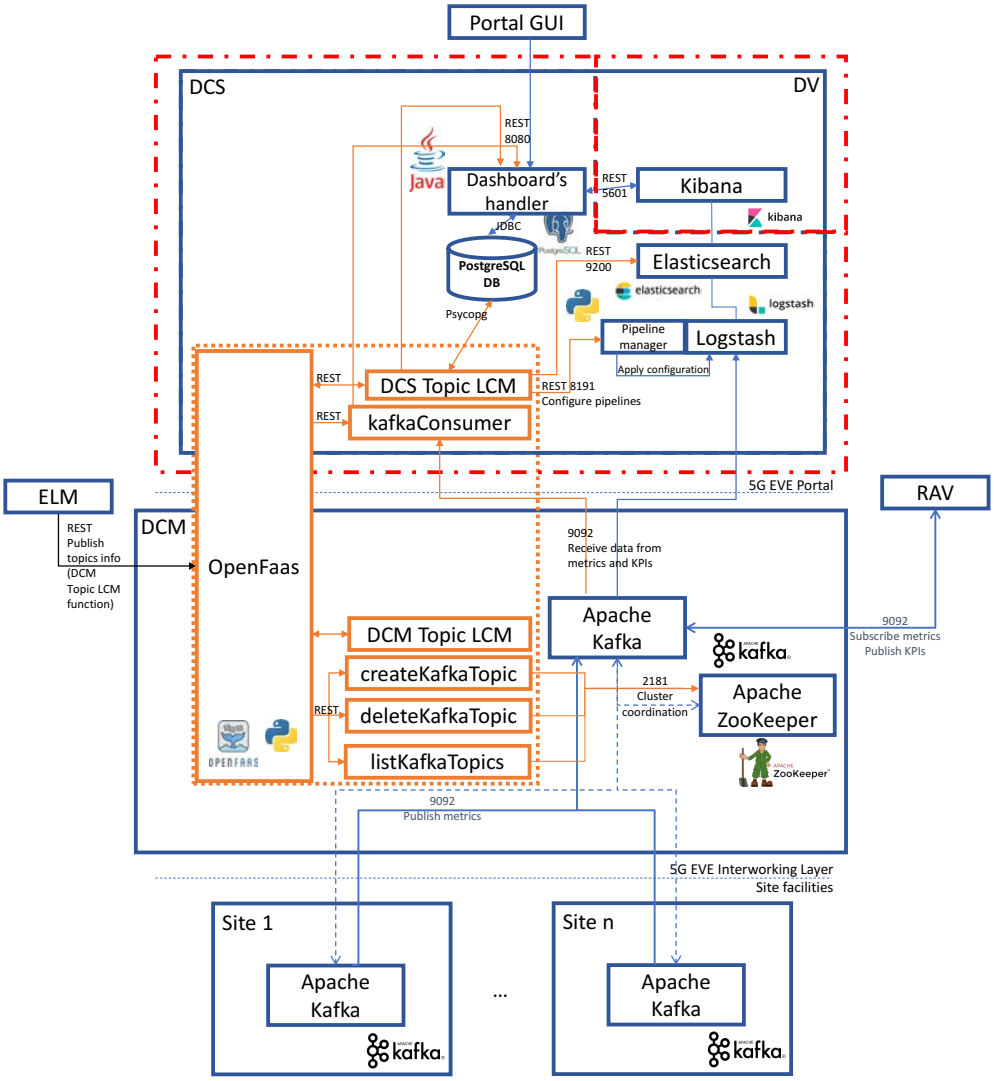

Figure 61: Serverless architecture of the Monitoring platform.

Consequently, the **workflows change** considerably, as the ***OpenFaaS*** platform becomes the **central component** of the serverless implementation, exposing the **interfaces** needed to **reach each serverless function**. So now, when a serverless function has to be invoked, the entity that requests the service needs to contact *OpenFaaS*, which will be in charge of **managing** the **lifecycle** of the **serverless function**, thus **instantiating** the **resources needed** to **execute** that **function**, and then **releasing** them when it **finishes** its **execution**.

This will be checked with the **topic creation** and **deletion workflows**. Starting with the **first one**, it is presented in Figure 62:

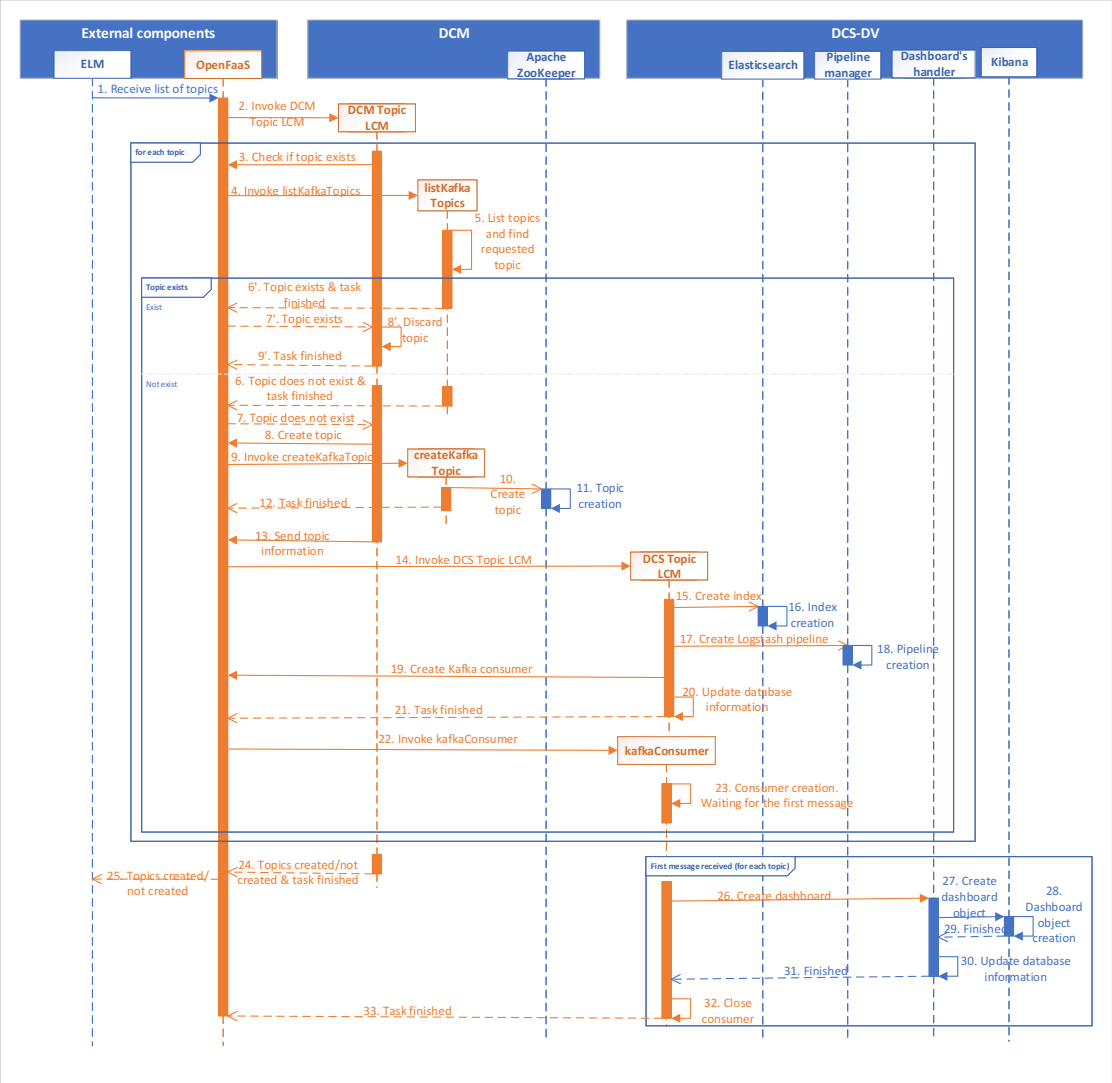

Figure 62: Topic creation workflow for the Monitoring serverless architecture.

The **full description** of the **messages** is the following:

1. The **topics** to be **created** in the platform are received by ***OpenFaaS*** from the **ELM** component.

2. ***OpenFaaS*** invokes the ***DCM Topic LCM*** to handle this request.

3. The ***DCM Topic LCM*** **check**, for **each topic**, if the topic **already exists** in the platform, checking if it needs to be created or not. This checking is requested to ***OpenFaaS***.

4. ***OpenFaaS*** invokes ***listKafkaTopics*** to handle this request.

5. The ***listKafkaTopics*** module **list the topics** present in the system and **checks** if the requested topic is **contained** in that list. It may happen that the **topic already exists**, so that it **replies back to *OpenFaaS*** (message 6'), also indicating that it has **finished** its execution, so ***OpenFaaS*** **decommissions it**. Then, it forwards the **response** to the ***DCM Topic***

LCM (message 7'), and this component automatically **discards the topic** because it is already present in the platform (message 8'), also informing to ***OpenFaaS*** that it has **finished** its execution (message 9').

6. The **typical case** is that the **topic does not exist**, message that is sent from ***listKafkaTopics*** to ***OpenFaaS***. It also indicates that it has **finished** its execution, so ***OpenFaaS*** **decomissions it**.

7. ***OpenFaaS*** forwards the response to the ***DCM Topic LCM***.

8. As the topic is **not present** in the system, the ***DCM Topic LCM*** requests to ***OpenFaaS*** its **creation**.

9. ***OpenFaaS*** invokes ***createKafkaTopic*** to handle this request.

10. ***createKafkaTopic*** invokes the specific **commands** to **create the topic** in ***Apache Kafka***.

11. ***ZooKeeper*** receives the instructions sent by ***createKafkaTopic*** and then **creates the topic**.

12. ***createKafkaTopic*** indicates to ***OpenFaaS*** that it has **finished** its execution, so ***OpenFaaS*** **decomissions it**.

13. In parallel, the ***DCM Topic LCM*** send the **information** related to the **topic** to the ***OpenFaaS***, to be received in the **DCS-DV**.

14. ***OpenFaaS*** invokes the ***DCS Topic LCM*** to handle this request.

15. The first operation triggered by the ***DCS Topic LCM*** is the **creation** of the related ***Elasticsearch*** **index**.

16. ***Elasticsearch*** receives the request and **creates the index**.

17. After this, the ***DCS Topic LCM*** requests the **creation** of the corresponding ***Logstash*** **pipeline**. This request is sent to the **Pipeline manager**.

18. The **Pipeline manager** present in the *Logstash* component **applies** the **pipeline configuration**, **creating it** as a result.

19. Finally, the ***DCS Topic LCM*** requests the **creation** of the ***Kafka*** **consumer** that will be **listening** to the **topic** until the **first message arrives**, so that the ***Kibana*** **dashboard is created**. This is requested to ***OpenFaaS***.

20. In parallel, the ***DCS Topic LCM*** **saves** some internal **information** on **database** to maintain the **system status**.

21. The ***DCS Topic LCM*** indicates to ***OpenFaaS*** that it has **finished its execution**, so ***OpenFaaS*** **decomissions** it.

22. After receiving message 19, ***OpenFaaS*** invokes ***kafkaConsumer*** to handle this request.

23. This module **creates the consumer** and **waits** for the **reception** of the **first message** in the **topic**. As it has not finished its execution, it does not contact *OpenFaaS* for being decommissioned.

24. After **processing all the topics** received by the ELM, the ***DCM Topic LCM*** informs to ***OpenFaaS*** the **status of each topic** requested (*i.e.* if each of them has been created or not). It also indicates to ***OpenFaaS*** that it has **finished** its execution, so ***OpenFaaS* decomissions it**.

25. ***OpenFaaS*** forwards the response to the **ELM**.

26. In parallel, when the **first message is received** for a given **topic**, the ***kafkaConsumer*** module triggers the **creation** of the corresponding ***Kibana* dashboard**, sending the request to the **Dashboard's handler** component.

27. This handler requests the **creation** of the **dashboard's object** to ***Kibana***.

28. ***Kibana* creates the object** and **replies back** with the **information** needed to **build the URL** to be served to **external components**.

29. ***Kibana*** indicates that it has **finished** the **creation of the dashboard's object**.

30. The **Dashboard's handler saves** this **data** into the **database**.

31. Finally, the handler indicates to the ***kafkaConsumer*** that it has **finished** the **creation of the dashboard**.

32. Consequently, the ***kafkaConsumer* closes** the ***Kafka* consumer** created before.

33. ***kafkaConsumer*** indicates to ***OpenFaaS*** that it has **finished** its execution, so ***OpenFaaS* decomissions it**.

In the same way, the **updated topic deletion workflow** can be seen in Figure 63:

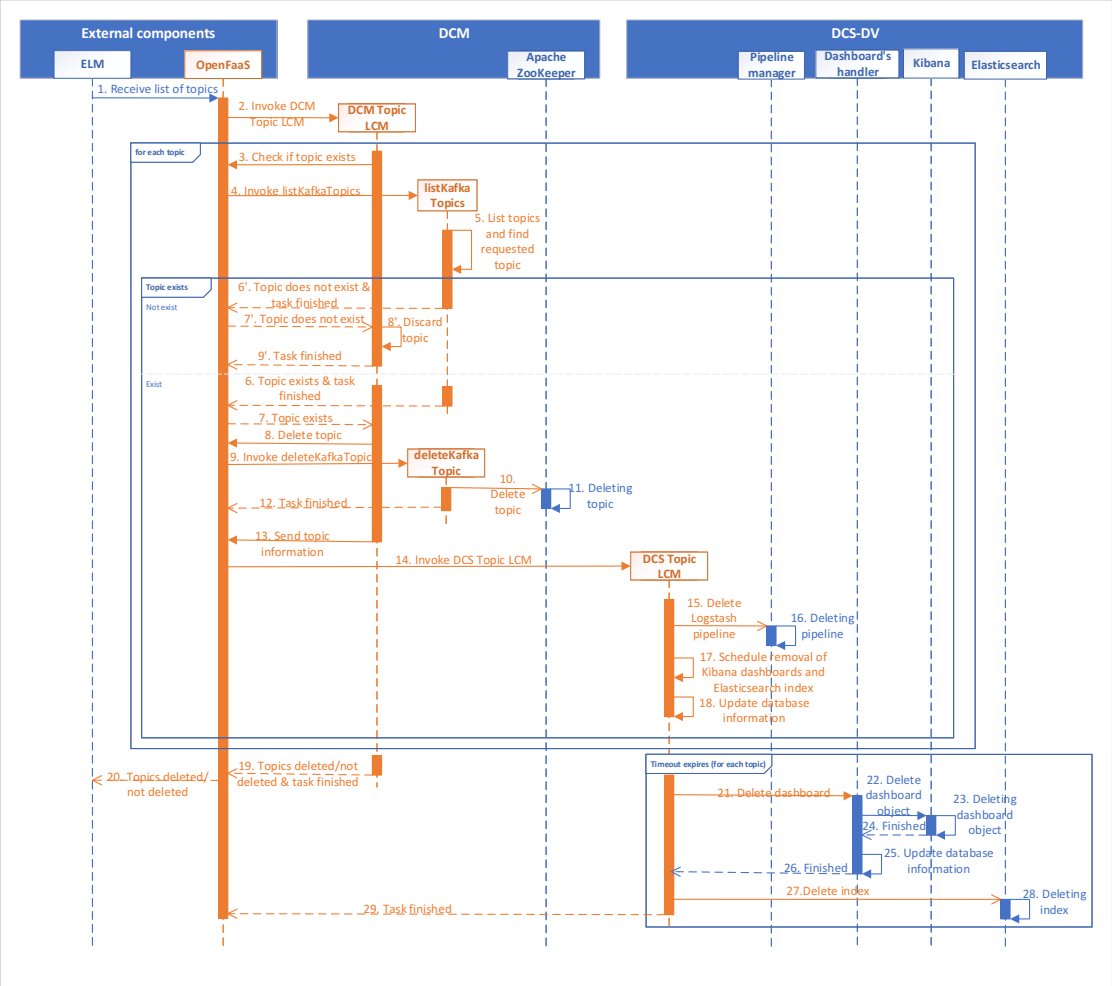

Figure 63: Topic deletion workflow for the Monitoring serverless architecture.

The **full description** of the **messages** is the following:

1. The **topics** to be **deleted** in the platform are received by ***OpenFaaS*** from the **ELM** component.

2. ***OpenFaaS*** invokes the ***DCM Topic LCM*** to handle this request.

3. The ***DCM Topic LCM*** check, for **each topic**, if the **topic already exists** in the platform, checking if it needs to be deleted or not. This checking is requested to ***OpenFaaS***.

4. ***OpenFaaS*** invokes ***listKafkaTopics*** to handle this request.

5. The *listKafkaTopics* module **list the topics** present in the system and checks if the **requested topic** is **contained** in that **list**. It may happen that the topic **does not exist**, so that it **replies back to *OpenFaaS*** (message 6'), also indicating that it has **finished** its execution, so ***OpenFaaS* decommissions it**. Then, it forwards the response to the ***DCM Topic LCM*** (message 7'), and this component automatically **discards the topic** because it is not present in the platform (message 8'), also informing to ***OpenFaaS*** that it has **finished** its execution (message 9').

6. The typical case is that the **topic exists**, message that is sent from ***listKafkaTopics*** to ***OpenFaaS***. It also indicates that it has **finished** its execution, so ***OpenFaaS*** **decomissions it**.

7. ***OpenFaaS*** forwards the response to the ***DCM Topic LCM***.

8. As the **topic is present** in the system, the ***DCM Topic LCM*** requests to ***OpenFaaS*** its **deletion**.

9. ***OpenFaaS*** invokes ***deleteKafkaTopic*** to handle this request.

10. ***deleteKafkaTopic*** invokes the specific **commands** to **delete the topic** in ***Apache Kafka***.

11. ***ZooKeeper*** receives the **instructions** sent by ***deleteKafkaTopic*** and then **deletes the topic**.

12. ***deleteKafkaTopic*** indicates to ***OpenFaaS*** that it has **finished** its execution, so ***OpenFaaS*** **decomissions it**.

13. In parallel, the ***DCM Topic LCM*** send the **information** related to the **topic** to the ***OpenFaaS***, to be received in the **DCS-DV**.

14. ***OpenFaaS*** invokes the ***DCS Topic LCM*** to handle this request.

15. Firstly, the ***DCS Topic LCM*** requests the **deletion** of the ***Logstash*** **pipeline** to the **Pipeline manager**.

16. The **Pipeline manager** present in the ***Logstash*** component **removes** the ***Logstash*** **pipeline** requested.

17. As the **monitoring data** has a **retention time**, it cannot be deleted directly. For this reason, the **removal** of the **data** present in the **platform** (*i.e.* the data saved in the *Elasticsearch* index and the *Kibana* dashboard that presents it through the GUI) is **scheduled** for **after the retention time**.

18. The ***DCS Topic LCM*** **saves** some **internal information** on **database** to maintain the **system status**.

19. After **processing all the topics** received by the **ELM**, the ***DCM Topic LCM*** informs to ***OpenFaaS*** the **status of each topic** requested (*i.e.* if each of them has been created or not). It also indicates to ***OpenFaaS*** that it has **finished** its execution, so ***OpenFaaS*** **decomissions it**.

20. ***OpenFaaS*** forwards the response to the **ELM**.

21. In parallel, when the timeout expires (*i.e.* the retention time has also expired), the ***DCS Topic LCM*** requests the **deletion** of the ***Kibana*** **dashboard** to the **Dashboard's handler**.

22. This **handler** requests the **deletion** of the **dashboard's object** to ***Kibana***.

23. ***Kibana*** **deletes** the **object**.

24. ***Kibana*** indicates that it has **finished** the **deletion** of the **dashboard's object**.

25. The **Dashboard's handler removes** this **data** into the **database**.

26. Finally, the **handler** indicates to the ***DCS Topic LCM*** that it has **finished** the **deletion** of the **dashboard**.

27. Then, the ***DCS Topic LCM*** requests the **deletion** of the ***Elasticsearch*** **index** to ***Elasticsearch***.

28. ***Elasticsearch*** manages to **remove the index**.

29. The ***DCS Topic LCM*** indicates to ***OpenFaaS*** that it has **finished** its execution, so ***OpenFaaS*** **decomissions it**.

7.3. Workflow's Validation

To **validate** the **serverless architecture** and the **workflows** presented in Section 7.2.3, a specific **testbed** has been built for this purpose. It consists of an **Ubuntu Server 16.04 LTS virtual machine** [90], with 12 vCPU and 12 GB of RAM, deployed in a server virtualized with *Proxmox* [89], which is equipped with 40 Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20 GHz and 128 GB RAM. For **deploying** the **Monitoring platform**, ***K3s*** [103] has been used to **orchestrate the containerized components**[15], integrating ***OpenFaaS*** for the **deployment** of the **serverless functions**.

The **components deployed** in this testbed can be checked in Figure 64:

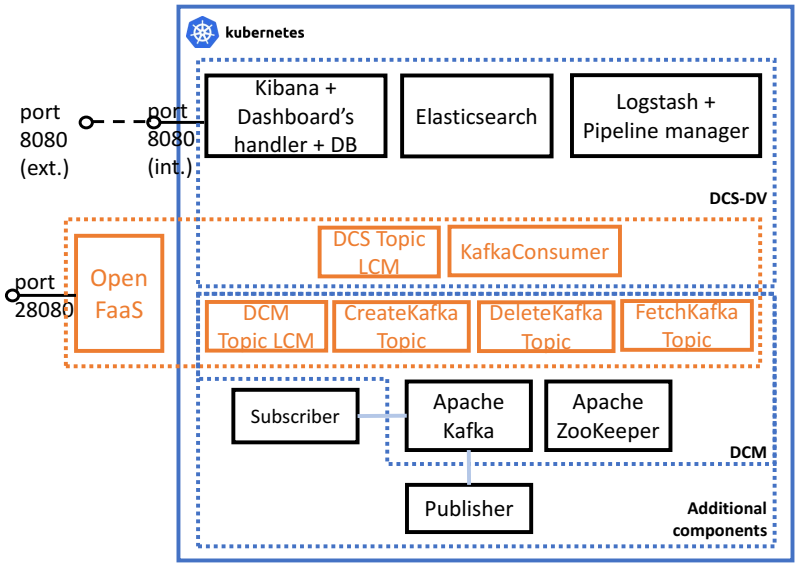

Figure 64: Testbed for validating the Monitoring serverless architecture.

[15]The images of these components can be found in [104].

As it can be seen, the **topic creation** and **deletion workflows** are **triggered** by **sending** the corresponding **REST request** to ***OpenFaaS***, that triggers the **creation** of the ***DCM Topic LCM*** and, then, the **consecutive operations** already described in Section 7.2.3.

Apart from this, the **testbed** also counts with a ***Kafka* publisher** to provide **data** to the **topics** created, managed by *Apache Kafka* and *ZooKeeper* instances. A ***Kafka* subscriber** is also present for **debugging purposes**. On top of all, the ***Elastic Stack*** is built, also including **additional features** such as the **Pipeline manager** (directly implemented in the *Logstash* container) and the **Dashboard's handler** (included in the *Kibana* container, together with the database instance).

7.4. Summary

Summing up the contributions provided in this Chapter, the **serverless paradigm** has been **introduced** within the field of **5G networks**, ranging from **general information** about the **concept, advantages and challenges** related to this new technology, to its **integration** into a **well-known 5G-related platform**, such as the **Monitoring platform** that is object under study in this thesis.

In this platform, a **two-stage process** was defined to properly define the **serverless approach**, starting with a **first transformation** to a **microservices architecture**, then evolving it to achieve the **serverless architecture**. In this process, it was also targeted the **challenges** related to the serverless paradigm already showed in Section 7.1.3, as **new VNFs** (*i.e.* new **serverless functions**) has been created for this purposes, being **managed and orchestrated** by a serverless-related component such as ***OpenFaaS***.

As a result, the system becomes **more efficient** in terms of **resource consumption**, as the **serverless functions** only **use compute resources** when they are **executed**, and it also implies **saving money** if the **infrastructure** is running in a **Cloud provider** that applies a pay-per-use model. However, it has the **drawback** of counting with a **more complicated workflow** to achieve this, probably resulting in a **higher execution time** of the overall workflow.

Finally, the **workflows** have been **validated** in a specific **testbed**. Note that some **performance tests** are **missed** in this Chapter, but this is because this topic will be **further analyzed** in the **next Chapter 8**.

8

Evaluation of the Monitoring Platform Deployment Based on Different Virtualization Techniques

In **Chapter 7**, the **serverless** paradigm was **introduced** in the context of **5G networks**, remarking its **characteristics** and **main advantages**. This new trend in the field of virtualization techniques allows to **adjust** the **amount of resources** allocated for a given **workload** with a **finer level of granularity**, achieving a **better performance profile** by using **functions** that are **triggered only when needed**.

Nevertheless, some **aspects** related to the **real performance** of the **serverless architectures** are still **open**, such as the **comparison** with other **virtualization techniques**, in order to confirm whether the **serverless** approach is really **more efficient** in terms of **resource consumpion**, or also the **verification** that **not only serverless functions**, but also **standard components** executed by using **serverless technologies**, achieve a **better performance profile**.

With these topics in mind, this Chapter presents a **full performance evaluation** of the **Monitoring platform**, with the main objetive of **comparing different virtualization techniques between them** in order to **position** the **serverless paradigm** in the **state of the art**. For this purpose, the **same procedure** followed in **Chapters 3 and 4** will be **applied** for doing these **tests**, *i.e.* building a specific **testbed** for each **virtualization technique** and **executing** a set of **experiments** in the **Monitoring platform**, then extracting useful **performance parameters** such as the **batch write latency** or the **I/O message rate**, already explained in these previous Chapters.

Being more precise, the **virtualization techniques under study** are the following: (1) the case of not using virtualization, so **physical servers** are directly used, (2) virtualization based on ***KVM***, (3) containerization with ***Docker***, (4) automatic deployment of containers with ***Kubernetes***, and (5) the usage of ***Kata Containers*** with ***Firecracker*** **or** ***QEMU*** as hypervisors, being *Kata* and

Firecracker two of the **serverless technologies** already presented in the state of the art, in Section 2.3.

To to this, the following **structure** of sections is proposed:

- First of all, Section 8.1 specifies the **testbed** used for the **set of tests proposed** in this Chapter, describing the **servers** used and also the **deployments for each virtualization technique** under study.
- Then, Section 8.2 presents the **test cases** to be performed, differentiating between a **single server deployment** and a scenario in which **horizontal scaling techniques** are **applied** to check if the **results**, in terms of performance parameters, can be **improved**.
- Section 8.3 details the **first set of test cases** related to the **single server performance evaluation**, evaluating the **CPU consumption, batch write latency and I/O message rate** on **each scenario** proposed.
- On the other hand, Section 8.4 introduces the **horizontal scaling performance evaluation** for the case of the ***Kubernetes* scenario**, **comparing** the **results** obtained with the **single server case**.
- Finally, Section 8.5 **summarizes and concludes**.

8.1. Testbed Setup

8.1.1. Servers' Description

For this evaluation process, **two servers** located in the **University of Perugia**, based on *Ubuntu Server 20.04* [90], have been used. Their **hardware specifications** are fully described in Table 2:

Table 2: Specification of the servers used in the testbed.

Server name	**Tardis**	**Saul**
CPU	Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40 GHz	Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30 GHz
RAM	128 GB @ 2133 MT/s	64 GB @ 2133 MT/s
Disk	280 GB (145 MB/s write speed)	280 GB (145 MB/s write speed)
Network interfaces	1x10 Gbps, 4x1 Gbps	1x10 Gbps, 4x1 Gbps

Each server has the following **utilities** installed:

- **Tardis:**
  - ***KVM*** [122] for **server's virtualization**, allowing to deploy virtual machines on the physical server.

- ***Docker*** [92] for **container virtualization**, using ***runc*** [123] as **runtime**.
- ***MicroK8s*** [124], a ***Kubernetes* minimal production version**, using ***containerd*** [125] as **runtime**.
- ***Kata Containers*** [126] as **container runtime** for the evaluation of **serverless tools**, used in combination with ***Docker*** and also using either ***Firecracker*** [127] or ***QEMU*** [128] as **hypervisors**.
- The image of the **Data Collection Manager** component from the **Monitoring platform**, using [78] for the deployment in physical and virtualized environments and the images provided in [104] for the containerized environments (*i.e. Docker*, *Kubernetes* and *Kata*). The main subcomponents used from the DCM for the tests were ***Kafka* and *ZooKeeper***.
- A **CPU collector script** based on the ***mpstat* command**.

▪ **Saul:**

- A ***Kafka* publisher** based on ***Sangrenel*** [91], for obtaining the **performance metrics under study** (*i.e.* batch write latency and I/O message rate).

Note that the **time** in both servers was **synchronized** by using the **NTP protocol**. And finally, although the **CPU model** of each server is **slightly different**, they do **not** present a **big difference** in terms of **performance**, according to the study performed on [129].

8.1.2. Testbed Specification for each Virtualization Technique

Despite having the **same distribution of tools and components** on each server for **all the deployments** evaluated in this study, there are some **particularities** on each of them that must be mentioned, at least related to the **configuration applied on Tardis**, as Saul remains in the same status for all the testbeds evaluated (it only contains the *Kafka* publisher script, which is the same for all cases).

Before starting with the **description of the testbeds**, note that **no limits** will be imposed to the **allocation of hardware resources** for the **tools used** during the tests. This means that the **Monitoring platform** can make use of **all the resources available** on the servers **without constraints**.

Having said this, the **base scenario** would the **physical testbed**, using directly both **Tardis and Saul servers without any virtualization technique**. In this case, ***Kafka* and *ZooKeeper*** are **directly installed** on **Tardis** and configured as **Linux services**. This configuration can be seen in Figure 65:

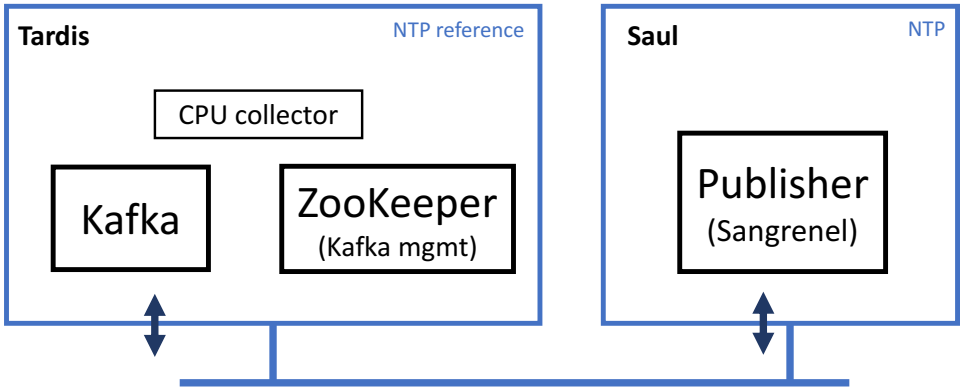

Figure 65: Physical testbed for the evaluation of the Monitoring platform.

The **second scenario** is the **virtual testbed**, presented in Figure 66, using ***KVM*** in **Tardis** to deploy ***Kafka*** **and** ***ZooKeeper*** in a specific **Ubuntu virtual machine**[16]. This **VM** is **synchronized** against **Tardis** to properly configure the **time with NTP**. Moreover, for **enabling** the **connectivity between the VM and Saul**, some ***iptables*** **and** ***UFW*** **rules** based on the following guide [130] were also applied on **Tardis**.

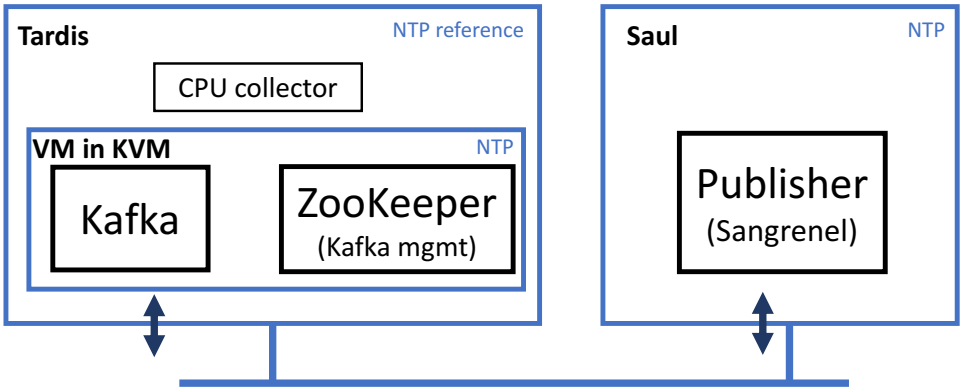

Figure 66: Virtual testbed for the evaluation of the Monitoring platform.

The **third scenario** is the ***Docker* testbed**, using ***runc*** as **container runtime** (the default one), in which a ***Kafka*** **and a** ***ZooKeeper*** **container** are deployed in **Tardis**, **exposing** the required **ports** to be **accessed from Saul**. This testbed is depicted in Figure 67:

[16]The **disk** used for the VM, based on the **qcow2 technology**, was configured with the **writeback mode** for the cache, and also using the **metadata property** for the **preallocation** parameter.

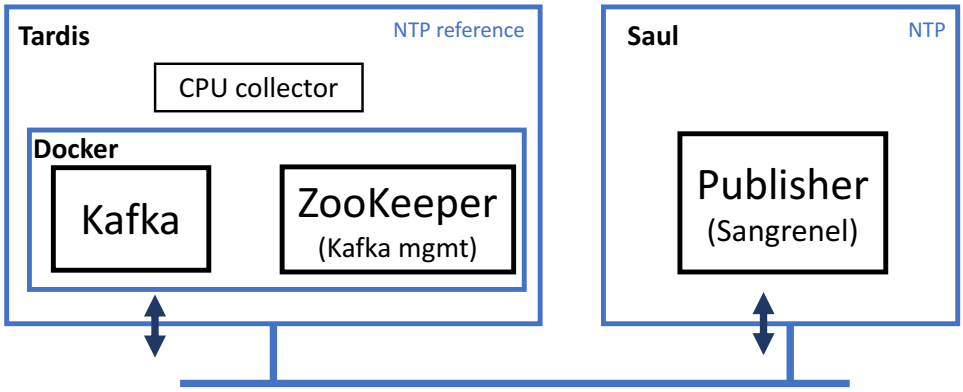

Figure 67: *Docker* (*runc*) testbed for the evaluation of the Monitoring platform.

The **fourth scenario** is the ***Kubernetes* testbed**, using ***containerd*** as **runtime** (also the default one), deploying a ***Kafka* and a *ZooKeeper* pod** in **Tardis**, and also **exposing** the **required ports** for the **access from Saul**. The testbed is reflected in Figure 68:

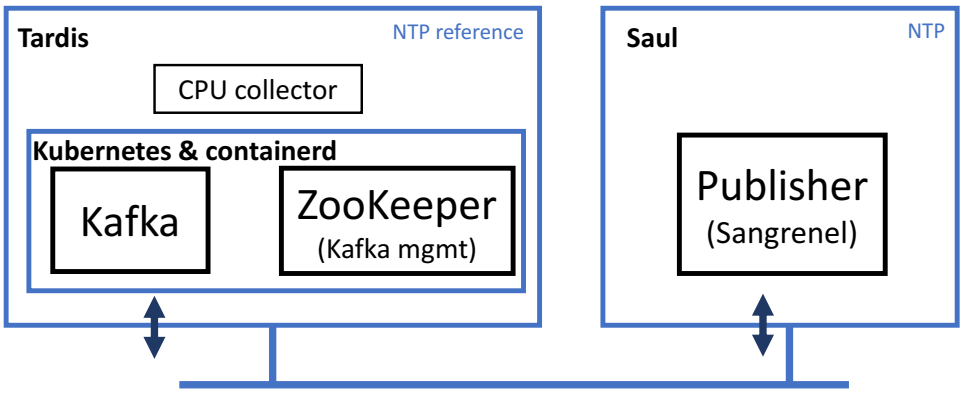

Figure 68: *Kubernetes* (*containerd*) testbed for the evaluation of the Monitoring platform.

And finally, the **fifth scenario** is the ***Kata* testbed**, showed in Figure 69, which is similar to the ***Docker* testbed**, but using a **customized runtime** that can be either ***Kata+Firecracker*** or ***Kata+QEMU***.

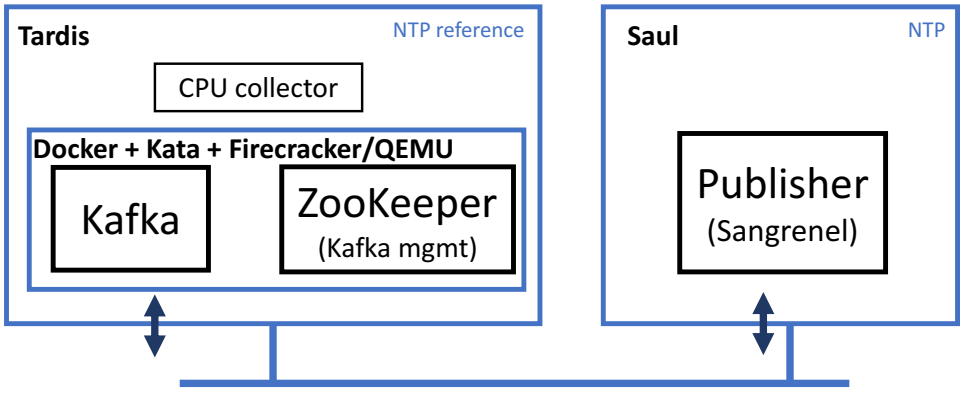

Figure 69: *Kata (Firecracker/QEMU)* testbed for the evaluation of the Monitoring platform.

8.2. Test Cases' Description

In general terms, the **same assumptions** commented on **Section 3.3.1** also apply to this **evaluation process**, having a **maximum number** of **six experiments** running **simultaneously** on the **Monitoring platform**, considering that **one experiment** implies the **creation of 20 topics** in the system, with a **concurrent publication rate** of approximately **102,4** *Mbps*.

Delving into the **particular tests** to be executed on each testbed, they consist on the **execution** of an **experiment** in the **Monitoring platform**, **publishing data** in ***Kafka*** at a given **data rate** (depending on the number of topics present in the platform) with *Sangrenel*. **Two types of tests** are carried out:

- **Single server tests** (reported in Section 8.3): this set of tests implies the **execution of tests** with a **single instance** of ***Apache Kafka*** running in the Monitoring platform. This is the configuration that have been used in all the tests reported in this memory until now.
- **Horizontal scaling tests** (reported in Section 8.4): this configuration is an **extension** of the previous one, in which it is also tested the case of having **two *Kafka* instances running in parallel**, processing **simultaneously** the **traffic** received by the **platform**. The idea is to **compare** the **performance** achieved in **this configuration** with the one obtained in the **single server case**. These tests were only performed in the ***Kubernetes* testbed**, as it is currently the **more mature technology** that enables the **orchestration of multiple instances** of the **same service**.

Regarding the **parameters** that allow to fully define the tests, this is the **main information** to have in mind about it:

- **Design parameters:** they are related to **input data** to the **system** in order to **configure properly** the **Monitoring platform** for the **tests**. We can distinguish between:

- **Fixed:**
  - **Message size:** for each experiment, there will be **8 topics** managing **100 B messages**, **8 topics** managing **1 KB messages**, **2 topics** managing **100 KB messages** and **2 topics** managing **1 MB messages**. The **graphs** reported in the **next sections** are related to the **100 B messages**.
  - **Test duration:** **5 minutes**.
  - **Number of test repetitions:** **10 repetitions**.
- **Variable:**
  - **Number of topics:** **20 topics per experiment**, varying **from 1 to 6 experiments**. This parameter determines the **throughput received** by the Monitoring platform (**around 102,4** *Mbps* **per experiment**).

- **Performance parameters:** these are the **parameters measured** during the **execution** of the **tests**, which can be:
  - **CPU consumption:** measured on **Tardis server** with a **CPU collector** based on the ***mpstat* command**. For having **similar results** on **all testbeds**, **all the tools** that are **not** going to be **used** for a particular **testbed** must be **turned off** (*e.g.* the virtual machine with *Kafka* and *ZooKeeper* only applies to the virtual testbed, but not in the other scenarios).
  - **Batch write latency:** the **time** spent until receiving an **ACK message** from the ***Kafka* broker**.
  - **I/O message rate:** the **received throughput divided** by the **publication rate**.

8.3. Single Server Performance Evaluation

For the **first set of tests** executed on each testbed, in which **only one *Kafka* broker** was present, the following **results** were obtained, in terms of the different **performance parameters** defined in Section 8.2:

- In the case of the **CPU consumption**, whose results for all testbeds can be found in Figure 70, the **saturation effect** observed in Chapters 3 and 4 **was present**. This means that the **CPU consumption increases its value** when **increasing the number of experiments** deployed **until** reaching a **hard limit** (*i.e.* the saturation point), obtaining around 27 % for the physical, virtual, *Docker* and *Kubernetes* testbeds, and 6-8 % for both *Kata* testbeds. Comparing all the scenarios, the following **tendencies** are observed:
  - The **physical, *Docker* and *Kubernetes* testbeds** present a **similar tendency**[17], **starting** with a value of **8-10 % for one experiment**

[17] In the case of the analysis made on Chapters 3 and 4, in which containers were also used within virtual machines, the results were similar to the virtual testbed presented in this analysis.

and reaching the **hard limit between 3 and 4 experiments** deployed in the system.

- The **virtual scenario** has a **higher consumption profile** at the **beginning**, as it **saturates sooner** (with **2 experiments**), but it eventually **reaches the same values** than the previous case.
- On the other hand, **both *Kata* options saturate much sooner**, with a **lower throughput received** (*i.e.* less than 102,4 *Mbps*). As a result, the **CPU consumption remains constant** with a **lower value** compared to the other scenarios, making also an impact on the other parameters under study.

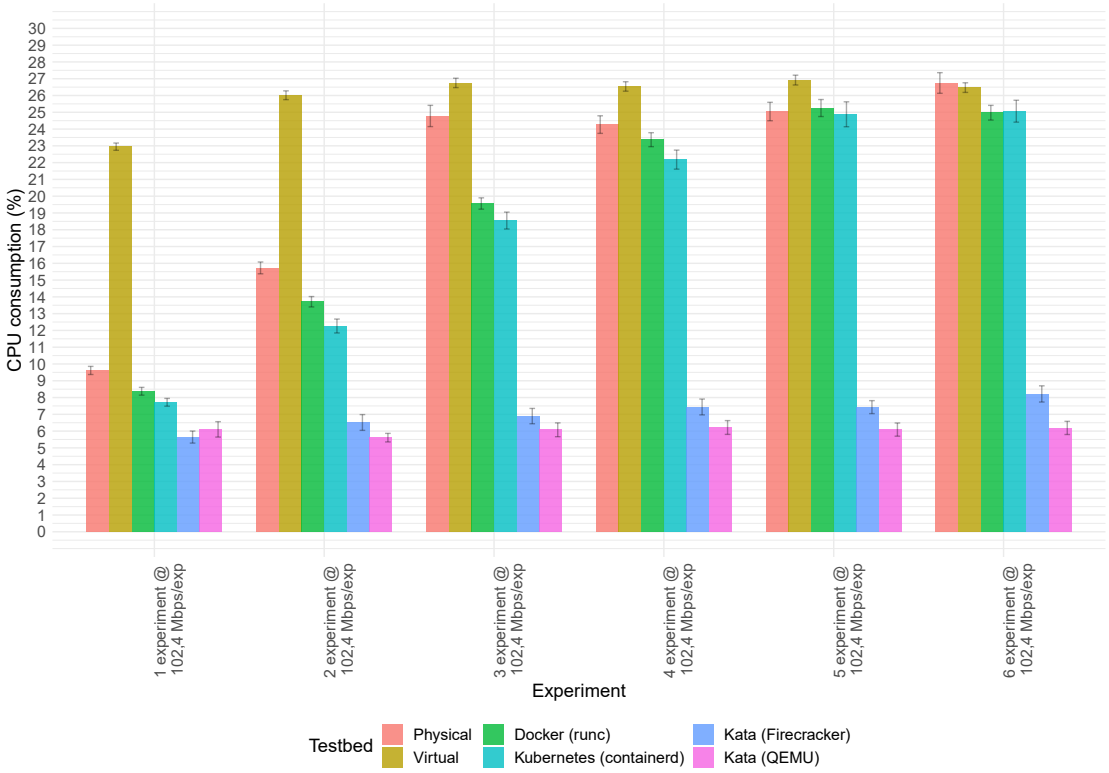

Figure 70: CPU consumption evolution for all the testbeds of the Monitoring platform (100 B messages).

- The **batch write latency**, whose related results are presented in Figure 71, also **increases its value** when the **number of experiments deployed becomes higher**. For this parameter, **three different trends are also observed**, but with **different implications** compared to the **CPU consumption**:
  - Again. the **best results** are observed for the **physical, *Docker* and *Kubernetes* testbeds**, with a batch write latency value **lower than 3 ms in the worst case** (*i.e.* with 6 experiments deployed).

This happened because containers adapt their performance and consumption to the environment in which they are deployed.

- In the second place, it comes the **virtual testbed**, and also the ***Kata+Firecracker* scenario**, with **one order of magnitude more** than the previous case (**around 40 ms** in the **worst case**). This happens because the **hypervisor's access to disk** is **different** than with **containers** (and obviously with a direct access to the disk, which is the case of the **physical scenario**), as **containers share resources with the host** (*i.e.* directly the physical server).
- Finally, the **worst results** were obtained for the ***Kata+QEMU* scenario**, having **another order of magnitude more** compared to the previous case (**around 200 ms** in the worst case) due to a ***heavy* packet loss** process experienced, as it will be observed in the analysis of the I/O message rate. According to these results, it seems that the capabilities offered by ***Firecracker***, with **similar features than *KVM*** to manage the **access to disk**, allows *Kata Containers* to achieve a **better performance profile**, and that is **not the case for *QEMU***, which is the default hypervisor used with *Kata*.

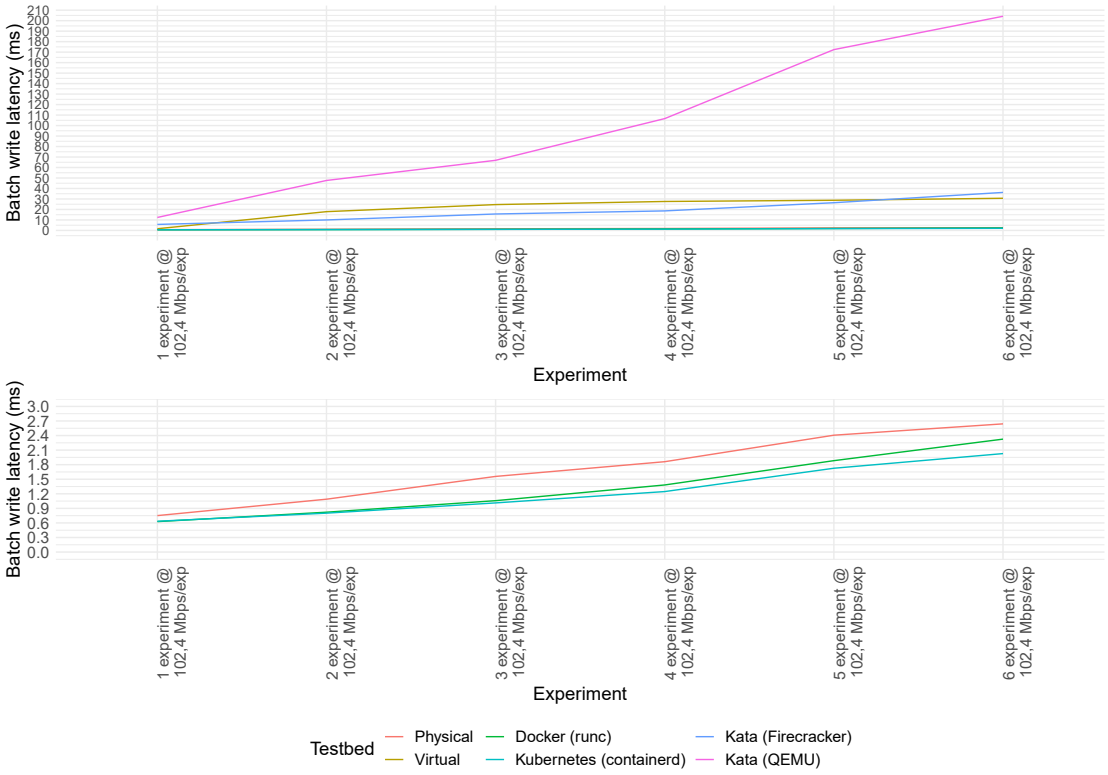

Figure 71: (Top) Batch write latency evolution for all the testbeds of the Monitoring platform, (bottom) also detailing the results for the testbeds with lower values (100 B messages).

- And finally, the **evolution of the I/O message rate**, according to the results presented on Figure 72, also depends on the **saturation effect** experienced in *Kafka*: when it **appears, packets start to be lost**, causing a **lower I/O message rate** when **increasing the number of experiments**

executed (*i.e.* the throughput received by the platform). In this way, the **three tendencies** observed for the **CPU consumption** are also **repeated** here with a clear **correlation** between results:

- First of all, the **physical, *Docker* and *Kubernetes* testbeds** present the **same tendency** and the **highest values possible**, with an **I/O message rate** of around **0,75** in the **worst case** (*Docker* testbed with 6 experiments deployed). According to the **moment** in which the **I/O message rate starts to fall**, it is confirmed that the **saturation point** seems to be **between 3 and 4 experiments** deployed for all cases.
- Then, it comes the **virtual testbed**, in which the **saturation point** is produced with **2 experiments**, as commented in the CPU consumption analysis, and achieving a value of around **0,25** in the **worst case**. This trend is, in fact, the one observed in the analysis done in Chapters 3 and 4, as containers were used in a VM as host.
- And finally, **both *Kata* options** have a **poor performance**, **starting** with **around 0,65** (*QEMU*) and **0,5** (*Firecracker*) **for 1 experiment** and **falling** to **less than 0,2 for 6 experiments**.

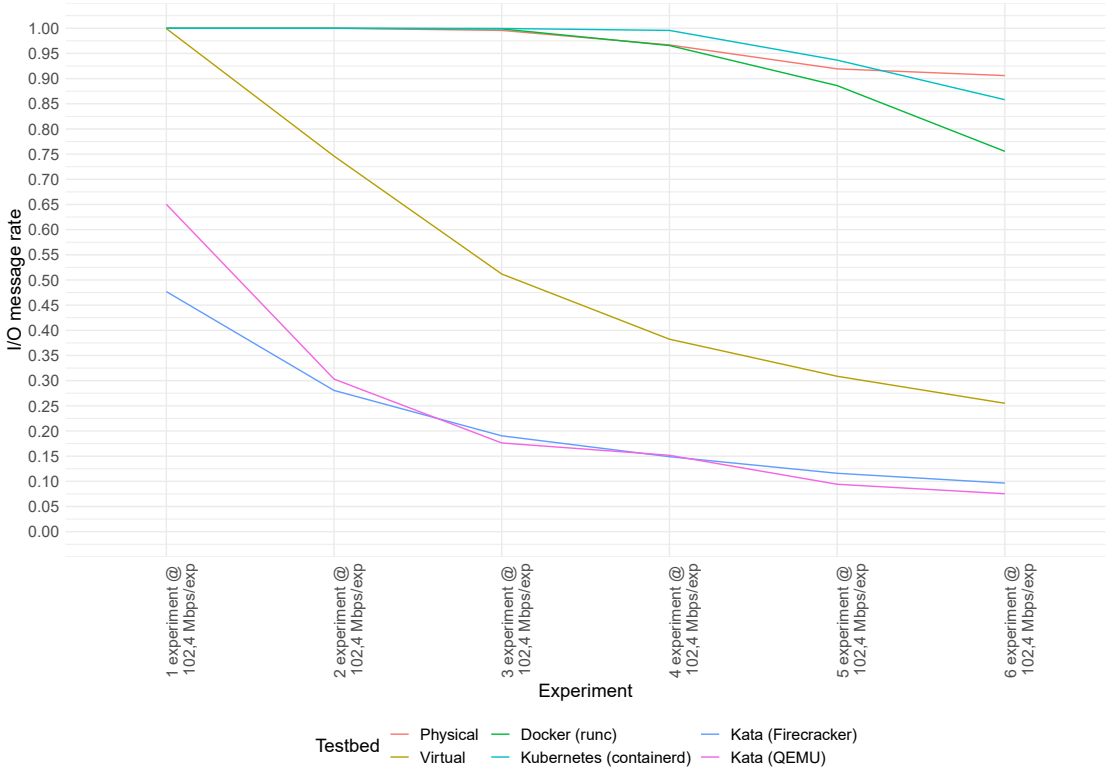

Figure 72: I/O message rate evolution for all the testbeds of the Monitoring platform (100 B messages).

To **conclude** with this analysis, and also to **understand** what is really **happening** on the ***Kata* testbed**, it is interesting to **analyze** if, according to

what it is expected after checking the other testbeds, the **saturation point** is **achieved** for a **throughput lower than 102,4** *Mbps* for the *Kata* testbed. For this purpose, a **new set of tests** was executed by creating **experiments with 1, 5, 10 and 15 topics**, and **dividing the total throughput** of one experiment (*i.e.* 102,4 *Mbps*) **on these topics** (*e.g.* the experiment with 1 topic would imply a total throughput of 5,12 *Mbps*.

After executing the results, the **same three performance parameters were analyzed**, observing the following:

- For the **CPU consumption**, depicted in Figure 73, it is observed that **it increases its value** until reaching a **limit of around 6 %** for **10 topics**, so that the **saturation point** may be **between 5 and 10 topics**.

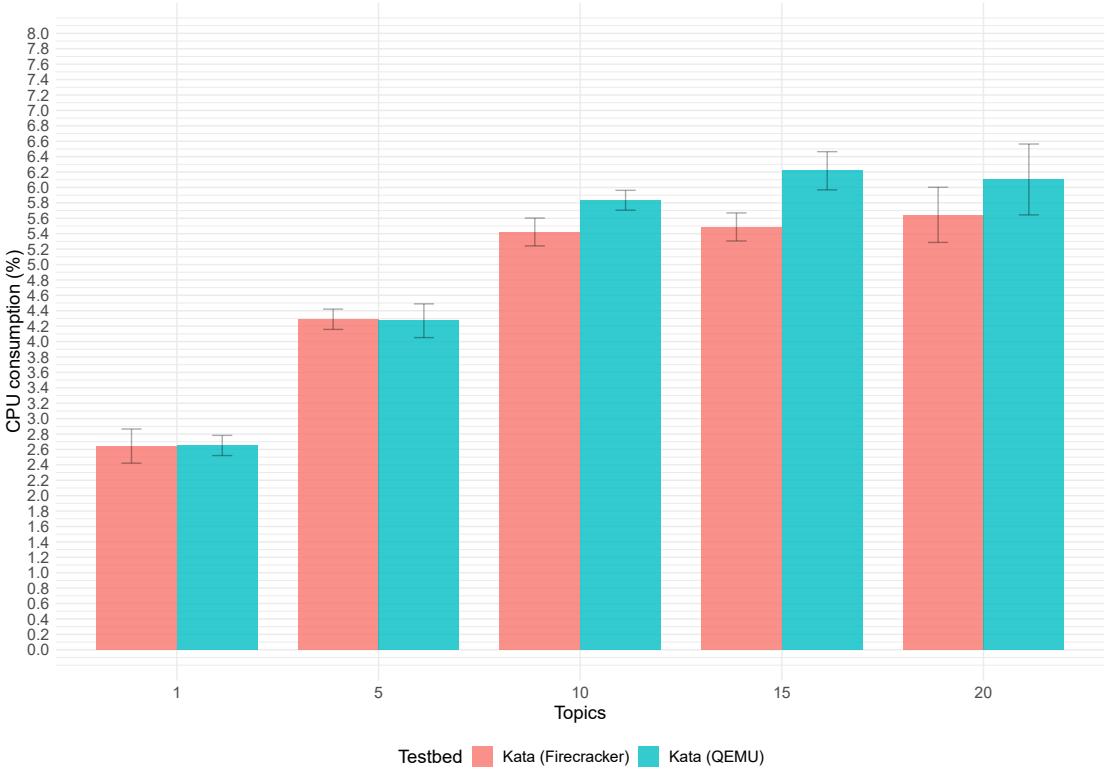

Figure 73: CPU consumption evolution for the *Kata* testbeds with a workload lower than 1 experiment (100 B messages).

- In the case of the **batch write latency**, observed in Figure 74, the **minimum value** obtained is **1 ms for 1 topic**, and **reaching 3 ms for 10 topics** (which was the **highest value** observed for the **physical, *Docker* and *Kubernetes* testbeds** in the **worst case**). This means that these **serverless technologies** are **not really good** at **I/O performance**, also taking into account that ***Kafka*** requires a **high disk performance**, so this would be the **worst scenario possible** to analyze the performance of these **tools**.

  Moreover, **from 10 topics in advance** (*i.e.* after reaching the saturation point), the **slope** of the **curve** related to the ***Kata+QEMU* testbed** starts

to be **greater**, according to what was commented before (*i.e.* ***Firecracker*** seems to **manage better** the **resources' usage**).

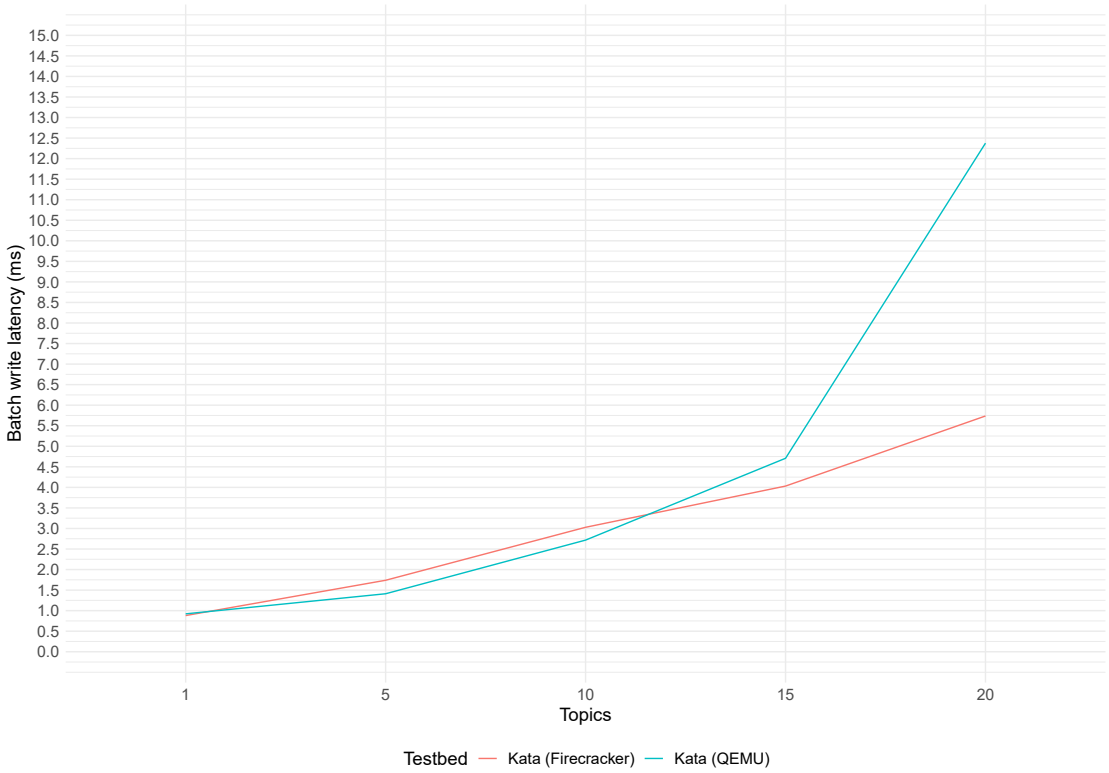

Figure 74: Batch write latency evolution for the *Kata* testbeds with a workload lower than 1 experiment (100 B messages).

- And finishing with the **I/O message rate**, presented in Figure 75, it is clear that the ***Kafka*** **saturation point** is achieved **between 5 and 10 topics**, as the **I/O message rate** is **1 for 5 topics**, and it **starts falling** in the **next case evaluated**. ***Kata+Firecracker*** has a **deeper fall** at the **beginning**, but as observed on Figure 72, the **results** for ***Kata+QEMU*** become **worse** when **increasing** the number of **experiments deployed**.

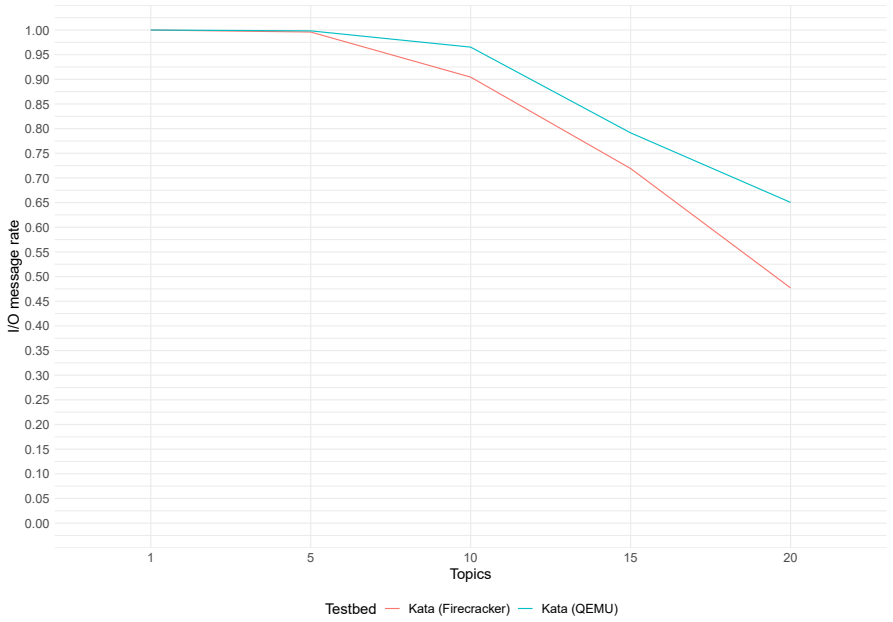

Figure 75: I/O message rate evolution for the *Kata* testbeds with a workload lower than 1 experiment (100 B messages).

8.4. Horizontal Scaling Performance Evaluation

8.4.1. Update of the Testbed Specification

In order to execute the tests related to the **horizontal scaling case** on the ***Kubernetes* testbed**, it has to be **updated beforehand**, in order to allow the **presence of more than one *Kafka* broker** in the scenario. This enhancement is presented in Figure 76, in which the only difference between this case and the one presented on Figure 68 is that **two *Kafka* brokers** are depicted now, achieved by **using *Kubernetes* services and deployments** instead of pods, so that *Kubernetes* is able to **automatically manage the number of *Kafka* instances running on the platform** (in this case, limited by two).

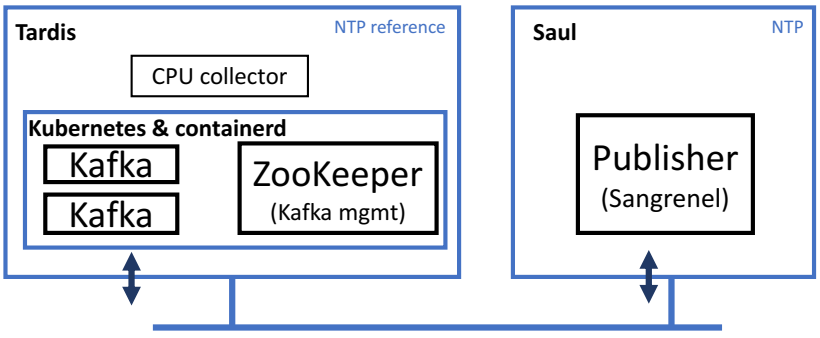

Figure 76: CPU consumption evolution for the *Kubernetes* testbed with two *Kafka* brokers (100 B messages).

8.4.2. Results Obtained

For the test cases, the **same procedure** explained on Section 8.2 were **followed**, obtaining the following **results**:

- In the case of the **CPU consumption**, showed on Figure 77, it was expected to have a **higher value**, as there were **two *Kafka* containers** running **instead of one**, having the **theoretical limit** of the **double value**. However, the result for two brokers **never reaches the double value** and it **always increases its value**, meaning that the **saturation point** was achieved for the **worst case** (or it was **even not reached**, directly).

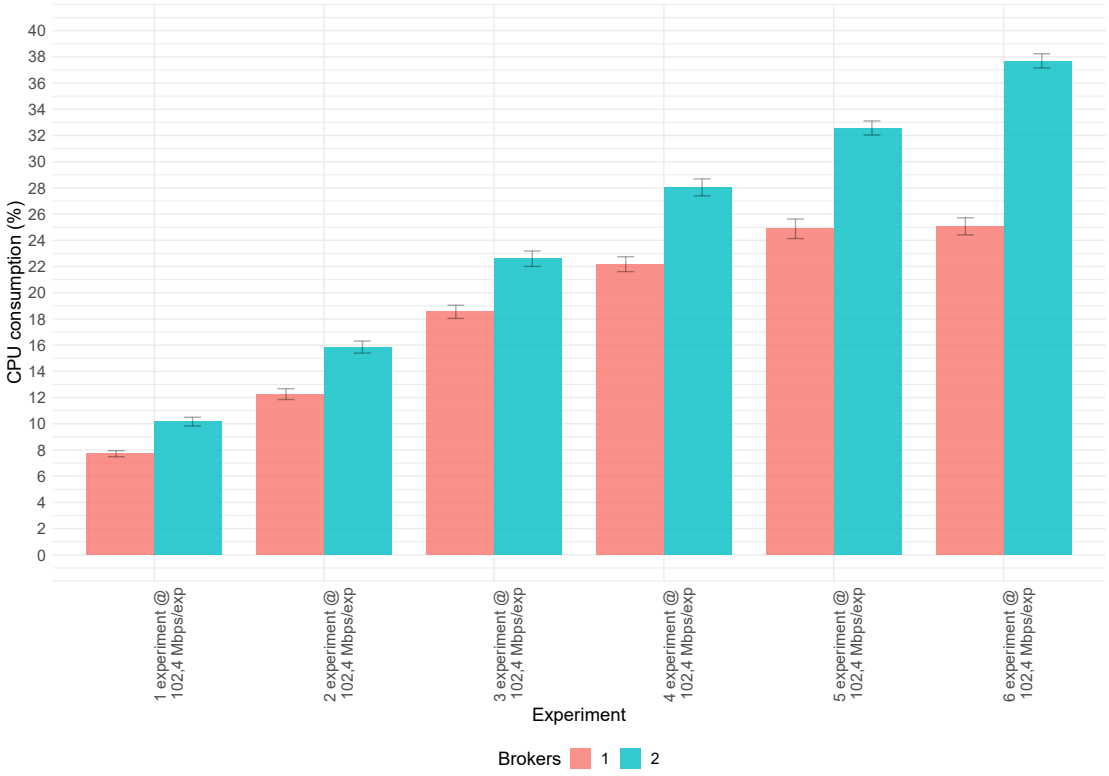

Figure 77: CPU consumption evolution for the *Kubernetes* testbed with two *Kafka* brokers (100 B messages).

- In the case of the **batch write latency**, presented in Figure 78, the **values** obtained for the **two-broker scenario** are always **lower** than the **single-broker case**, also with a **lower slope**. In fact, for the **worst case**, the **latency** is **reduced** to the **half** approximately, achieving around **1 ms**.

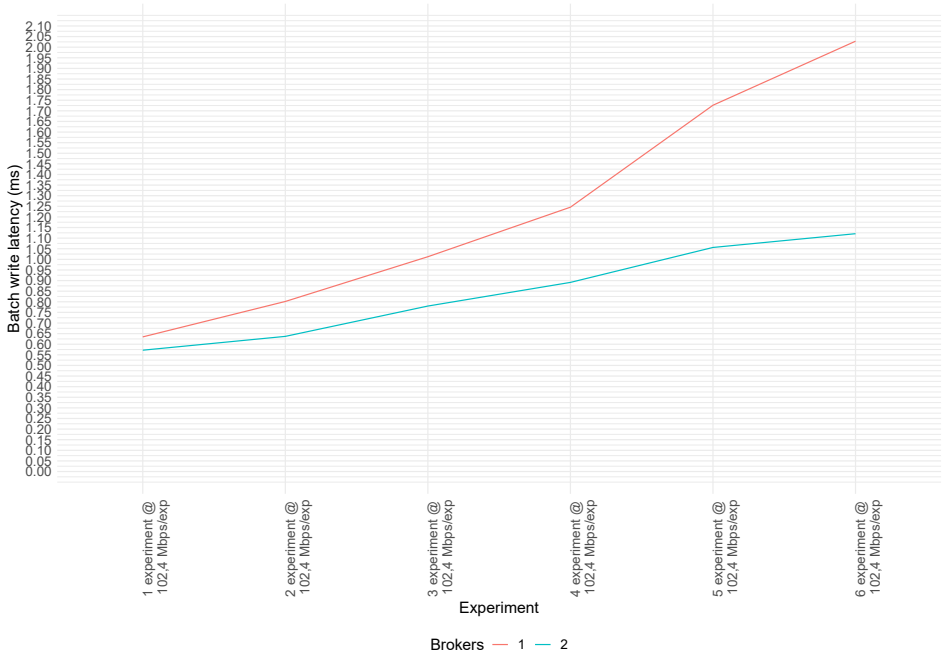

Figure 78: Batch write latency evolution for the *Kubernetes* testbed with two *Kafka* brokers (100 B messages).

- And finally, Figure 79 depicts the **I/O message rate evolution**, which is **clearly constant** for the **two-broker scenario**, validating that the **saturation point is not reached** before the worst case analyzed and **achieving**, consequently, a **better system's performance** by including horizontal scaling capabilities.

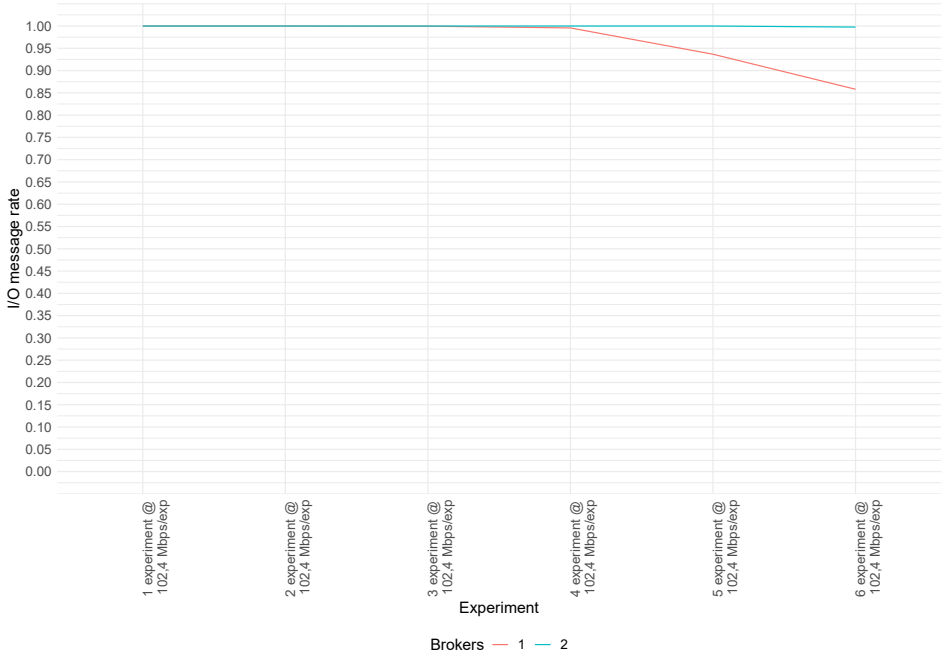

Figure 79: I/O message rate evolution for the *Kubernetes* testbed with two *Kafka* brokers (100 B messages).

8.5. Summary

This Chapter has analyzed in depth a set of **testbeds** related to the **Monitoring platform**, in which **different virtualization techniques** have been **applied**, in order to check the **performance** obtained on each option, with the main objective of **positioning the serverless-related testbeds** (*i.e. Kata+QEMU* and *Kata+Firecracker*) in the **state of the art** and validating it **suitability** for this kind of scenarios.

From this analysis, it can be concluded that the different **testbeds** can be **classified** in **three groups**, according to the results obtained: in the **first group**, it appears the **physical, *Docker* and *Kubernetes* testbeds**, which have a **similar performance profile**[18] and the **best results**, making sense to use **containers over physical servers** for **lightweight software**.

The **second group** is mainly composed by the **virtual testbed**, offering a **worse performance** (probably due to **not having direct access to the host's resources**), but with **reasonable values** to use it in **production environments**.

And in **third place**, the results for **both *Kata* testbeds**, compared with the others, were the **worst ones**, **going to scenarios** with **less than 20 topics** to **confirm** that these technologies are **working properly**. As a result, there are **clear evidences** that, based on the issues reported and the problems found during the tests, it can be supposed that the **serverless technology** is **not mature yet** for using it in **real, production environments**.

Of course, this evidence cannot be extrapolated at all for all cases from the results obtained, but this can be done, at least, for the testing process followed. And also, it is expected that the **technology** will be **optimized somehow** in the **near future**, maybe in the topics analyzed on this Chapter or in others, eventually **allowing** then to **achieve a better performance profile**.

Finally, preliminary **horizontal scaling techniques** in the ***Kubernetes* testbed** have also been analyzed, confirming that the **performance profile improves considerably** (*e.g.* the I/O message rate for the scenario evaluated was maintained to 1 for all experiments evaluated) and that could be a candidate for the **extension and improvement** of the **platform** in **future releases**.

[18] Remember that *Docker* and *Kubernetes* testbeds are in this position due to the fact that the host is directly the physical server and not a VM.

9
Conclusions and Future Work

To **summarize** the different **topics** analyzed over this thesis, the main **conclusions** related to **each topic**, together with the **future work** which we expect that will result in a **single solution** that integrates all the technologies, platforms and services, will be described.

9.1. Conclusions

First of all, starting with the topic related to the **Monitoring of Network Slices**, the work presented in terms of the **Monitoring platform** evaluated, together with its **extension** to fit in **Beyond 5G scenarios**, has resulted in a system capable of managing multiple **streams of data** with a **flexible and adaptable architecture**, based on the **publish-subscribe paradigm**, which has been **evaluated** in terms of some **performance parameters**, validating its **suitability** for **multi-site** and **multi-stakeholder** scenarios.

Moreover, this solution has been **tested** and **validated** in a **real deployment** in the **5G EVE platform**, showing that the **system workflow behaves correctly** when instantiating an experiment in the platform. This aspect is important to confirm that the Monitoring system not only works correctly in a **test environment**, but it is also ready to be integrated in **realistic environments**, which can be even based on **Edge deployments**, as reflected in Chapter 4.

Continuing with the **Orchestration of 5G Transport Networks** topic, the **main contribution** that has been provided in this work is the **definition** of a full **SD-WAN orchestration solution**, which is capable of **managing different SDN-based scenarios** presenting different configurations that are usual in **enterprise and academic networks**, also presenting **specific use cases** and their complete **workflow**.

To get to this point, firstly, the **modeling of the network infrastructure** has been addressed, so as to allow the introduction of the **network intents**. The definition of these intents, in terms of the so-called **Network Intent Descriptors**, has been another relevant **contribution** to the **IBN state of the art**, followed by **Alviu's modular architecture**, which has been designed to properly manage

the **intent-based networking capabilities** applied over **SD-WAN scenarios**. This platform has been **evaluated** to confirm the **system scalability** in terms of the **deployment time** spent by Alviu to achieve **intent's fulfillment**, as the **measured deployment time** depends **linearly** (and not exponentially) on the **number of branches** present in the scenario.

Despite the **promising features** offered by Alviu, it also presents some **limitations**, mainly related to **its maturity** in a market that is growing more and more. For instance, while Alviu allows the **integration** between **SDN and external domains**, this is **limited to specific deployments**, *e.g.* when only having **one endpoint in each branch**. However, this is not what we have in **real environments**, where **the same external domain may be connected to several edge SDN switches** for offering a **fault-tolerant service**.

Finally, regarding the **New Virtualization Techniques** topic, our contribution is driven by the fact that the **Cloud Computing technology** has already identified **solutions** for a **more efficient service provisioning** through the **microservices and the serverless paradigms**, while the **mobile networking community** is **lagging**, still implementing solutions based on **Network Function Virtualization**.

In this way, the introduction of the **serverless** paradigm into the **mobile network stack** implementation could be the key to find the **best trade-off** between **service customization** and **resource efficiency** with the implementation of a really **flexible solution**. However, several **research questions** have to be solved before successfully introducing this paradigm: **new VNFs** shall be **designed** to exploit this paradigm, the **underlying infrastructure** needs to be **prepared**, and **novel orchestration frameworks**, possibly based on machine learning, are **required**.

These **challenges** have been **addressed** with the **design, implementation and testing** of the **Monitoring platform** based on the **serverless paradigm**, also analyzing the **performance level** that can be achieved with the **technologies currently available** in the **state of the art**, **compared** with **other virtualization and containerization techniques**.

From this evaluation process, it can be **concluded** that there is still a **long way to improve** the performance of the **serverless techniques** evaluated, as **poor results** were obtained after evaluating the **performance parameters** under study. It seems, in fact, that ***Kata Containers*** are having problems in the **access to disk**, a capability which is **highly required by *Kafka* to work properly**. In fact, this issue has been already covered in the **state of the art**, with studies like [131] pointing out that *Kata Containers* are **not really efficient** in terms of **memory consumption** and **speed**, but it provides a **secure environment** to run **containers** in **multi-tenant environments**.

9.2. Future Work

Building on the **results obtained** during the evaluation process performed for each topic under study in this thesis, several **subjects** for **future research** can be addressed to **enhance** the **capabilities** offered for all the technologies analyzed over this document, also taking into account their evolution towards

a final **integration** of all the technologies into a **complete solution for 5G networks**, bringing together all the topics studied.

First of all, knowing that the evaluation process in all cases has been based on **synthetic data** (*i.e.* generated by testing scripts), it is desirable to **repeat the experiments with real data** in order to **confirm the results obtained**. This change may require the **adaptation of the technologies used to fit in the requirements** expected for these experiments. For example:

- In the case of the **Monitoring platform**, even though the system has been validated in a real deployment in the 5G EVE platform, a **real implementation** that operates in **Edge environments** is still missing. In any case, the **components needed** to perform that deployment have already been developed and are **publicly available** [104], so it would be just a matter of finding a proper **use case** that may need this functionality in order to perform and test the **integration** in a real case.

- In **Alviu**, considering its **flexibility** to implement **new capabilities** in a relatively **simple way**, its introduction in **other network architectures** could be considered; such as **mobile networks**, where **5G** is currently enabling the introduction of a **wide ecosystem of software** and **Cloud-native technologies**. This way, examples like [132], where SDN is used in a 5G dense multi-infrastructure provider, are now possible with the technologies available.

- In the the case of a **serverless platform** based on ***Kata Containers***, as it has been showed in Section 8.3 that it has the **lowest service capacity** of all the technologies studied, it may requre a **queuing system** to **avoid the packet loss** or the **deployment of more containers** to scale the platform, as proposed for the Kubernetes testbed. Nevertheless, the problem of the **maturity of the technology** and the **suitability of using serverless** techniques over **really critical components** (such as Kafka) will still be there.

In some particular cases, it is also interesting to consider **extending the testing plan to be carried out**, trying to cover **potential features** that may be interesting for **real deployments**. This is the case, for example, of the **integration** between the **Monitoring platform** and the **Data Analytics Framework** presented on Chapter 4.2.2, in order to evaluate a **real Monitoring system combining Artificial Intelligence (AI) and Machine Learning (ML)** techniques in order to improve the **system scalability process**, thus being able to allocate **new compute resources** based on the **information extracted and analyzed** from the **network**. This last topic is really useful for the implementation of **serverless functions**, as they are quite **flexible** to be **allocated** wherever they are required.

The usage of **AI/ML techniques** can also fit in **Alviu**, not only managing a **programmable network**, but also providing **intelligence** to it based on the **current status of the network infrastructure** and the **forecasts made** by these technologies. One example of this is [133], where Machine Learning modules

are introduced to ensure the **automation** and **self-assurance** of the designed Intent-Based Orchestration platform.

Moreover, with regards to the **serverless technologies** under study in Chapter 8, note that the **performance evaluation** done only evaluates *Kafka* and *ZooKeeper*, but it **skips the analysis of the serverless functions** proposed for the Monitoring platform in Section 7.2.3. By including these functions in the performance evaluation study, it could be possible to **compare some time-related metrics** in **different scenarios** involving the **serverless functions**, *e.g.* with **warm or cold start deployments**. These **metrics** may be, for example, the **setup time** (*i.e.* **time** spent to properly **deploy the resources** needed in a specific **moment**) or the **execution time** (*i.e.* **time** spent to **perform** the **complete monitoring workflow**). Moreover, the **impact on resource's isolation** may be also helpful for **guaranteeing the Network Slices' SLAs** created in this kind of environments.

Another topic for future work is the **alignment with standardization efforts**. For example, the **Monitoring framework** can be useful for **filling specific gaps** in **3GPP** standards for certain **5G value-added functionalities**, so that it can be easily integrated as a **complementary module** in those cases, with the goal of **evolving** the Monitoring platform towards a **multi-purpose framework**. This is motivated by the way that **Network Functions' (NFs) Service-based interfaces (SBIs) expose their services** in the **5G Core Control Plane**, based on **publish-subscribe mechanisms** [55]. Some **examples** detected and under tracking are the following: (*i*) **data collection framework** for data retrieval from **Application Functions (AFs)** to be processed by the **Network Data Analytics Functionality (NWDAF)** for enabling **Network Automation processes** [134] (which has been **partially covered** with the inclusion of the **AI-driven Data Analytics framework** from [3]), or (*ii*) inclusion of the **publish-subscribe messaging pattern** in the **communication** between **management service producers and consumers** in a **Management and Orchestration (MANO) architecture**, being an approach similar to the use of the "subscribe-notify"paradigm proposed in [15].

Apart from that example, there are also other **active initiatives**, such as the **ETSI-NFV MANO platform** for the **management and orchestration of network functions** deployed in a given **infrastructure**. In that case, the Monitoring platform may help in the **collection of metrics** from **different sources** (infrastructure, VNFs, etc.) to easily **deliver** them to the **entities** interested in that data. Between these entities, **Alviu** could be placed, **extending** its **capabilities** in order not to only be able to manage the **networking configuration**, but it could also control the **lifecycle of the infrastructure** in a virtualized environment, creating or deleting instances depending on system's workload.

Finally, in addition to **introducing new features** to **widen the use cases**, other future research lines are related to the **introduction of new technologies and paradigms to enhance the user experience** and the **performance of the system**. Taking **Alviu** as example of this; in the first case, the focus is set on the evolution of **intent** definition towards a ***human-based language***, where the intent could be defined in terms of **understandable sentences** that

are eventually **translated into operations and configurations** to be managed by the orchestrator. Current solutions like [135], which introduces the Intent-Based Network Modeling (NEMO) language, are possible models to follow, with the objective of **leveraging SDN and IBN to the next level**, with the so-called **Human-Defined Networking**; where **information and interactions** are provided by the **network managers** (*i.e.* the human part of the network) instead of being strictly software-dependent or even hardware-dependent [136].

A
Examples of Descriptors

A.1. General Configuration Descriptor

The **structure** of the **General Configuration Descriptors** is the following:

- ***regions:*** list of **regions** that will be present in the scenario managed by Alviu. The **parameters** that define a region are:
  - ***id:*** unique **identification** for the region.
  - ***name:*** **name** of the region.
  - ***controller:*** specification of the **controller** used in this region, including the following **attributes**:
    - ***type:*** **type of controller** used, mainly ONOS.
    - ***version:*** **version of the controller** used.
    - ***nodes:*** list of **nodes** that are used to build the **controller**, so that it can be configured in **standalone mode** (in case of using only one node) or in **cluster mode** (if there are several nodes). To describe each node, the following parameters must be provided: ***ip*** (the IP address of the controller node) and ***description*** (of the node).
- ***virtual-networks:*** list of **virtual networks** that will be present in the scenario managed by Alviu. The **parameters** that define a virtual network are:
  - ***id:*** unique **identification** for the virtual network.
  - ***name:*** **name** of the virtual network.
  - ***ip_prefix:*** **IP prefix** of the **network**, in CIDR format.
  - ***gateway:*** **IP address** of the **gateway** for this virtual network, being typically a pointer to the SDN Controller.

A possible **example** of this descriptor is the following:

Code 2: General Configuration Descriptor example.

```
{
"regions": [{
   "id": "A",
   "name": "A",
   "controller": {
      "type": "ONOS",
      "version": "1.12.0",
      "nodes": [{
         "ip": "10.10.10.50",
         "description": "ONOS",
       }]
    }
 }],
"virtual-networks": [{
   "id": "virtual.network.spain",
   "name": "Virtual Network Spain",
   "ip_prefix": "10.107.0.0/16",
   "gateway": "10.107.0.1"
 }]
}
```

A.2. Branch Intent Descriptor

The **structure** of the **Branch Intent Descriptors** is the following:

- ***branch:*** **general data** related to the **branch** to be defined. As a result, to declare N branches in the scenario, N Branch Intent Descriptors will be needed. The **parameters** that are included in this field are the following:
  - ***id:*** unique **identification** for the branch.
  - ***name:*** **name** of the branch.
  - ***region:*** identification of the **region** (and related **SDN Controllers**) to which this branch have to be attached. The definition of the region is done in a different descriptor related to general configuration of Alviu; including, among others, the type of controller used (*e.g.* ONOS), its version and the IP addresses exposed to Alviu and to the branches. An example of region configuration can be found in Annex A.1.
- ***switches:*** list of the **physical SDN switches** that belongs to the branch. The **parameters** that can be configured are:
  - ***id:*** unique **identification** for the switch.
  - ***name:*** **name** of the switch.
  - ***branch:*** identification of the **branch** to which this switch belongs.
  - ***ip_mgt:*** **management IP address** of the switch, reachable from the SDN Controller that manages the corresponding branch.
  - ***port:*** **TCP port** to establish a **reverse SSH tunnel** between the SDN Controller and the SDN switch for configuration purposes.
  - ***user:*** **user name** for the reverse SSH tunnel authentication.
  - ***password:*** **password** for the reverse SSH tunnel authentication.
  - ***product_uuid:*** unique **UUID** of the switch, for zero-touch deployment activation of the switch during the on-boarding phase.
  - ***location:*** **physical location** of the switch, which can be used to fix the position of the switch in the network diagram presented through the Alviu's GUI. To define this, the following attributes are used:
    - ***latitude:*** **latitude** related to the location, in Decimal Degrees format.
    - ***longitude:*** **longitude** related to the location, in Decimal Degrees format.
- ***virtual-switches:*** indicates the **virtual switches** (mainly implemented with Open vSwitches) that have to be deployed in a given switch, needing at least one to work properly. The **attributes** that allow to configure an OVS are:
  - ***id:*** unique **identification** for the virtual switch.

- ***switch:*** identification of the **switch** to which this virtual switch belongs.
- ***name:*** **name** of the virtual switch.
- ***type:*** virtual switch **classification**, which can be **WAN** (if the virtual switch only manages WAN traffic), **LAN** (the same but for LAN traffic) or **both** (if it manages WAN and LAN traffic).
- ***ports:*** the list of **ports** that belongs to the virtual switch, which can be **physical ports** of the switch in which the virtual switch is deployed, or specific-purpose **virtual ports** (*e.g.* to have a loopback interface for a service that may need it). The following **fields** have to be provided to fully define a port:
  - ***interface:*** name of the **port interface**.
  - ***type:*** **type of traffic** that the port will handle, including **WAN** traffic, **LAN** traffic or connections with switches from **external domains**, using legacy protocols for that purpose. This last type of traffic is called **TRUNK** within Alviu's scope.

- ***networks:*** the list of **networks** managed by the branch, characterized by the following **parameters**:
  - ***id:*** unique **identification** for the network.
  - ***name:*** **name** of the virtual switch.
  - ***ip_prefix:*** **network IP address**, in CIDR format.
  - ***virtual_network:*** identification of the **virtual network** to which this network belongs. The definition of the virtual network, in the same way that the region, is done in a different descriptor related to general configuration of Alviu; but it mainly includes the IP prefix of the virtual network. An example of virtual network configuration can be found in Annex A.1.
  - ***dhcp:*** configuration to be provided to the **DHCP module** from the SDN Controller to automatically provide **network configuration** to the **virtual switches**. If defined, the **attributes** that have to be provided are:
    - ***virtual_switch:*** identification of the **virtual switch** to which this network belongs.
    - ***ports:*** the list of **ports** of the virtual switch, identified by the ***interface*** name, where the network is present.
    - ***static_ips:*** a list of **static MAC-IP mapping**, used for cases where some hosts must have a specific IP address. Each element of the list must have the ***mac*** and the ***ip*** address.

- ***igp-speakers:*** modules that interacts with **external domains** that are present in the scenario to be managed. In this case, a IGP speaker is defined for each switch of the branch that is connected to one or several external domains (*i.e.* although a switch is connected to N external domains, only

one IGP speaker is declared). The **parameters** that characterizes each IGP speaker are:

- ***id:*** unique **identification** for the IGP speaker.
- ***mac:*** **MAC address** used by the IGP speaker for the communication with the external domains.

- ***transit-points:*** the **transit points** related to a given **IGP speaker** that may interact with external domains in case of **receiving IGP traffic** from them. The **attributes** that have to be provided are:
  - ***id:*** unique **identification** for the transit point.
  - ***name:*** **name** of the transit point.
  - ***igp-speaker:*** identification of the **IGP speaker** to which this transit point belongs.
  - ***virtual_switch:*** identification of the **virtual switch** to which this transit point belongs.
  - ***port:*** the **port** of the virtual switch, identified by the ***interface*** name, where the transit point is present. The **port type** must be **TRUNK**.
  - ***ip:*** the **IP address** used by the IGP speaker for the communication to the external domain in the interface defined above.
  - ***ip_prefix:*** the **network**, in CIDR format, used for the **connection** between the **IGP speaker** and the **external domain**.
  - ***igp-context:*** **information** related to the type of IGP connection to be established with the external domain, including:
    - ***proto:*** **IGP protocol** used (*e.g.* OSPF or BGP).
    - ***id:*** **identificator** to be used by the IGP protocol (*e.g.* area in OSPF, AS in BGP).

A possible **example** of this descriptor is the following:

Code 3: Branch Intent Descriptor example.

```
{
"branch": {
   "id": "spain.madrid",
   "name": "Madrid",
   "region": "A",
 },
"switches": [{
   "id": "spain.madrid.switch",
   "name": "Madrid Switch",
   "branch": "spain.madrid",
   "ip_mgt": "10.10.10.100",
   "port": 30000,
   "user": "user",
   "password": "changeme",
```

```
    "product_uuid": "A19044A5-14D3-E841-B9BB-87B2244G5013"
    "location": {
      "latitude": 40.4167,
      "longitude": -3.70325
    }
  }],
 "virtual-switches": [{
    "id": "spain.madrid.switch.wan",
    "switch": "spain.madrid.switch",
    "name": "Madrid",
    "type": "BOTH",
    "ports": [ {
        "interface": "ge-1/1/1",
        "type": "WAN"
     }, {
        "interface": "ge-1/1/2",
        "type": "TRUNK"
     }, {
        "interface": "ge-1/1/3",
        "type": "LAN"
     }, {
        "interface": "ge-1/1/4",
        "type": "LAN"
     }
    ]
  }],
 "networks": [{
    "id": "spain.madrid.switch.wan.A.network",
    "name": "Madrid Network A",
    "ip_prefix": "10.107.0.16/28",
    "virtual_network": "virtual.network.spain",
    "dhcp": {
       "virtual_switch": "spain.madrid.switch.wan",
       "ports": [{
          "interface": "ge-1/1/3"
        }],
       "static_ips": [{
          "mac": "00:00:00:00:0a:01",
          "ip": "10.107.0.17"
        }]
     }
  }],
 "igp-speakers": [{
    "id": "speaker",
    "mac": "da:00:00:00:00:01"
  }],
 "transit-points": [{
    "id": "spain.madrid.switch.transit.network",
    "name": "Spain Transit Network",
    "igp-speaker": "speaker",
```

```
65      "virtual_switch": "spain.madrid.switch.wan",
66      "port": {
67         "interface": "ge-1/1/2"
68       },
69      "ip": "172.16.1.1",
70      "ip_prefix": "172.16.1.0/24",
71      "igp-context": {
72         "proto": "OSPF",
73         "id": 0
74       }
75    }]
76  }
```

A.3. Connection Intent Descriptor

The **structure** of the **Connection Intent Descriptors** is the following:

- ***connections:*** list of **connection intents** to be declared simultaneously, with the objective of connecting two switches by using a tunneling mechanism. The **parameters** that define a connection are:
  - ***id:*** unique **identification** for the connection.
  - ***name:*** **name** of the connection.
  - ***type:*** **tunneling protocol** used for building the connection (*e.g.* GRE).
  - ***first_switch/second_switch:*** these two entities describes the two **endpoints** of the connection, characterized by the following **attributes:**
    - ***id:*** identification of the **switch**.
    - ***key:*** attribute that allows to configure a **tag** to **mark the packets** that are sent through this logical **tunnel**, mechanism that is used for load balancing purposes.
    - ***interface:*** name of the **WAN port** of the switch used for establishing the tunnel.
    - ***ip:*** **IP address** of the **switch** to be used in the tunnel.
    - ***mac:*** **MAC address** of the **switch** to be used in the tunnel.

A possible **example** of this descriptor is the following:

Code 4: Connection Intent Descriptor example.

```
{
"connections": [{
   "id": "spain.madrid.barcelona",
   "name": "Connection Madrid Barcelona",
   "type": "GRE",
   "first_switch":{
      "id": "spain.madrid.switch",
      "key": "1",
      "interface": "ge-1/1/1",
      "ip": "92.90.100.134",
      "mac": "00:00:01:22:22:01",
   },
   "second_switch":{
      "id": "spain.barcelona.switch",
      "key": "1",
      "interface": "ge-1/1/1",
      "ip": "92.90.100.147",
      "marc": "00:00:01:22:22:02",
   }
 }]
}
```

A.4. Policy Intent Descriptor

The **structure** of the **Policy Intent Descriptors** is the following:

- ***firewall-rules:*** list of **firewall rules** to be declared in specific networks. The **attributes** that can be defined are:
  - ***id:*** unique **identification** for the firewall rule.
  - ***networks:*** **list of networks** in which this firewall rule is going to be applied. For this purpose, the **identification** value of the corresponding networks is used.
  - ***action:*** **action to apply** to the **traffic** that matches the firewall rule (*e.g.* ALLOW or DENY).
  - ***priority:*** the **priority** of the rule (*e.g.* LOW, HIGH), which serves to order the rules. In case of having several rules with the same priority, the deployment time is used to order them.
  - ***ipProto:*** **transport protocol** (*e.g.* TCP and UDP) that matches the rule.
  - ***srcPort:*** **source port** that matches the rule, for **inbound traffic**.
  - ***dstPort:*** **destination port** that matches the rule, for **outbound traffic**.
  - ***traffic:*** **type of traffic** that matches the rule (*e.g.* INBOUND, OUTBOUND or BOTH). Depending on its value, *srcPort*, *dstPort* or both must be defined.
- ***dns-filtering-rules:*** list of **DNS filtering rules** to be declared in specific networks. The **attributes** that can be defined are:
  - ***id:*** unique **identification** for the DNS filtering rule.
  - ***networks:*** **list of networks** in which this DNS filtering rule is going to be applied. For this purpose, the **identification** value of the corresponding networks is used.
  - ***blacklistUrl:*** **URL** to which the DNS filtering rule will be applied (*e.g.* www.facebook.com). This attribute marks the URL as **blacklisted**.
  - ***whitelistDomains:*** **list of networks**, in CIDR format, in which the **traffic** from/to the **blacklisted URL** is **allowed**.
- ***qos-rules:*** list of **QoS rules** to be declared for specific networks. The **attributes** that can be defined are:
  - ***id:*** unique **identification** for the QoS rule.
  - ***networks:*** **list of networks** in which this QoS rule is going to be applied. For this purpose, the **identification** value of the corresponding networks is used.
  - ***name:*** **name** to identify the QoS rule.

- ***weight:*** value between 1 and 7 that serves to **order** the QoS rules in case a traffic flow matches several QoS rules. A higher value means more importance. In case of matching rules with the same weight value, the most specific one is applied.
- ***dscp:*** **DCSP tag** to be applied to the traffic flow that matches the QoS rule, so that it can be redirected to a specific **QoS queue** in the switches that belong to the network in which the rule is applied.
- ***minRate:*** if specified, it fixes the **minimum data rate** for the traffic that matches the rule.
- ***maxRate:*** if specified, it fixes the **maximum data rate** for the traffic that matches the rule.
- ***traffic_classifier:*** finally, this attribute specifies the **traffic flow information** that matches with this rule. It is codified in the following way:
  - ***ethType:*** it indicates the **layer-3 protocol** used (*e.g.* IPv4).
  - ***ipProto:*** it is the **transport protocol** used (*e.g.* TCP or UDP).
  - ***ipSrc:*** it references the **source IP address**, in CIDR format.
  - ***ipDst:*** it references the **destination IP address**, in CIDR format.
  - ***srcPort:*** it references the **source port**.
  - ***dstPort:*** it references the **destination port**.

A possible **example** of this descriptor is the following:

Code 5: Policy Intent Descriptor example.

```
{
"firewall-rules": [{
   "id": "allow.http.traffic.inbound",
   "networks": ["spain.madrid.switch.wan.A.network"],
   "action": "ALLOW",
   "priority": "LOW",
   "ipProto": "TCP",
   "srcPort": "80",
   "traffic": "INBOUND"
 }, {
   "id": "allow.http.traffic.outbound",
   "networks": ["spain.madrid.switch.wan.A.network"],
   "action": "ALLOW",
   "priority": "LOW",
   "ipProto": "TCP",
   "dstPort": "80",
   "traffic": "OUTBOUND"
 }],
"dns-filtering-rules": [{
   "id": "block.facebook",
   "networks": "spain.madrid.switch.wan.A.network",
   "blacklistUrl": "www.facebook.com",
```

```
23      "whitelistDomains": ["10.107.0.17/32"]
24    }],
25   "qos-rules": [{
26      "id": "q0",
27      "networks": ["spain.madrid.switch.wan.A.network"],
28      "name": "Limit outbound HTTP traffic to 1 Mbps",
29      "weight": 9,
30      "dscp": 21,
31      "maxRate": 1000000,
32      "traffic_classifier":{
33         "ethType": "IPv4",
34         "ipProto": "TCP",
35         "ipSrc": "10.107.0.17/32",
36         "ipDst": "0.0.0.0/0",
37         "srcPort": "ALL",
38         "dstPort": "80",
39      }
40    }]
41   }
```

References

[1] M. Gramaglia, V. Sciancalepore, F. J. Fernandez-Maestro, R. Perez, P. Serrano, and A. Banchs, "Experimenting with SRv6: a Tunneling Protocol supporting Network Slicing in 5G and beyond," in *2020 IEEE 25th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 2020, pp. 1–6.

[2] R. Perez, J. Garcia-Reinoso, A. Zabala, P. Serrano, and A. Banchs, "A Monitoring Framework for Multi-Site 5G Platforms," in *2020 European Conference on Networks and Communications (EuCNC)*, 2020, pp. 52–56.

[3] D. Bega, M. Gramaglia, R. Perez, M. Fiore, A. Banchs, and X. Costa-Perez, "AI-Based Autonomous Control, Management, and Orchestration in 5G: From Standards to Algorithms," *IEEE Network*, vol. 34, no. 6, pp. 14–20, 2020.

[4] M. Gramaglia, P. Serrano, A. Banchs, G. Garcia-Aviles, A. Garcia-Saavedra, and R. Perez, "The case for serverless mobile networking," in *2020 IFIP Networking Conference (Networking)*, 2020, pp. 779–784.

[5] W. Nakimuli, G. Landi, R. Perez, M. Pergolesi, M. Molla, C. Ntogkas, G. Garcia-Aviles, J. Garcia-Reinoso, M. Femminella, P. Serrano, F. Lombardo, J. Rodriguez, G. Reali, and S. Salsano, "Automatic deployment, execution and analysis of 5G experiments using the 5G EVE platform," in *2020 IEEE 3rd 5G World Forum (5GWF)*, 2020, pp. 372–377.

[6] R. Perez, J. Garcia-Reinoso, A. Zabala, P. Serrano, and A. Banchs, "A Distributed Framework Based on Publish-Subscribe to Monitor Beyond 5G Networks," in *EURASIP Journal on Wireless Communications and Networking*, 2020.

[7] R. Perez, A. Zabala, and A. Banchs, "Alviu: An Intent-Based SD-WAN Orchestrator of Network Slices for Enterprise Networks," in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft) (NetSoft 2021)*, 2021.

[8] "5G EVE," https://www.5g-eve.eu/, last accessed: 11 January 2021.

[9] Telcaria Ideas S.L., "Alviu Brochure," Feb. 2018. [Online]. Available: https://www.telcaria.com/docs/sd-wan/Alviu_Brochure.pdf

[10] M. Gupta, R. Legouable, M. M. Rosello, M. Cecchi, J. R. Alonso, M. Lorenzo, E. Kosmatos, M. R. Boldi, and G. Carrozzo, "The 5G EVE End-to-End 5G Facility for Extensive Trials," in *2019 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2019, pp. 1–5.

[11] 5G EVE, "5G EVE end to end reference architecture for vertical industries and core applications," Deliverable D1.3, Dec. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3628333

[12] C. Papagianni, J. Mangues-Bafalluy, P. Bermudez, S. Barmpounakis, D. De Vleeschauwer, J. Brenes, E. Zeydan, C. Casetti, C. Guimarães, P. Murillo, A. Garcia-Saavedra, D. Corujo, and T. Pepe, "5Growth: AI-driven 5G for Automation in Vertical Industries," in *2020 European Conference on Networks and Communications (EuCNC)*, 2020, pp. 17–22.

[13] A. de la Oliva, X. Li, X. Costa-Perez, C. J. Bernardos, P. Bertin, P. Iovanna, T. Deiss, J. Mangues, A. Mourad, C. Casetti, J. E. Gonzalez, and A. Azcorra, "5G-TRANSFORMER: Slicing and Orchestrating Transport Networks for Industry Verticals," *IEEE Communications Magazine*, vol. 56, no. 8, pp. 78–84, 2018.

[14] 3GPP, "Architecture Enhancements for 5G System (5GS) to Support Network Data Analytics Services (Release 16)," TS 23.288 v16.1.0, Jun. 2019. [Online]. Available: https://www.3gpp.org/DynaReport/23288.htm

[15] ——, "Management and orchestration; Architecture framework," TS 28.533, v16.2.0, Dec. 2019. [Online]. Available: https://www.3gpp.org/DynaReport/28533.htm

[16] O-RAN Alliance, "O-RAN: Towards an Open and Smart RAN," White Paper, Oct. 2018.

[17] ETSI, "Network Transformation; (Orchestration, Network and Service Management Framework)," White Paper No. 32, Oct. 2019.

[18] 3GPP, "Study on integration of Open Network Automation Platform (ONAP) and 3GPP management for 5G networks (Release 16)," TR 28.890 v16.0.0, Mar. 2019. [Online]. Available: https://www.3gpp.org/DynaReport/28890.htm

[19] A. Javed, K. Heljanko, A. Buda, and K. Främling, "CEFIoT: A fault-tolerant IoT architecture for edge and cloud," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, 2018, pp. 813–818.

[20] C. Martín, D. Garrido, M. Díaz, and B. Rubio, "From the Edge to the Cloud: Enabling Reliable IoT Applications," in *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2019, pp. 17–22.

[21] Q. Yuan, X. Ji, H. Tang, and W. You, "Toward Latency-Optimal Placement and Autoscaling of Monitoring Functions in MEC," *IEEE Access*, vol. 8, pp. 41 649–41 658, 2020.

[22] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, 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.

[23] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN Control: Survey, Taxonomy, and Challenges," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 333–354, 2018.

[24] L. Le, B. P. Lin, L. Tung, and D. Sinh, "SDN/NFV, Machine Learning, and Big Data Driven Network Slicing for 5G," in *2018 IEEE 5G World Forum (5GWF)*, 2018, pp. 20–25.

[25] R. Trivisonno, R. Guerzoni, I. Vaishnavi, and D. Soldani, "SDN-based 5G mobile networks: architecture, functions, procedures and backward compatibility," *Transactions on Emerging Telecommunications Technologies*, vol. 26, no. 1, pp. 82–92, 2015.

[26] X. Costa-Perez, A. Garcia-Saavedra, X. Li, T. Deiss, A. de la Oliva, A. di Giglio, P. Iovanna, and A. Moored, "5G-Crosshaul: An SDN/NFV Integrated Fronthaul/Backhaul Transport Network Architecture," *IEEE Wireless Communications*, vol. 24, no. 1, pp. 38–45, 2017.

[27] P. Neves, R. Calé, M. Costa, G. Gaspar, J. Alcaraz-Calero, Q. Wang, J. Nightingale, G. Bernini, G. Carrozzo, Ángel Valdivieso, L. J. García Villalba, M. Barros, A. Gravas, J. Santos, R. Maia, and R. Preto, "Future mode of operations for 5G – The SELFNET approach enabled by SDN/NFV," *Computer Standards & Interfaces*, vol. 54, pp. 229 – 246, 2017, sI: Standardization SDN&NFV.

[28] Cisco, "Cisco Annual Internet Report (2018–2023) White Paper," Mar. 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[29] "AT&T Works With VMware to Combine SD-WAN, 5G Capabilities," Feb 25 2019, verizon Communications Inc; AT&T Inc; Copyright - © 2019 Global Data Point. All Rights Reserved. Provided by SyndiGate Media Inc. (Syndigate.info); Last update - 2019-02-25. [Online]. Available: https://search.proquest.com/docview/2185519006?accountid=14501

[30] G. Pujolle, "Fabric, SD-WAN, vCPE, vRAN, vEPC," in *Software Networks: Virtualization, SDN, 5G, and Security*, 2nd ed. Wiley, 2020, pp. 33–50.

[31] Gartner, "Gartner Magic Quadrant for WAN Edge Infrastructure," Nov. 2019. [Online]. Available: https://www.gartner.com/en/documents/3975600/magic-quadrant-for-wan-edge-infrastructure

[32] J. Casey, "Who are the 20 top/best SD-WAN providers & vendors?" Sep. 2020. [Online]. Available: https://www.netify.co.uk/learning/top-best-sd-wan-providers-vendors

[33] N. Rickard and A. Lerner, "Gartner's 2018 Strategic Roadmap for Networking," May 2018. [Online]. Available: https://www.gartner.com/en/documents/3873650/2018-strategic-roadmap-for-networking

[34] L. Pang, C. Yang, D. Chen, Y. Song, and M. Guizani, "A Survey on Intent-Driven Networks," *IEEE Access*, vol. 8, pp. 22 862–22 873, 2020.

[35] ONF, "Intent NBI – Definition and Principles," ONF TR-523, Oct. 2016. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/TR-523_Intent_Definition_Principles.pdf

[36] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an Open, Distributed SDN OS," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, 2014, p. 1–6.

[37] M. Pham and D. B. Hoang, "SDN applications - The intent-based Northbound Interface realisation for extended applications," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, 2016, pp. 372–377.

[38] ONOS, "ONOS Intent Framework," May 2016. [Online]. Available: https://wiki.onosproject.org/display/ONOS/Intent+Framework

[39] Y. Han, J. Li, D. Hoang, J. Yoo, and J. W. Hong, "An intent-based network virtualization platform for SDN," in *2016 12th International Conference on Network and Service Management (CNSM)*, 2016, pp. 353–358.

[40] T. Irfan, R. Hakimi, A. C. Risdianto, and E. Mulyana, "ONOS Intent Path Forwarding using Dijkstra Algorithm," in *2019 International Conference on Electrical Engineering and Informatics (ICEEI)*, 2019, pp. 549–554.

[41] D. Sanvito, D. Moro, M. Gullì, I. Filippini, A. Capone, and A. Campanella, "ONOS Intent Monitor and Reroute service: enabling plug play routing logic," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, 2018, pp. 272–276.

[42] P. Lin, J. Bi, S. Wolff, Y. Wang, A. Xu, Z. Chen, H. Hu, and Y. Lin, "A west-east bridge based SDN inter-domain testbed," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 190–197, 2015.

[43] F. X. A. Wibowo and M. A. Gregory, "Software Defined Networking properties in multi-domain networks," in *2016 26th International Telecommunication Networks and Applications Conference (ITNAC)*, 2016, pp. 95–100.

[44] ——, "Multi-domain Software Defined Network Provisioning," in *2018 28th International Telecommunication Networks and Applications Conference (ITNAC)*, 2018, pp. 1–7.

[45] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally Deployed Software Defined WAN," in *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China, 2013. [Online]. Available: http://cseweb.ucsd.edu/~vahdat/papers/b4-sigcomm13.pdf

[46] L. He, X. Zhang, Z. Cheng, and Y. Jiang, "Design and implementation of SDN/IP hybrid space information network prototype," in *2016 IEEE/CIC International Conference on Communications in China (ICCC Workshops)*, 2016, pp. 1–6.

[47] P. Jakma and D. Lamparter, "Introduction to the quagga routing suite," *IEEE Network*, vol. 28, no. 2, pp. 42–48, 2014.

[48] A. Rego, S. Sendra, J. M. Jimenez, and J. Lloret, "OSPF routing protocol performance in Software Defined Networks," in *2017 Fourth International Conference on Software Defined Systems (SDS)*, 2017, pp. 131–136.

[49] H. Nakayama, T. Mori, S. Ueno, Y. Watanabe, and T. Hayashi, "An implementation model and solutions for stepwise introduction of SDN," in *The 16th Asia-Pacific Network Operations and Management Symposium*, 2014, pp. 1–4.

[50] C. Marquez, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "How Should I Slice My Network? A Multi-Service Empirical Evaluation of Resource Sharing Efficiency," in *Proceedings of 24th ACM MobiCom 2018*, 2018.

[51] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, May 2016.

[52] P. Aditya, I. E. Akkus, A. Beck, R. Chen, V. Hilt, I. Rimac, K. Satzke, and M. Stein, "Will Serverless Computing Revolutionize NFV?" *Proceedings of the IEEE*, vol. 107, no. 4, pp. 667–678, Apr. 2019.

[53] M. Condoluci and T. Mahmoodi, "Softwarization and virtualization in 5G mobile networks: Benefits, trends and challenges," *Computer Networks*, vol. 146, pp. 65–84, Dec. 2018.

[54] 3GPP, "NG-RAN; Architecture description," TS 38.401, v15.7.0, Jan. 2020. [Online]. Available: https://www.3gpp.org/DynaReport/38401.htm

[55] ——, "System architecture for the 5G System (5GS)," TS 23.501, v16.3.0, Sep. 2019. [Online]. Available: https://www.3gpp.org/DynaReport/23501.htm

[56] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration," *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1657–1681, Sep. 2017.

[57] P. Aditya, I. E. Akkus, A. Beck, R. Chen, V. Hilt, I. Rimac, K. Satzke, and M. Stein, "Will Serverless Computing Revolutionize NFV?" *Proceedings of the IEEE*, vol. 107, no. 4, pp. 667–678, 2019.

[58] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless Computing: Current Trends and Open Problems," *CoRR*, vol. abs/1706.03178, 2017. [Online]. Available: http://arxiv.org/abs/1706.03178

[59] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless Computation with OpenLambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, Jun. 2016.

[60] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 923–935.

[61] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 57–70.

[62] K. Kritikos and P. Skrzypek, "A Review of Serverless Frameworks," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 161–168.

[63] S. K. Mohanty, G. Premsankar, and M. di Francesco, "An Evaluation of Open Source Serverless Computing Frameworks," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2018, pp. 115–120.

[64] A. Randazzo and I. Tinnirello, "Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019, pp. 209–214.

[65] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434.

[66] S. E. Elayoubi, M. Fallgren, P. Spapis, G. Zimmermann, D. Martín-Sacristán, C. Yang, S. Jeux, P. Agyapong, L. Campoy, Y. Qi *et al.*, "5G service requirements and operational use cases: Analysis and METIS II vision," in *2016 European Conference on Networks and Communications (EuCNC)*. IEEE, 2016, pp. 158–162.

[67] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5G: Survey and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 94–100, 2017.

[68] 5G EVE, "Requirements definition and analysis from participant vertical industries," Deliverable D1.1, Oct. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.3530391

[69] ——, "Second implementation of the interworking reference model," Deliverable D3.4, Jun. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3946323

[70] J. Garcia-Reinoso, M. M. Roselló, E. Kosmatos, G. Landi, G. Bernini, R. Legouable, L. M. Contreras, M. Lorenzo, K. Trichias, and M. Gupta, "The 5G EVE Multi-site Experimental Architecture and Experimentation Workflow," in *2019 IEEE 2nd 5G World Forum (5GWF)*. IEEE, 2019, pp. 335–340.

[71] 5G EVE, "Interworking Reference Model," Deliverable D3.2, Jun. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3625689

[72] "Apache Kafka," https://kafka.apache.org/, last accessed: 11 January 2021.

[73] P. Sommer, F. Schellroth, M. Fischer, and J. Schlechtendahl, "Message-oriented Middleware for Industrial Production Systems," in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, Munich, 2018, pp. 1217–1223.

[74] 5G EVE, "First implementation of the interworking reference model," Deliverable D3.3, Oct. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3628179

[75] L. Magnoni, "Modern Messaging for Distributed Sytems," *Journal of Physics: Conference Series*, vol. 608, p. 012038, May 2015.

[76] "Apache ZooKeeper," https://zookeeper.apache.org/, last accessed: 11 January 2021.

[77] 5G EVE, "5G EVE WP3 DCM Handler," Jul 2020. [Online]. Available: https://github.com/5GEVE/5geve-wp3-dcm-handler

[78] ——, "5G EVE WP3 DCM Deployment," Jul 2020. [Online]. Available: https://github.com/5GEVE/5geve-wp3-dcm-deployment

[79] "Beats," https://www.elastic.co/beats/, last accessed: 11 January 2021.

[80] "ELK Stack," https://www.elastic.co/what-is/elk-stack, last accessed: 11 January 2021.

[81] 5G EVE, "Experimentation tools and VNF repository," Deliverable D4.1, Oct. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3628201

[82] ——, "First version of the experimental portal and service handbook," Deliverable D4.2, Dec. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3628316

[83] ——, "Report on benchmarking of new features and on the experimental portal (2nd version)," Deliverable D4.4, Jun. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3946283

[84] ——, "5G EVE WP4 DCS Signalling Topic Handler," Jul 2020. [Online]. Available: https://github.com/5GEVE/5geve-wp4-dcs-signalling-topic-handler

[85] ——, "5G EVE WP4 DCS Kibana Dashboards Generator," Jul 2020. [Online]. Available: https://github.com/5GEVE/5geve-wp4-dcs-kibana-dashboards-generator

[86] ——, "5G EVE WP4 DCS Kibana Dashboards Handler," Jul 2020. [Online]. Available: https://github.com/5GEVE/5geve-wp4-dcs-kibana-dashboards-handler

[87] ——, "5G EVE WP4 DCS-DV Deployment," Jul 2020. [Online]. Available: https://github.com/5GEVE/5geve-wp4-dcs-dv-deployment

[88] B. Nogales, I. Vidal, D. R. Lopez, J. Rodriguez, J. Garcia-Reinoso, and A. Azcorra, "Design and Deployment of an Open Management and Orchestration Platform for Multi-Site NFV Experimentation," *IEEE Communications Magazine*, vol. 57, no. 1, pp. 20–27, 2019.

[89] "Proxmox," https://www.proxmox.com/en/, last accessed: 11 January 2021.

[90] "Ubuntu," https://ubuntu.com/, last accessed: 11 January 2021.

[91] J. Alquiza, "Sangrenel," Mar 2020. [Online]. Available: https://github.com/jamiealquiza/sangrenel

[92] "Docker," https://www.docker.com/, last accessed: 11 January 2021.

[93] 5G EVE, "5G EVE WP4 Monitoring Dockerized Environment," Mar 2020. [Online]. Available: https://github.com/5GEVE/5geve-wp4-monitoring-dockerized-env

[94] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper," in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 227–238.

[95] M. Femminella, M. Pergolesi, and G. Reali, "Simplification of the design, deployment, and testing of 5G vertical services," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–7.

[96] 5G EVE, "Models for vertical descriptor adaptation," Deliverable D4.3, Apr. 2020. [Online]. Available: https://www.5g-eve.eu/wp-content/uploads/2020/05/5geve-deliverabled4.3-final.pdf

[97] V. Ziegler, T. Wild, M. Uusitalo, H. Flinck, V. Räisänen, and K. Hätönen, "Stratification of 5G evolution and Beyond 5G," in *2019 IEEE 2nd 5G World Forum (5GWF)*, 2019, pp. 329–334.

[98] A. Ghosh, A. Maeder, M. Baker, and D. Chandramouli, "5G Evolution: A View on 5G Cellular Technology Beyond 3GPP Release 15," *IEEE Access*, vol. 7, pp. 127 639–127 651, 2019.

[99] I. Tomkos, D. Klonidis, E. Pikasis, and S. Theodoridis, "Toward the 6G Network Era: Opportunities and Challenges," *IT Professional*, vol. 22, no. 1, pp. 34–38, 2020.

[100] G. Wikström, J. Peisa, P. Rugeland, N. Johansson, S. Parkvall, M. Girnyk, G. Mildh, and I. L. Da Silva, "Challenges and Technologies for 6G," in *2020 2nd 6G Wireless Summit (6G SUMMIT)*, 2020, pp. 1–5.

[101] S. Wang, T. Sun, H. Yang, X. Duan, and L. Lu, "6G Network: Towards a Distributed and Autonomous System," in *2020 2nd 6G Wireless Summit (6G SUMMIT)*, 2020, pp. 1–5.

[102] 3GPP, "Architecture enhancements for 5G System (5GS) to support network data analytics services," TS 23.288, v16.1.0, Sep. 2019. [Online]. Available: https://www.3gpp.org/DynaReport/23288.htm

[103] "K3s," https://k3s.io/, last accessed: 11 January 2021.

[104] 5G EVE, "5G EVE Monitoring Dockerized Environment," Oct 2020. [Online]. Available: https://github.com/5GEVE/monitoring_dockerized_environment

[105] "Python," hhttps://www.python.org/, last accessed: 11 January 2021.

[106] Apache Kafka, "Allow consumers to fetch from closest replica," KIP-392, Nov. 2019. [Online]. Available: https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica

[107] "Node.js," https://nodejs.org/en/, last accessed: 11 January 2021.

[108] F. Z. Yousaf, M. Bredel, S. Schaller, and F. Schneider, "NFV and SDN—Key Technology Enablers for 5G Networks," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2468–2478, 2017.

[109] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The Design and Implementation of Open vSwitch," *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, pp. 117–130, 2015.

[110] M. Jiménez, "Extension and evaluation of a SDN orchestration system with multi-table functionalities in ASICs," *End of Degree Project for the Undergraduate Degree in Telecommunication Technology Engineering. University of Seville*, 2020.

[111] P. Rost, C. Mannweiler, D. S. Michalopoulos, C. Sartori, V. Sciancalepore, N. Sastry, O. Holland, S. Tayade, B. Han, D. Bega, D. Aziz, and H. Bakker, "Network Slicing to Enable Scalability and Flexibility in 5G Mobile Networks," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 72–79, May 2017.

[112] D. M. Gutierrez-Estevez, M. Gramaglia, A. de Domenico, N. di Pietro, S. Khatibi, K. Shah, D. Tsolkas, P. Arnold, and P. Serrano, "The path towards resource elasticity for 5G network architecture," in *IEEE WCNCW*, Apr. 2018, pp. 214–219.

[113] I. Sarrigiannis, K. Ramantas, E. Kartsakli, P. Mekikis, A. Antonopoulos, and C. Verikoukis, "Online VNF Lifecycle Management in a MEC-enabled 5G IoT Architecture," *IEEE Internet of Things Journal*, pp. 1–1, 2019.

[114] V. Nagendra, A. Bhattacharya, A. Gandhi, and S. R. Das, "MMLite: A Scalable and Resource Efficient Control Plane for Next Generation Cellular Packet Core," in *Proceedings of the 2019 ACM Symposium on SDN Research*, ser. SOSR '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 69–83.

[115] 3GPP, "Management and orchestration; Provisioning," TS 28.531, v15.2.0, Mar. 2019. [Online]. Available: https://www.3gpp.org/DynaReport/28531.htm

[116] D. Bega, A. Banchs, M. Gramaglia, X. Costa-Pérez, and P. Rost, "CARES: Computation-Aware Scheduling in Virtualized Radio Access Networks," *IEEE Transactions on Wireless Communications*, vol. 17, no. 12, pp. 7993–8006, Dec. 2018.

[117] "DPDK," https://www.dpdk.org/, last accessed: 11 January 2021.

[118] Cilium, "BPF and XDP Reference Guide," 2020. [Online]. Available: https://cilium.readthedocs.io/en/latest/bpf/

[119] Y. Wang et al., "Network Management and Orchestration Using Artificial Intelligence: Overview of ETSI ENI," *IEEE Communications Standards Magazine*, vol. 2, no. 4, pp. 58–65, Dec. 2018.

[120] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "DeepCog: Cognitive Network Management in Sliced 5G Networks with Deep Learning," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, Apr. 2019, pp. 280–288.

[121] "OpenFaaS," https://www.openfaas.com/, last accessed: 11 January 2021.

[122] "KVM," https://www.linux-kvm.org/page/Main_Page, last accessed: 11 January 2021.

[123] "runc," https://github.com/opencontainers/runc, last accessed: 11 January 2021.

[124] "MicroK8s," https://microk8s.io/, last accessed: 11 January 2021.

[125] "containerd," https://containerd.io/, last accessed: 11 January 2021.

[126] "Kata Containers," https://katacontainers.io/, last accessed: 11 January 2021.

[127] "Firecracker," https://firecracker-microvm.github.io/, last accessed: 11 January 2021.

[128] "QEMU," https://www.qemu.org/, last accessed: 11 January 2021.

[129] "Xeon comparison UniPG," https://gist.github.com/TheWall89/01688a7f448d8403da7798bcdf0185bc, last accessed: 11 January 2021.

[130] Vivek Gite, "KVM forward ports to guests VM with UFW on Linux," Aug. 2020. [Online]. Available: https://www.cyberciti.biz/faq/kvm-forward-ports-to-guests-vm-with-ufw-on-linux/

[131] V. Aggarwal and B. Thangaraju, "Performance Analysis of Virtualisation Technologies in NFV and Edge Deployments," in *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 2020, pp. 1–5.

[132] J. Lun and D. Grace, "Software defined network for multi-tenancy resource sharing in backhaul networks," in *2015 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, 2015, pp. 1–5.

[133] T. A. Khan, A. Mehmood, J. J. Diaz Ravera, A. Muhammad, K. Abbas, and W. Song, "Intent-Based Orchestration of Network Slices and Resource Assurance using Machine Learning," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–2.

[134] 3GPP, "Study of Enablers for Network Automation for 5G," TR 23.791, v16.2.0, Jun 2019. [Online]. Available: https://www.3gpp.org/DynaReport/23791.htm

[135] Y. Tsuzaki and Y. Okabe, "Reactive configuration updating for Intent-Based Networking," in *2017 International Conference on Information Networking (ICOIN)*, 2017, pp. 97–102.

[136] E. Rojas, "From Software-Defined to Human-Defined Networking: Challenges and Opportunities," *IEEE Network*, vol. 32, no. 1, pp. 179–185, 2018.