Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

TITLE: Design and analysis of fully virtualized cellular networks based on open-source frameworks

MASTER DEGREE: Master's degree in Applied Telecommunications and Engineering Management (MASTEAM)

AUTHOR: Javier Palomares Torrecilla

ADVISORS: Coronado Calero, Estefanía
Siddiqui Muhammad, Shuaib

DATE: September, 8th 2021

Title: Design and analysis of fully virtualized cellular networks based on open-source frameworks

Author: Javier Palomares Torrecilla

Advisors: Coronado Calero, Estefanía
Siddiqui Muhammad, Shuaib

Date: September 8, 2021

Abstract

Virtualization of cellular networks is one of the key areas of research where technologies, infrastructure and challenges are rapidly changing as 5G system architecture demands a paradigm shift. Adding these necessities to the appearance of new technologies such as Software Defined Networks (SDN) and Network Function Virtualization (NFV), encourage the seeking for the implementation of a more flexible and adaptive architecture, for mobile cellular networks.

In this Master Thesis, a comprehensive view is provided upon various scenarios to enable the deployment of a fully distributed and open-source cellular network, as well as to provide an analysis on the impact that both Radio Access Network (RAN) and Core Network have on the resource utilization (radio and virtualized infrastructure) as the network conditions vary. The prototype proposed has been performed using a 4G setup given the currently limited availability of 5G setup and its high cost and compare with a state-of-the-art deployment using a different virtualization tool. Nevertheless, the results presented in this document could be extended to the 5G scenarios. The reason for this is that this Master Thesis aims to study the viability and the performance of virtualized infrastructure to host the aforementioned network segments, and the frameworks used implement both the 4G and the 5G stacks, and even share the same software modules to implement the network functions regardless of the cellular network.

The analysis of the gathered data expose that fully virtualized cellular network deployments present better performance in case of the flexibility, low setting time and ease to deploy, while keeping the same level of resources usage as the non-virtualized deployments. Furthermore, some future works and directions for research to and develop more flexible and more adaptive deployments are proposed, as well as to expand the analysis with higher network capacities available in 5G.

Acknowledgements

First of all, I would like to thank the i2CAT Foundation, where this Master Thesis has been carried out. I would like to particularly mention the Software Networks Research Area for creating a professional and close working atmosphere, while providing all the necessary help to solve the technical problems that I have faced. They gave me the opportunity to participate in such a current and cutting-edge research project, sharing their efforts and knowledge with me. I want to make a special mention to Dr. Estefanía Coronado Calero, because she has been a great tutor directing this project.

Finally, my deepest thanks to my family and friends for their love and for all the support received during these difficult months for everyone, which have required an additional effort, due to the atypical situation caused by COVID-19.

TABLE OF CONTENTS

INTRODUCTION 1

CHAPTER 1. BACKGROUND TECHNOLOGY 3

1.1. Cellular networks 3
1.1.1. 4G 3
1.1.2. 5G 5
1.1.3. Comparison between 4G and 5G 9

1.2. Network Virtualization 10
1.2.1. Software Defined Network 10
1.2.2. Network Function Virtualization 11
1.2.3. NFV and SDN comparison 13

CHAPTER 2. STATE OF THE ART OF VIRTUALIZED CELLULAR NETWORKS 14

2.1. Cloud native architecture 14

2.2. Virtualization Frameworks 15
2.2.1. Docker 15
2.2.2. Kubernetes 16

2.3. Open-source 5G Frameworks 19
2.3.1. Radio Access Network 19
2.3.2. Core Network 21

2.4. Related work on virtualization tools for building distributed cellular networks 24

CHAPTER 3. RESEARCH WORK 27

3.1. Introduction 27

3.2. Baremetal Deployment 27
3.2.1. Installing software and dependencies 28
3.2.2. Setup and configuration 28
3.2.3. Get the deployment running 30
3.2.4. Problems found 31

3.3. Kubernetes-based Deployment 32
3.3.1. Installing software and dependencies 33
3.3.2. Kubernetes cluster setup 33
3.3.3. Design and creation of the Docker images 33
3.3.4. Design the descriptor file 34
3.3.5. Get the deployment running 37
3.3.6. Problems found 39

3.4. Docker-based Deployment 40
3.4.1. Installing software and dependencies 41
3.4.2. Setup and configuration 41
3.4.3. Get the deployment running 42

CHAPTER 4. PERFORMANCE EVALUATION AND ACHIEVED RESULTS . 43

4.1. Methodology 43

4.2. Results discussion 45
4.2.1. Measuring throughput 46
4.2.2. Measuring resource consumption 48
4.2.3. Measuring forced reconnecting time 53

CONCLUSIONS AND FUTURE WORKS 55

5.1. Conclusions of the work 55

5.2. Future lines of development and research 56

5.3. Sustainability considerations 57

5.4. Ethical and security considerations 57

ACRONYMS 58

ANNEX I: INSTALLING SRSLTE AND OPEN5GS 60

ANNEX II: NODES PREPARATION TO DEPLOY THE K8S CLUSTER 61

ANNEX III: DEPLOYMENT OF THE K8S CLUSTER 63

ANNEX IV: DOCKER IMAGES 65

srsLTE 65
Dockerfile: 65
dns_replace.sh: 66
config.sh: 67
launcher.sh: 67
conf/enb.conf: 68

Open5GS 69
Dockerfile: 69
config.sh: 70
conf/mme.yaml: 70
launcher.sh: 71

ANNEX V: KUBERNETES DESCRIPTOR FILE 72

K8s_deployment.yaml 72

ANNEX VI: CODE TO CLEAN AND PLOT THE RESULTS 74

Throughput average 74
extract_average.py: 74
clean_average.py: 75
plot_avera ge.py: 77

Tempor al 79

extract_temporal.py: 79
clean_temporal.py: 80
plot_temporal.py: 82

Resources **83**
extract_resources.py: 83
plot_resources.py: 85

Forced disconnection **89**
extract_disc.py: 89
clean_disc.py: 90
plot_disc.py: 91

REFERENCES **92**

LIST OF FIGURES

Figure 1.1: Architecture of the 4G network. Image based from [7]. 3
Figure 1.2: 5G Service Pillars. Image taken from [10]. 6
Figure 1.3: Point-to-point 5GC architecture. Image based from [12]. 6
Figure 1.4: SBA 5GC Architecture. Image based from [12]. 7
Figure 1.5: SA and NSA deployment modes. Image based from [13]. 8
Figure 1.6: Simplified SDN architecture. Image taken from [17]. 11
Figure 1.7: ETSI NFV ref. architectural framework. Image taken from [23]. 12
Figure 2.1: Compatibility issues matrix in a traditional deployment. 14
Figure 2.2: Overview of running Docker and some containers. 15
Figure 2.3: Kubernetes cluster diagram. Image taken from [28]. 16
Figure 2.4: CNI connection overview. Image based from [32]. 18
Figure 2.5: OAI 5G RAN project phases. Image taken from [38]. 19
Figure 2.6: 5G-EmPOWER system architecture. Image taken from [44]. 21
Figure 2.7: OAI 5G CN developments in the 5GC. Image taken from [47]. 21
Figure 2.8: Open5GS 4G/5G function representation. Image taken from [49].. 22
Figure 2.9: Cloudification design of mobile network. Image taken from [52]. ... 24
Figure 3.1: Baremetal deployment setup. 27
Figure 3.2: Subscriber registered in the Web-UI. 30
Figure 3.3: Baremetal connectivity achieved. 31
Figure 3.4: Failure on connectivity due to wrong APN. 32
Figure 3.5: Two and three nodes deployments. 32
Figure 3.6: Output of the launched descriptor. 37
Figure 3.7: Open5GS MME console output. 38
Figure 3.8: Open5GS SGW-U console output. 38
Figure 3.9: Open5GS SGW-C console output. 39
Figure 3.10: srsLTE eNB console output. 39
Figure 3.11: Calico pod running error. 40
Figure 3.12: Docker deployment setup. 40
Figure 3.13: Intermittent connectivity achieved with Docker. 42
Figure 4.1: Results comparison between two and three nodes in K8s. 45
Figure 4.2: Comparison between the temporal response of Docker and K8s. . 45
Figure 4.3: Baremetal throughput comparison at diff. distnces with each PRB. 46

Figure 4.4: K8s throughput comparison at different distances with each PRB. 46
Figure 4.5: Docker throughput comparison at diff. distances with each PRB. .. 46
Figure 4.6: Comparison between deployments and PRBs at diff. distances. ... 47
Figure 4.7: Comparison of baremetal CPU resources at different bandwidths. 48
Figure 4.8: Comparison of K8s CPU resources at different bandwidths........... 49
Figure 4.9: Comparison of Docker CPU resources at different bandwidths. 49
Figure 4.10: Comparison of baremetal MEM resources at diff. bandwidths. 50
Figure 4.11: Comparison of K8s MEM resources at different bandwidths........ 51
Figure 4.12: Comparison of Docker MEM resources at different bandwidths... 51
Figure 4.13: Comparison of K8s MEM resources at different distances........... 52
Figure 4.14: Comparison of K8s CPU resources at different distances............ 52
Figure 4.15: K8s forced reconnection time.. 53
Figure 4.16: Baremetal forced reconnection time.. 54
Figure 4.17: Docker forced reconnection time... 54

LIST OF TABLES

Table 1.1: SA and NSA architectures comparison. 9
Table 1.2: Comparison between 4G and 5G technologies. 9
Table 4.1: List of the experiments with their parameters................................. 43
Table Annex.1: K8s cluster nodes information.. 64
Table Annex.2: Calico pods information... 64

INTRODUCTION

In recent years, mobile networks have experienced great developments and wireless communications have become an essential part in our daily lives. Due to this need, consecutive generations of enhanced communication networks have been deployed globally. The new network's main requirements are scalability, and ease of deployment, especially in a new fully distributed network paradigm where network functions are deployed in different locations and even provided by several software frameworks. Adding these necessities is possible thanks to the appearance of new technologies such as Software Defined Networks (SDN) and Network Function Virtualization (NFV), as well as of recent virtualized infrastructure managers, which encourage seeking for the implementation of a more flexible architecture for mobile cellular networks and analyzing their impact in the underlying physical components.

In this context, the main objective of this Master Thesis is to enable the deployment of a fully distributed and open-source cellular network based on virtualization tools and virtualized infrastructure managers, as well as to analyze the impact that both Radio Access Network (RAN) and Core Network (CN) have on the resource utilization (radio and virtualized infrastructure) as the network conditions vary. This central objective can be divided in several ones:

The first objective focuses on studying the state of the art, in order to identify the contributions that other authors have provided in the literature. The second target seeks to explore several open-source frameworks for cellular networks, making a differentiation between frameworks that implement the RAN and the CN functions. Similarly, the third objective completes the analysis of the technical background by getting familiar and understanding some container virtualization environments and their orchestration tools, such as Kubernetes.

The fourth objective represents one of the core parts of this Master Thesis, introducing the prototype proposed based on virtualization tools to deploy a distributed cellular network. The prototype proposed has been performed using a 4G setup given the limited availability of 5G setup and its high cost at the moment. The process to fulfill this goal started with a first phase that consisted on achieving connectivity in a baremetal deployment, to verify that the basic deployment of the open-source modules behaves as expected. The second stage consisted in the deployment of the network using Kubernetes to automate the deployment and the management of the containers. This deployment has been performed separating the CN and the RAN logic into two different worker nodes.

The fifth objective includes a thorough analysis on the impact of deploying such virtualization tools and how different network conditions could determine the resource capacity used. This analysis takes as baseline a baremetal deployed and is compared with a state-of-the-art deployment using Docker containers, without a container orchestration platform. Nevertheless, the process followed to virtualize, interconnect, and orchestrate the functions does not vary regarding the type of cellular network, and the results presented could be extended to the 5G

modes, especially because the frameworks used and analyzed, as srsLTE and open5GS, implement both the 4G and the 5G functionalities, and even share the same software modules to implement the network functions regardless of the cellular network. It is worthy to highlight that this Master Thesis does not aim to study the performance of the network at radio or throughput level, but by contrast, to analyze the behavior of the virtualization resources in the distributed scenarios introduced before.

Finally, the last objective verses on the study of future research lines of work taking as a basis the contributions and conclusions reached in this Master Thesis, as well as to identify possible ethical or social impacts.

This Master Thesis is organized in five chapters and several annexes. The first chapter presents the background technology, where a review of the background on cellular networks is explained in detail. Also, the concept of network virtualization is introduced, explaining and comparing the technologies that enable it. Chapter 2 focuses on the state of the art of virtualized 5G networks. The concepts of cloud native architecture, virtualization frameworks and open-source 5G Frameworks are defined, giving examples in each of them. The last section of Chapter 2 is dedicated to related work in the literature on virtualization tools for building distributed cellular networks. Chapter 3 contains the research work of this project, explaining the configuration and launching of the three deployments (Baremetal, Kubernetes-based deployment and the state-of-the-art Docker-based deployment). The fourth chapter presents the performance evaluation and achieved results of the deployments mentioned before, highlighting the methodology used and the results discussion. Furthermore, Chapter 5 presents the conclusions and future lines of works of this Master Thesis, including some sustainability and ethical considerations. Finally, the content of the annexes includes: (i) the installation and setup processes of the different deployments; (ii) the complete content of the files used for the proper deployment of the fully distributed and open-source cellular network; and (iii) the code used to get the data clean for future analysis.

CHAPTER 1. BACKGROUND TECHNOLOGY

In this chapter, the background technology context in which the Master Thesis is based, is presented. The first part covers an overall description on different cellular network generations such as 4G and 5G. Then, the second part discusses different Network Virtualization technologies such as Software Defined Network (SDN) and Network Function Virtualization (NFV).

1.1. Cellular networks

Over the last years, mobile networks have experienced great developments and wireless communications have become an essential part in our lives. Due to this need, consecutive generations of communication networks have been deployed globally. Nowadays, 4G and 5G networks coexist in commercial deployments. The next subsections detail the evolution of cellular technologies and the network architectures behind them.

1.1.1. 4G

The need to increase the capacity and speed while reducing the latency of the mobile networks caused the creation of the fourth-generation networks, under the name Long Term Evolution (LTE). 4G technology started to be standardized from Release 8 [1] of the 3rd Generation Partnership Project (3GPP). It is an evolution of the LTE standard (LTE-A standard from Release 10 [2] of 3GPP). It has practically the same characteristics as LTE [3], except it supports a mobility reaching of 350 km/h and it has a higher data rate flow (300 Mbps in the uplink and 1 Gbps in the downlink). The LTE-A system employs a SCFDMA [4] scheme for the uplink and a OFDMA [5] scheme for the downlink. It also uses a 4x4 antenna or an 8x8 antenna MIMO technique.

In [6] and, the authors list the main characteristics, as well as the main advantages and disadvantages of 4G. This network generation aims to guarantee a minimum Quality of Service (QoS) level and an improvement of the services provided, even when the user is moving at high speeds. The architecture of 4G is displayed on Figure 1.1.

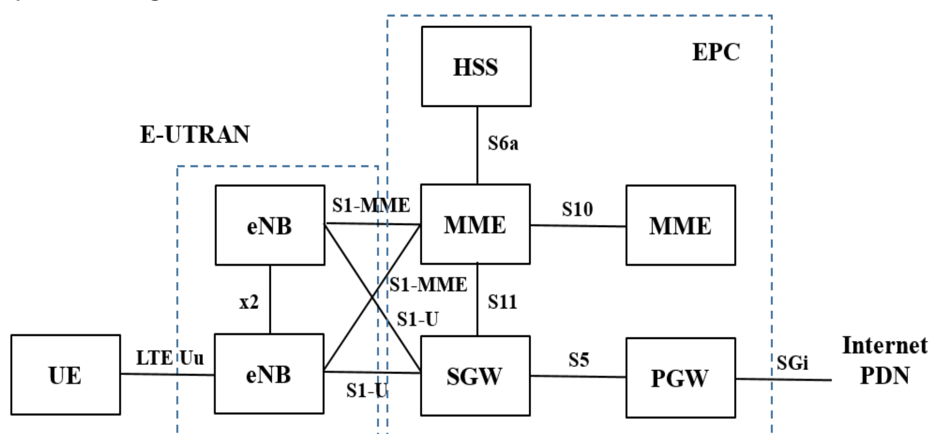


Figure 1.1: Architecture of the 4G network. Image based from [7].

As shown in the previous figure, the 4G architecture is composed of two main blocks:

- **Evolved UMTS Terrestrial Radio Access Network (E-UTRAN):** it is the Radio Network Subsystem and forms the access network. It is composed of the eNodeB's (eNB), which are responsible for communicating with the User Equipment (UE) over the air. Also the eNB manages some radio resources management functions.

- **Evolved Packet Core (EPC):** it is the core of the network, which optimizes traffic delivery. It is composed of a set of functions:

  - Packet Data Network Gateway (PGW): it is the EPC gateway to the Internet or other external Packet Data Networks (PDNs). A PDN is a generic description for a network that provides data services. In packet switching the message is broken into smaller pieces (packets) that are sent independently over an optimal route for each one. Those packets are reassembled when they reach the destination. In order to provide internet connectivity, each UE is assigned a default PGW when it is first connected to the EPC. Furthermore, the PGW is responsible for assigning IP addresses to the UE.

  - Serving Gateway (SGW): it is responsible for only managing tunnels to interconnect the eNB and the PGW. The creation and modification of those tunnels is controlled by the Mobile Management Entity (MME). A SGW is assigned to each UE after authentication.

  - Mobile Management Entity (MME): it controls the high-level operation of the UE by being responsible for the signaling between eNB's and the EPC. As mentioned before, it is tasked to create the tunnels that interconnect the eNB and the PGW. Moreover, MME modules are responsible for tasks such as: authentication, handover support, NAS mobility management, interworking with other radio networks and SMS and voice support. MME modules are grouped in pools and can serve several eNB's simultaneously. Each UE is assigned a single MME, known as serving MME.

  - Home Subscriber Server (HSS): it is responsible for storing subscriber's information. The most important user parameters are:

    - The user's International Mobile Subscriber Identity (IMSI): a unique identifier of each subscriber also stored in the Subscriber Identity Module (SIM) card of the user. It includes the Mobile Country Code (MCC) and the Mobile Network Code (MNC).

    - Authentication information to authenticate the subscribe and generate keys during session establishment.

    - The Mobile Subscriber Integrated Services Digital Network (MSISDN): the telephone number, used generally for circuit-switched services.

    - The Access Point Names (APN) the subscriber can use.

1.1.2. 5G

5G technology represents a complete change on the foundations of wireless communications. Unlike the previous upgrades, 5G defines a new network architecture, aiming to provide service delivery on a global scale, not only worrying about bandwidth and speed. 5G networks started to be standardized from Release 14 [8] and Release 15 [9] of 3GPP around three main service pillars:

- **Enhanced Mobile Broadband (eMBB):** This requires a big capacity enhancement, in order to be able to manage multimedia contents, augmented reality, virtual reality, video 360, etc.
- **Massive Machine Type Communication (mMTC):** A massive connectivity between devices is necessary to be able to manage sensors and actuators that compose the Internet of Things (IoT).
- **Ultra Reliable Low Latency (URLLC):** To be able to manage industrial IoT, vehicle-to-vehicle connections, vehicle-to-infrastructure connectivity and real-time applications, an Ultra-high reliability and a Low Latency is required.

This service differentiation allows applications with distinct QoS and performance requirements, such as: cloud virtual and augmented reality, connected automotive, smart manufacturing, connected energy or wireless e-health, to coexist with each other while meeting the user and services expectations.

The core of the 5G networks described in Release 15 [9] of 3GPP have been defined to meet the following characteristics:

- Support a service-based architecture for modularized network services.
- Consistent user experience between 3GPP and non-3GPP access networks.
- Harmonization of identity, authentication, QoS, policy and charging paradigms.
- Adaption to cloud native and web scale technologies.
- Edge Computing and nomadic/fixed access. Bringing computing closer to the point would reduce latency.
- M2M communication services that could bring low latency connectivity to devices, such as self-driving cars.

Also, in [6], the authors list the main characteristics, as well as the main advantages and challenges of 5G systems. These three service pillars can be observed in Figure 1.2.

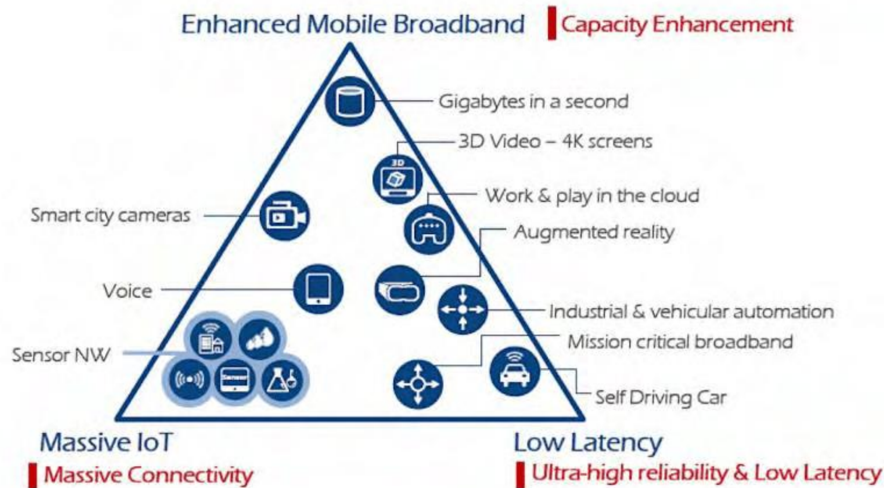


Figure 1.2: 5G Service Pillars. Image taken from [10].

The 5G network is composed of three different functional blocks: The UE, the 5G Core (5GC) and the 5G Access Network (5G-AN). Moreover, the network also has different deployment modes that will be explained later in this work.

1.1.2.1. 5G Core Functions

From Release 14 [11] of 3GPP, the 5GC architecture has two different approaches: the Point-to-Point architecture and the Service Based Architecture (SBA). Figure 1.3 shows the 5GC Point-to-Point architecture, which displays the traditional structure of interconnecting the Core functions. On the other hand, in the SBA approach, each Network Function (NF) offers one or more services to the other NF in the network. Moreover, the NF are self-contained, independent and reusable. Also, they are exposed via a South-Bound Interface (SBI) through an API as shown in Figure 1.4.

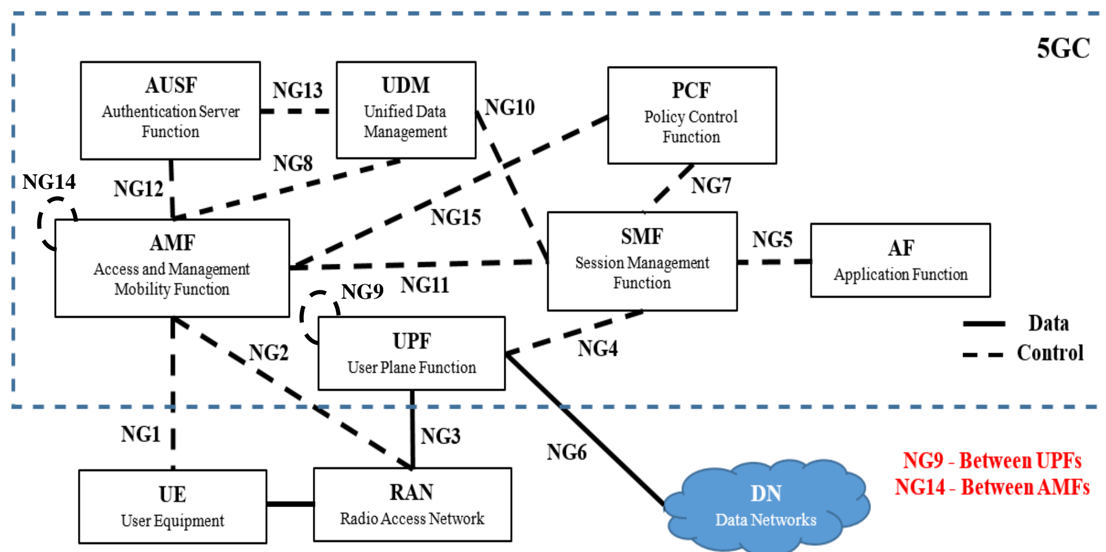


Figure 1.3: Point-to-point 5GC architecture. Image based from [12].

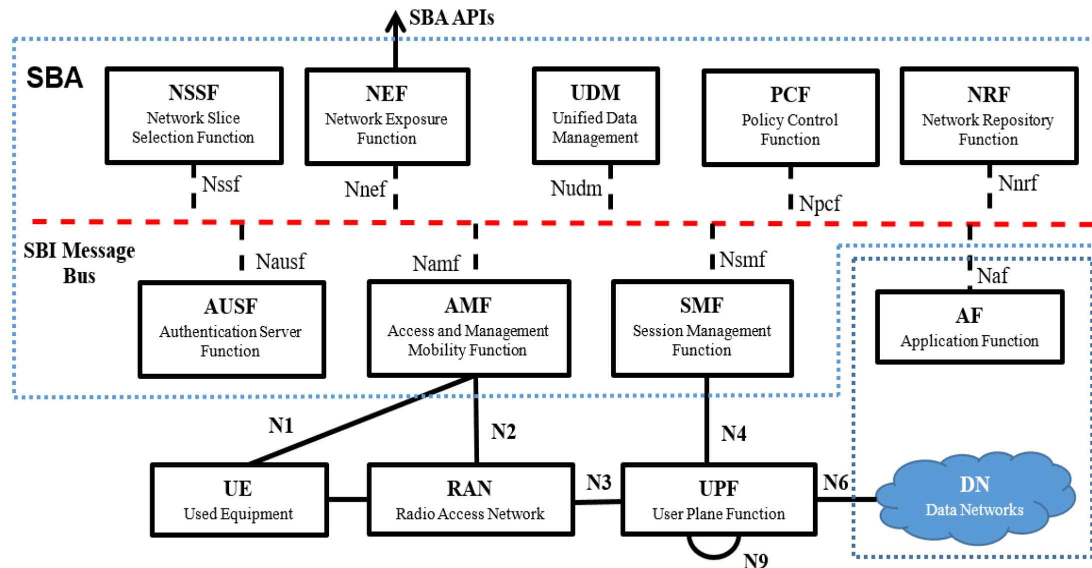


Figure 1.4: SBA 5GC Architecture. Image based from [12].

The main difference between both architectures is the introduction of the SBI, where all the 5GC modules get connected, giving the 5GC more flexibility.

The UE is connected to the 5GC and to the Internet or other Data Networks (DN), over the 5G New Radio Access Network (RAN). As shown in the previous figures, the 5GC architecture is composed of many modules, which can be considered an evolution of the 4G functions:

- **Access and Mobility Management Function (AMF):** Handles the connection between the UE and the access to the network core. The AMF supports encrypted signaling connections. This allows the UE to register, authenticate and move between different radio cells.
- **User Plane Function (UPF):** Manages the forwarding data traffic of the UE. It connects with external networks and acts as a stable IP reference for them. It is also responsible for the QoS, the buffer handling, the packet classification and the packet inspection.
- **Policy Control Function (PCF):** Provides a unified policy of rules and the framework to enforce them and govern the network behavior.
- **Session Management Control Function (SMF):** Is responsible for the establishment, the modification and the release of UE sessions, as well as the assignment of the IP addresses for each session. It also manages the UPF and interacts with the PCF to support charges.
- **Authentication Server Function (AUSF):** Performs authentication processes with the UE towards the network.
- **Unified Data Management (UDM):** Stores the long term security credentials used in authentication as well as the subscription information. The UDM also keeps track of the AMF and SMF serving each UE (in case there is more than one in the network).

- **Application Function (AF):** Requests dynamic policies and/or charging control.

1.1.2.2. 5G Deployment modes

For the 5G network architecture, 3GPP describes an LTE access support. Additionally, there are two different combinations of LTE and the 5G New Radio (NR) access: Non Stand Alone (NSA) and Stand Alone (SA) architectures.

The SA setup contains only one Radio Access Technology (RAT), either LTE radio or 5G Next Generation NodeB (gNB). Both control and user planes go through the same RAN element. As shown in Figure 1.5, there are three different deployment options:

- **Option 1:** EPC and 4G eNB.
- **Option 2:** 5GC and 5G gNB.
- **Option 5:** 5GC and 4G ng-eNB.

The NSA setup combines multiple RATs. The control plane goes through a master node whereas the data plane is split across the master node and a secondary node. As shown in Figure 1.5 there are three different deployment options:

- **Option 3:** EPC and 4G eNB master node plus 5G gNB secondary node.
- **Option 4:** 5GC and 5G gNB master node plus 4G ng-eNB secondary node.
- **Option 7:** 5GC and 4G ng-eNB master node plus 5G gNB secondary node.

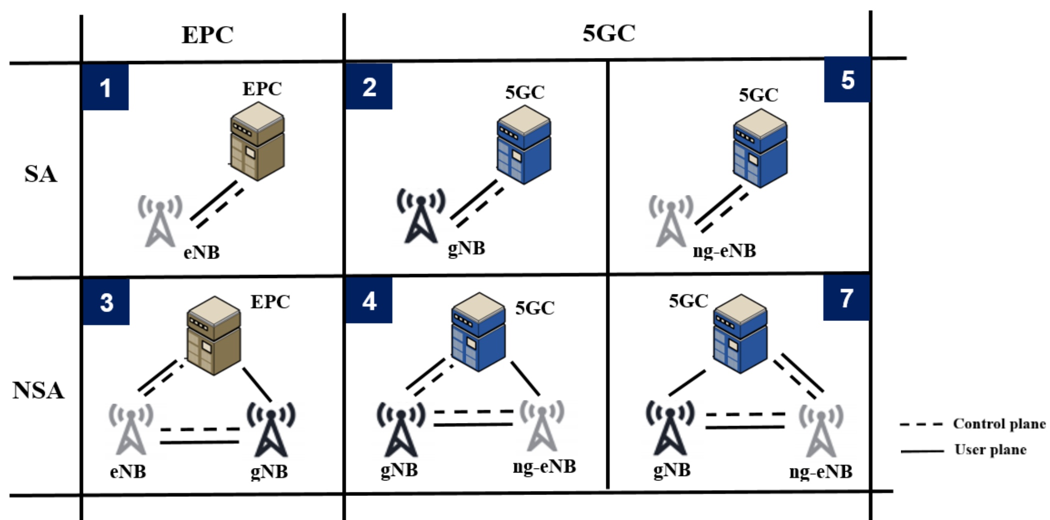


Figure 1.5: SA and NSA deployment modes. Image based from [13].

The NSA architecture was designed to maximize the reuse of the 4G architecture to make the transition to 5G progressively. Table 1.1 summarizes the different options, remarking the 3GPP term and release of each case.

Table 1.1: SA and NSA architectures comparison.

	Core Network	Principal RAT	Secondary RAT	3GPP term	3GPP release
Option 1	EPC	eNB	-	LTE	Rel. 8
Option 3	EPC	eNB	gNB	EN-DC	Rel. 15 Dec 2017
Option 2	5GC	gNB	-	NR	Rel. 15 June 2018
Option 4	5GC	gNB	ng-eNB	NE-DC	Rel. 15 March 2019
Option 5	5GC	ng-eNB	-	eLTE	Rel. 15 June 2018
Option 7	5GC	ng-eNB	gNB	NGEN-DC	Rel. 15 March 2019

1.1.3. Comparison between 4G and 5G

Taking into consideration what has been explained and the comparison of both technologies in [6], the following conclusions have been achieved:

- In contrast with 4G, 5G is designed to support a diversity of applications such as augmented reality, IoT, self-driving cars and immersive gaming. It offers the faculty to handle loads of many different types of traffic and a massive number of devices connected.
- Compared to 4G networks, 5G decreases the latency in less than five milliseconds. Also, the mobility speed range of 5G against 4G increases and it is more energy efficient.
- 5G technology expands its frequency domain to a wider range than 4G.

Table 1.2 shows a more detailed comparison between features of the 4G and 5G technologies.

Table 1.2: Comparison between 4G and 5G technologies.

Features	4G	5G
Start From	2010	2016
Ultra low latency	10 ms to 100 ms	1 ms to 4 ms
Ultra high data rate	1.2 Gbps	10 - 100 Gbps
Massive connectivity	2,000 devices/km^2	1,000,000 devices/km^2
Ultra high mobility	350 km/h	500 km/h
Ultra low energy consumption	90% more than 5G	Up to 10 year battery life for low power MTC
Frequency domain	2 – 8 GHz	3 – 300 GHz
Handover	Horizontal and Vertical	Horizontal and Vertical
Core network	All IP networks	Flatter IP network, 5G network interfacing
Multiple Access	OFDMA	OFDMA, BDMA

To sum up, due to the big differences with regards to throughput, architecture and latency, the integration of 4G and 5G is almost impossible. So, there must be a progressive transition from the current 4G to 5G.

Most of the services that emerge with 5G networks demand very low latency that arise the need and demand of Edge and Multi-access Edge Computing (MEC). Authors of [14] and [15] show why MEC is a key solution to enable operators to open their networks to new services and IT ecosystems. Operators would take advantage of the edge-cloud benefits in their networks and systems. When MEC is located in proximity from the end user, it provides extremely low latency and high bandwidth.

1.2. Network Virtualization

Due to the recent exponential growth in the number of users and their demands and requirements, cellular network technologies have evolved greatly following these needs, trying to meet these necessities. 5G technology will have to confront many challenges related to being able to enable multiple use cases and multi-vendor integration. Another goal of 5G networks is the ability to adapt in real time to dynamic changes in traffic and complexity of the network. This will translate into a more flexible network regarding service demands. To do so, many enablers have been proposed. One of them is network slicing, which allows the creation of different and separate logical networks over the same physical infrastructure. Furthermore, SDN and NFV represent an essential part. These last two are explained with more detail as background of this work.

1.2.1. Software Defined Network

Commonly, mobile networks are composed of two main planes, the control plane and the data (also called forwarding or user) plane, since the introduction of the Control User Plane Separation (CUPS) concept in Release 14 [11] of 3GPP.

The control plane manages the necessary operations to assure connectivity in the network. Some of those operations could: be the identification of the overall network topology, the discovery of the shortest path between two nodes and to make decisions about the allocation of the traffic, to name a few. The data plane contains the messages generated by the users of the network, which should be transferred according to a defined policy.

When traditional networks scale up, this approach of operation becomes very complex to manage. SDN technology was developed to grant the control plane the flexibility needed to support the traffic forwarding requirements of the data plane [9]. SDN is a dynamic architecture that guarantees an automation of the network. To do so, SDN is conceived around four aspects:

- To separate the network control plane from the forwarding plane. This concept is also taken in 4G and 5G networks through the Control User Plane Separation concept described in 3GPP Release 14 [11]. This allows separating the Core Network (CN) functionalities into a control

plane and a user plane, which can be placed closer to the users and will be an essential enabler for MEC computing. Moreover, it allows the distribution and deployment of network functions on different nodes on demand.

- To be able to set up new connections in a fast and agile procedure.
- To provide the ability to respond rapidly to changes in the network conditions.
- To make the connectivity services programmable using standardized Southbound APIs [16].

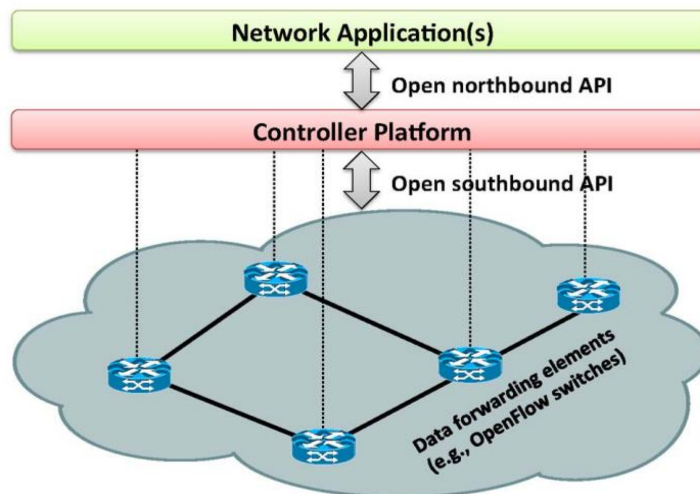


Figure 1.6: Simplified SDN architecture. Image taken from [17].

By creating a physical separation between the network control plane and the forwarding plane, the network intelligence is removed from the hardware (forwarding equipment) and it is implemented into a logical instance called SDN controller. This translates into less complex elements in the forwarding plane. SDN controllers are now directly programmed through applications. The centralization of the intelligence into the control plane, provides a global view of the entire network. Also, this makes the management of the network highly agile and adaptable. Figure 1.6 shows a simplified representation of what has been explained. These concepts have been extended from the wired to the wireless domain and the same vision can be found in Wi-Fi and cellular networks by means of the deployment of Software-Defined RAN (SD-RAN) controllers such as 5G-EmPOWER [18], Odin [19], FlexRAN [20] or even one step beyond, through the vision promoted by the O-RAN Alliance, discussed in [21].

1.2.2. Network Function Virtualization

As for SDN, the need to deliver network services faster and to replace the physical network devices performing such services, to one or more software programs executing network functions, while running on generic hardware has caused the appearance of the NFV. NFV implements a new way to abstract the

network functions. As the authors discussed in [22] and [23], NFV enables network functions to be created, managed, distributed and controlled by software in an agile way.

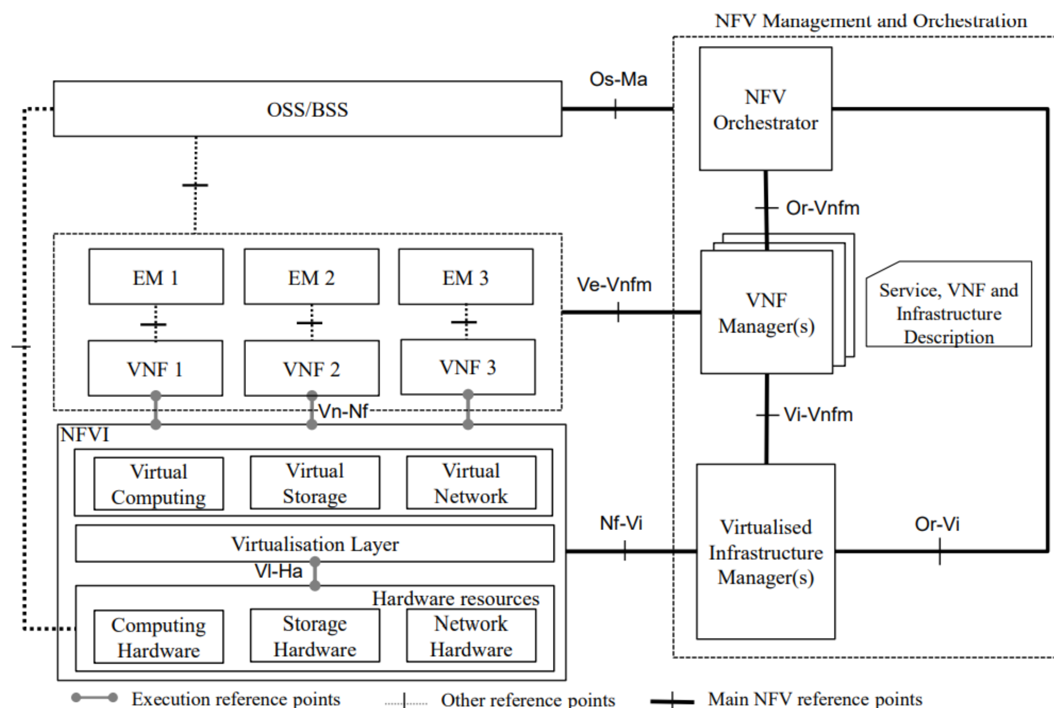


Figure 1.7: ETSI NFV ref. architectural framework. Image taken from [23].

Figure 1.7 displays, the main components of the NFV architecture according to the ETSI NFV MANO reference architecture [23], which include:

- **Virtual Network Functions (VNFs):** elements that provide part (or all) of the network services. It can be composed of many components known as VNF Components (VNFCs). Also, a VNF can be used in one or multiple virtual machines.

- **NFV Infrastructure (NFVI):** aggregation of physical, virtual and software necessary resources, to build the scenario where the VNFs are deployed. Resources such as computing, storage or networking components are virtualized employing a hypervisor or a container system.

- **NFV Management and Orchestration (MANO):** it is the module that performs the management and orchestration of both the infrastructure and all the VNFs that are deployed. It consists of (i) the Virtualized Infrastructure Manager (VIM), which allocates and handles the resources dedicated to each VNF; (ii) the VNF Manager (VNFM), which manages the lifecycle, the configuration, the performance and the security of the VNFs and (iii) the NFV Orchestrator (NFVO), which coordinates all the VNFMs and VIMs, as there can be more than one of each coexisting in the same system at the same time, to ensure proper operation.

1.2.3. NFV and SDN comparison

Both NFV and SDN technologies are software-oriented solutions that are complementary but can be implemented separately. The NFV target is to facilitate flexible and scalable deployments of network functions in any data center. In contrast with that, the SDN target is the control of the packet transmission. It provides functionalities handling enormous quantities of traffic. Additionally, SDN separates the forwarding and the control plane, while NFV decouples functions from hardware. Besides that, a SDN controller can be executed as a VNF. Also, SDN makes the communication between VNFs more flexible and can concatenate VNFs in an automated way. On the other hand, NFV simplifies the management of SDN (due to the generalization of the network).

In conclusion, both technologies are key enablers for the flexible deployment of 5G networks as well as for the recent MEC-enabled systems, especially as communication networks continue becoming more agile and highly distributed systems in the road to cloud native networks.

CHAPTER 2. STATE OF THE ART OF VIRTUALIZED CELLULAR NETWORKS

2.1. Cloud native architecture

The cloud native architecture is a structure based on a series of patterns [24] that are constituted for applications and services, and specifically built for running in the cloud. Micro services are the core of this kind of architecture. Each one of them is created to execute a particular function (implementing, communicating, or running processes). Micro services are often packaged into containers. A container is a runnable instance of an image, that is a lightweight, standalone and executable package of software. This package includes everything needed to run an application: code, settings, system tools, system libraries, etc.

An end-to-end application stack usually includes different technologies such as a web server (e.g., using node.js), a database (e.g., MongoDB), a messaging system (e.g., Kafka) and an orchestration tool (e.g., Jenkins). All these components must have compatibility not only with the underlying OS but also with the libraries and dependencies of the OS. Some incompatibility problems might occur when one service requires one version of a dependent library whereas another service requires another one. Also, each time an application changes, there might be a need to do a modification (i.e., upgrade) on the libraries or the dependencies and the precautions mentioned previously, must be taken into consideration. These compatibility issues can be observed in Figure 2.1.

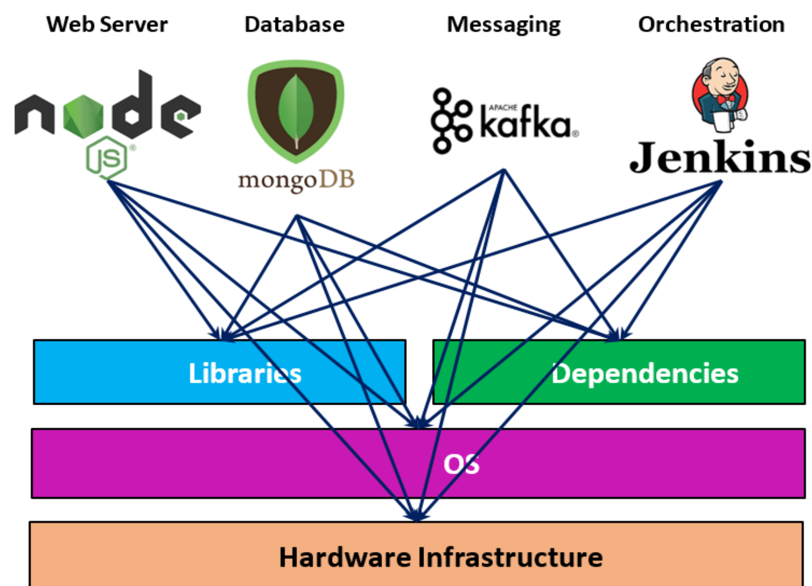


Figure 2.1: Compatibility issues matrix in a traditional deployment.

Every time a developer tries to set up the environment, a very long and tedious procedure has to be followed. They have to follow a large set of instructions and make sure they are using the right Operating System (OS) and the right versions of each of these components. This makes the launch

of the environment very difficult for new developers. In this context, containers allow the modification of these components without affecting the other ones and even the modification of the underlying operating system as required. Furthermore, in a distributed scenario where one or more of the modules have any kind of problem, such as failing, the rest of the modules would keep working properly. Once those failing modules start working properly, the connection will resume.

To be able to manage these container images, a cloud project infrastructure is needed. There are many options, such as: LXD [25], Windows Containers [26], Docker [27], etc. This particular project leverages Docker as a containerization tool. Moreover, besides the design and crafting of the applications, cloud native requires orchestration tools that enable the deployment of applications and containers. Despite the existence of several frameworks for this task (e.g., Docker swarm), this project studies the use of Kubernetes (K8s) that will be explained in the following sections.

2.2. Virtualization Frameworks

2.2.1. Docker

Docker is an open-source project that automates the deployment of applications within software containers. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably decoupled from the underlying host infrastructure. By design, a container is immutable: the code cannot be modified after being run. Making changes in a containerized application, requires building a new container image that includes the changes. Once the modification is done, then the container has to be started from the updated image. A container image is a lightweight, standalone, executable package of software that includes everything needed to run an application.

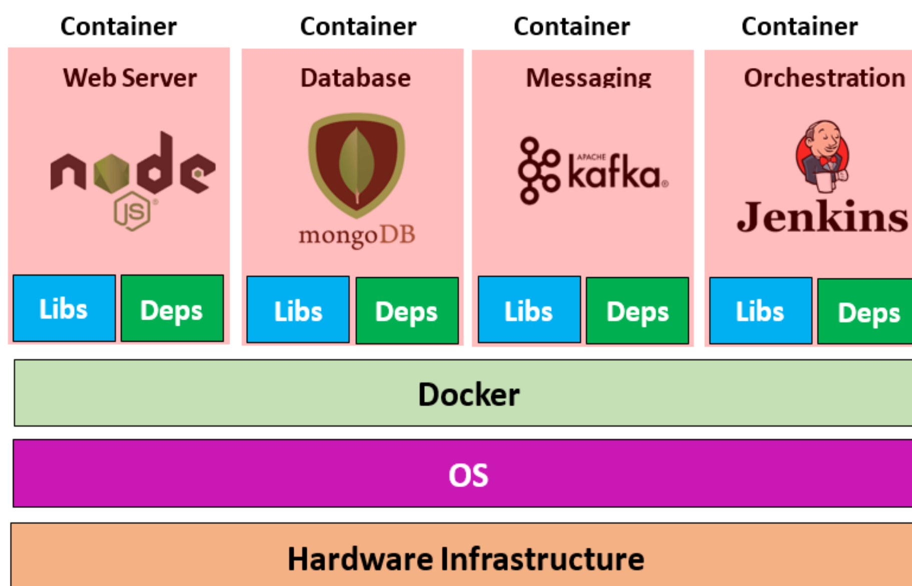


Figure 2.2: Overview of running Docker and some containers.

This containerization provides an additional layer of abstraction and an application virtualization automation across multiple operating systems. Figure 2.2 exhibits the overview of Docker and some containers running on top of the OS. Notice that the tools exposed in the figure are mere examples and other ones with similar capabilities could perform the same operations.

2.2.2. Kubernetes

Kubernetes (K8s) [28] is a portable, extensible and open-source container orchestrator. It is a platform that manages containerized workloads and services that facilitate the configuration and the automation. This lets the user run distributed systems in a resilient way, with scalability and failover for all the applications.

2.2.2.1. Kubernetes Cluster

Kubernetes is deployed in the form of a cluster, which consists of a set of worker machines, called nodes, that run containerized applications and are connected to work as a single unit. A K8s cluster is formed out of two types of resources: (i) the Master node, which manages and coordinates the activity in the cluster and (ii) the worker nodes, where the applications run. To be considered a cluster, there must be at least a master and a worker node connected. The worker nodes can be Virtual Machines (VM) or physical devices that are used as worker machines in the cluster.

The different node(s) host the Pods that contain the application workload. By the official Kubernetes definition, a Pod is "*the smallest deployable unit of computing that you can create and manage in Kubernetes*". Additionally, a Pod is a group of one or more containers, with shared storage and network resources.

Figure 2.3 shows the diagram of a Kubernetes cluster with all the components.

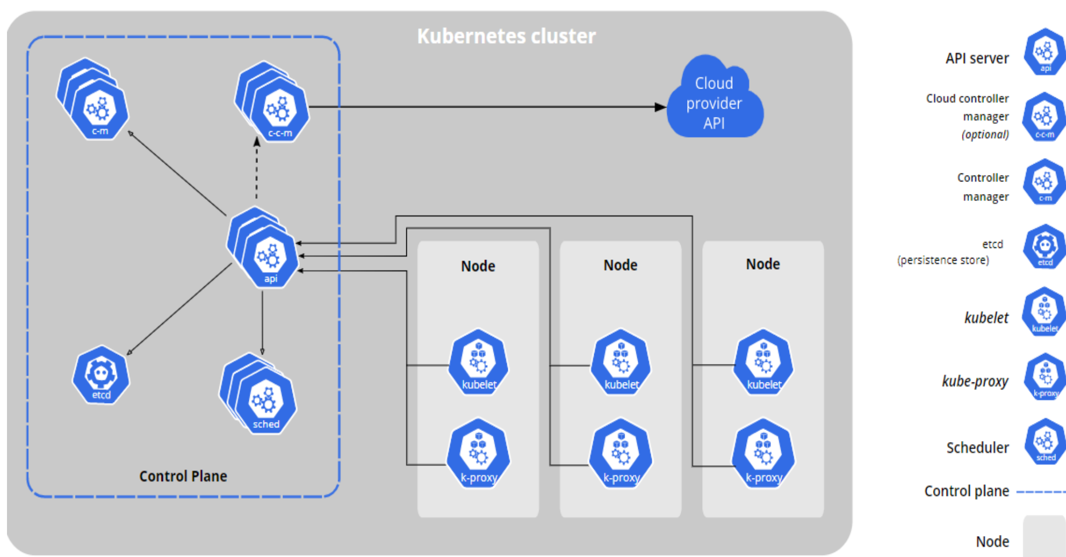


Figure 2.3: Kubernetes cluster diagram. Image taken from [28].

It is important to point out that the control plane manages all the worker nodes in the cluster and all the pods inside them. Usually, the control plane runs across multiple computers and the cluster runs multiple nodes, providing fault-tolerance and high availability.

2.2.2.2. Cluster Networking

Networking is a central and powerful part of Kubernetes, but usually there are four different networking problems that arise when the cluster is created:

- The Container-to-Container communication, that is solved by Pods and localhost communications.
- The Pod-to-Pod communications, that is solved by networks and services.
- The Pod-to-Service communications, that is covered by services.
- The External-to-Service communications, that is covered by services.

A service is a way to expose an application running on a set of Pods as a network service. Kubernetes gives each service and Pod their own IP address and can load-balance across them. The difference in the address assignment is that Pods IP's are dynamic and each time a Pod, for any reason, restarts, a new address is assigned. On the other hand, even if the Pod fails and restarts, the service that exposes the application keeps the same IP.

Kubernetes requires that two applications do not try to use the same ports or a conflict will arise. Coordinating ports across multiple machines and developers is a very difficult task, even more so when the cluster scales up. Instead of dynamically create port allocation, Kubernetes imposes the following fundamental requirements:

- Pods on a node can communicate with all the Pods on all nodes in the cluster without the need of a Network Address Translation (NAT).
- Agents on a node (e.g. system daemons, kubelet) can communicate with all Pods on that node.

This means that it is not required to create links between Pods and that the effort in mapping container ports to the host ports is minimal. From the perspective of load balancing, naming, configuration and port allocation, this helps the user to treat each Pod as a physical host.

2.2.2.3. Container Network Interface

The Container Network Interface (CNI) is a container networking specification [29] proposed by CoreOS and adopted by container runtimes such as Apache

Mesos [30], Cloud Foundry [31] and Kubernetes. The CNI is a set of standards that define how programs should be developed to solve networking challenges in a container runtime environment. The programs are going to be referred to as plugins. CNI was created to be a simple interconnection between the container runtime and the network plugins as shown in Figure 2.4. It defines how the plugin should be developed and how the container runtime should invoke them. Also, it defines a set of responsibilities for container runtimes and plugins:

- For container runtimes, the CNI specifies that they are responsible for: (i) creating a network namespace for each container; (ii) identifying the network the container must attach to; (iii) invoking the plugin when a container is created and also when it is deleted; and (iv) defining how to configure a network plugin in the container runtime environment using a JSON file.
- For plugins, the CNI specifies that they must: (i) support command line arguments such as ADD/DEL/CHECK; (ii) accept parameters such as container id, network namespace, etc. (iii) take care of assigning IP addresses to the PODs and any associated routes required for the containers to reach other containers in the network; and (iv) return results in a specific format.

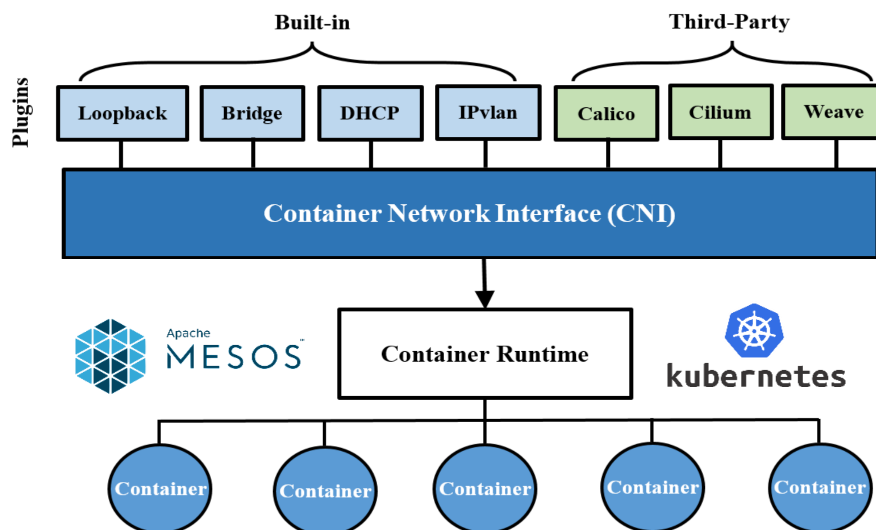


Figure 2.4: CNI connection overview. Image based from [32].

CNI comes with a set of supported plugins, such as the ones in Figure 2.4. All the container runtimes mentioned before implement CNI standards so any of them can work with any of these supported plugins. There are also plugins created by third parties organizations such as Contiv Networking [33], Project Calico [34] and Weave [35]. In this Master Thesis Calico is going to be used because of its very detailed and useful documentation and due to, it is trusted by companies all over the world such as: L3Harris, Discover or AT&T.

Docker does not implement CNI, it has its own set of standards known as Container Network Model (CNM). But, CNI can be used with Docker by creating a Docker container without any network configuration and then manually invoking

the bridge plugin yourself. By contrast, when K8s creates Docker containers, it creates them in a none network and then invokes the configuration CNI plugins who will take care of the rest of the configuration.

2.3. Open-source 5G Frameworks

Open-source frameworks are software for which the original source code has been made freely available and may be redistributed, modified or enhanced according to the user requirements. In this section, some open-source projects providing RAN and CN implementations are discussed.

2.3.1. Radio Access Network

The RAN is an essential part of a mobile telecommunication system. Conceptually, it is located between a remotely controlled machine and its CN to provide connectivity. Four main open-source RAN initiatives widely used in the research community are described below.

2.3.1.1. Open Air Interface 5G Radio Access Network

The scope of the Open Air Interface (OAI) 5G RAN project [36] is to build the 5G protocol stack for both the gNB and UE, allowing an end-to-end deployment of a 5G network. The OAI RAN source code can be found at [37]

Their first target is to develop a 5G Non-Stand Alone RAN software and enable connection and traffic flow through an NSA-capable 5G commercial UE. The OAI 5G stack supports: (i) NSA gNB software stack; (ii) SA gNB software stack; (iii) 5G UE software stack; (iv) RAN Intelligent Controller (RIC) interfaces; and (v) a Continuous Integration/Continuous Deployment (CI/CD) framework allowing for testing and data-center deployment of the 5G split architecture.

The OAI 5G RAN project consists of three different phases attending to the availability of NSA and SA connectivity over the course of two years, starting the summer of 2020. Figure 2.5 displays a high-level view of those phases.

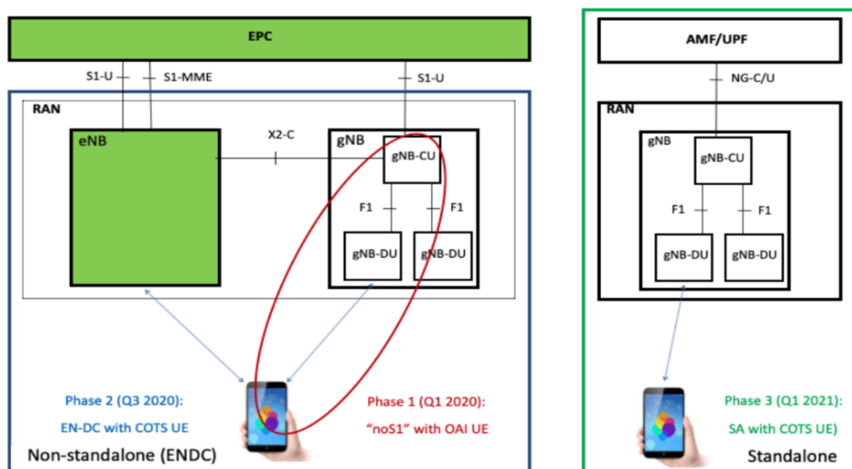


Figure 2.5: OAI 5G RAN project phases. Image taken from [38].

2.3.1.2. srsLTE

The srsLTE project has evolved and has been renamed Software Radio Systems RAN (srsRAN) [39]. srsRAN is a free and open-source 4G and 5G NSA software radio suite that features both the UE and eNB/gNB applications. It can be used with a third-party CN to build a complete end-to-end mobile wireless network. The srsRAN source code can be found at [40].

Currently, srsRAN includes: (i) srsUE, which is a full-stack 4G and 5G NSA UE application with a 5G SA version still under development; (ii) srsENB, which is a full-stack 4G eNB application with a 5G NSA and 5G SA version still under development. This solution is portable with x86, ARM and PowerPC platforms; and (iii) srsEPC, which is a light-weight 4G EPC implementation with MME, HSS and S/P-GW.

The srsGNB is a full-stack software radio gNB solution for 5G NR Standalone, which is still under development and will be commercially available in Q2 2022. All srsRAN software runs in Linux with off-the-shelf compute and radio hardware.

2.3.1.3. free5GRAN

free5GRAN [41] is an open-source 5G RAN stack. It works in SA mode and the current version includes a receiver which decodes Master Information Block (MIB) and System Information Block#1 (SIB1) data. Moreover, free5GRAN acts as a cell scanner and it includes a library which can be reused for further developments. The free5GRAN source code can be found at [42].

As 5G NSA uses 4G cell for attachment, SIB are transmitted on a 4G cell and this receiver cannot decode SIB1 data from 5G NSA mode. However, this receiver should be able to detect 5G NSA cells and decode MIB data.

2.3.1.4. 5G-EmPOWER

5G-EmPOWER [43] is an open-source framework that implements a SDN Platform for 5G RAN. Its flexible architecture provides an open ecosystem where new 5G services can be tested in realistic conditions. Figure 2.6 displays the 5G-EmPOWER system architecture that is composed by the following components.

- The empower-core, which is the core library used to develop the 5G-EmPOWER controller.
- The empower-runtime, which is the Python-based 5G-EmPOWER Controller. This allows network apps to control Wi-Fi APs and LTE eNB's using either a representational state transfer (REST) API or a Python API.
- The empower-enb-agent, which is the 5G-EmPOWER LTE agent library. This agent allows controlling the LTE eNB's using the empower-runtime.

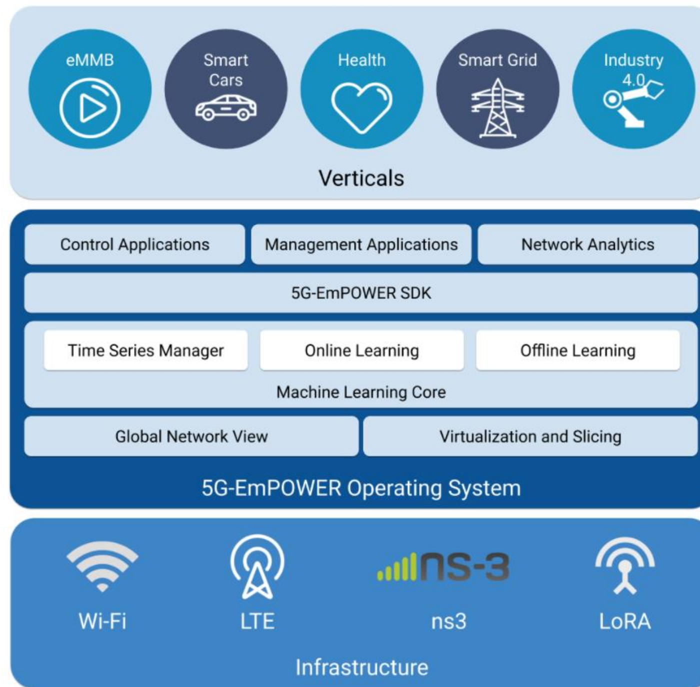


Figure 2.6: 5G-EmPOWER system architecture. Image taken from [44].

2.3.2. Core Network

The CN is an essential part of any IT infrastructure. It is the central element of a network, which provides services and a path to exchange of information to customers who are connected by the access network.

2.3.2.1. Open Air Interface 5G Core Network

The scope of OAI 5G Core Network [45] project is to deliver a 3GPP compliant 5G CN under the OAI Public License. In Figure 2.7 and marked in orange, are the developments in the sphere of the OAI 5G CN project, that covers all parts of the 5G core. The OAI 5G CN source code can be found at [46].

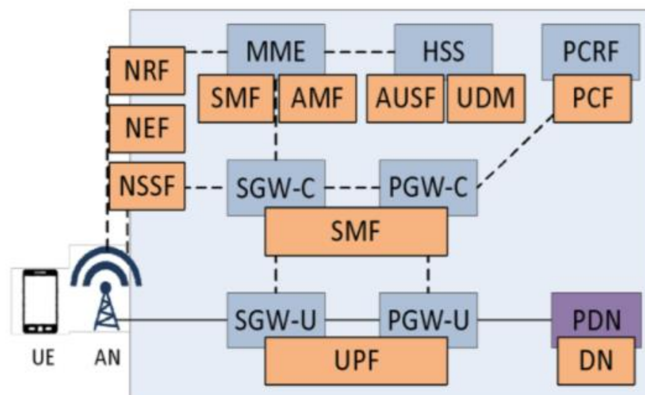


Figure 2.7: OAI 5G CN developments in the 5GC. Image taken from [47].

As the OAI 5G RAN project, the OAI 5G CN project consists of three different phases over the course of two years, starting the summer of 2020:

- **Phase I:** Consists on the basic deployment of AMF, SMF and UPF in Docker containers based on the Ubuntu bionic.
- **Phase II:** Consists on continuous implementation of features and updates for the existing network components (AMF, SMF and UPF) and the addition of extra network components like UDM and AUSF.
- **Phase III:** Consist on a full SA 5GC implementation and the deployment of a framework for a microservices-based architecture. This phase is still under development.

2.3.2.2. Open5GS

Open5GS [48] is a free and open-source initiative that contains a series of software components and network functions that implement the 4G/5G NSA and 5G SA core functions.

The Open5GS 4G/5G NSA core contains the following components: (i) MME; (ii) HSS; (iii) Policy and Charging Rules Function (PCRF); (iv) Serving Gateway Control Plane (SGWC); (v) Serving Gateway User Plane (SGWU); (vi) Packet Gateway Control Plane (PGWC) that is contained in Open5GS SMF; and (vii) Packet Gateway User Plane (PGWU) that is contained in Open5GS UPF.

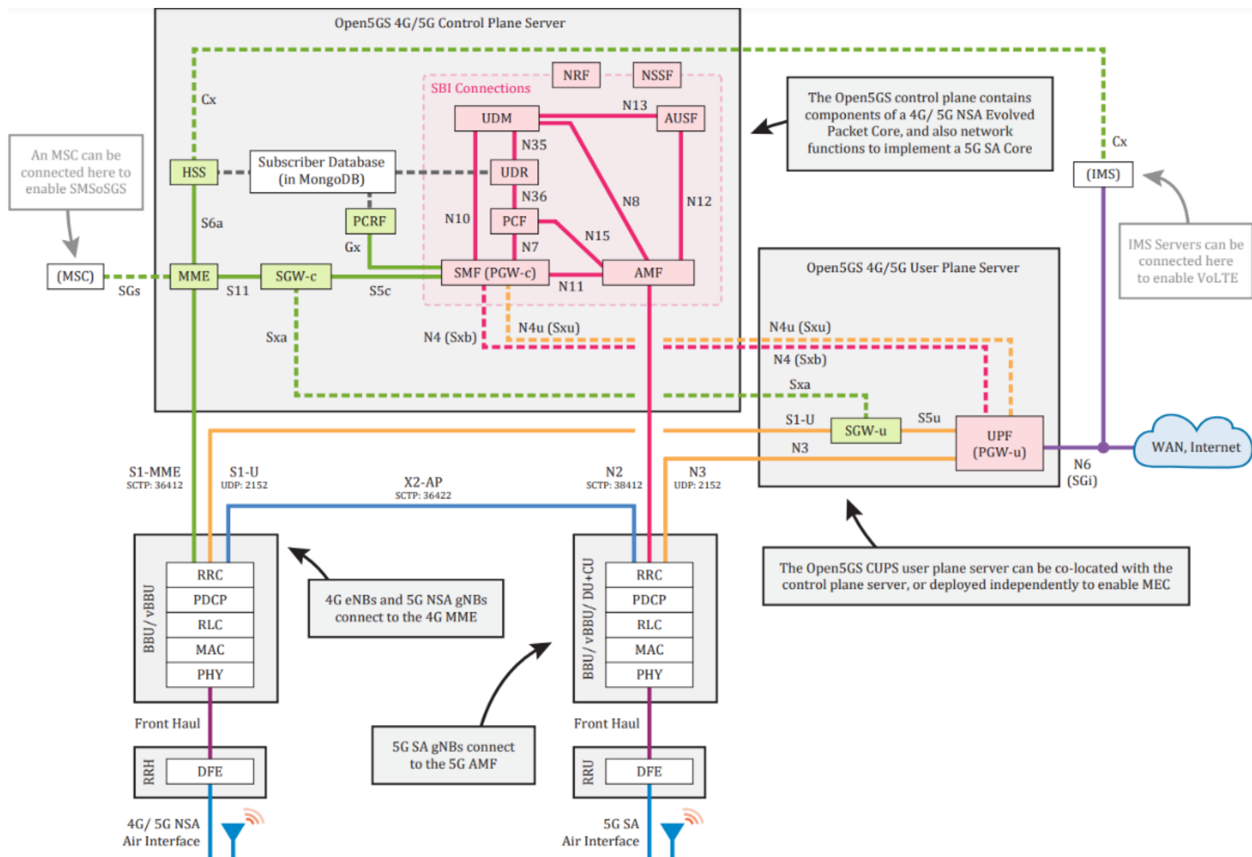


Figure 2.8: Open5GS 4G/5G function representation. Image taken from [49].

The core network has two main planes: the control plane and the user plane. These are physically separated in Open5GS as CUPS is implemented. Figure 2.8 shows the interconnection of the components already mentioned and the separation between the main planes.

In the control plane of this deployment, the MME is the main hub of the core. It primarily manages sessions, mobility, paging and bearers. It links to the HSS, which generates SIM authentication vectors and holds the subscriber profile. Additionally, it links to the SGWC and PGWC, which are the control planes of the gateway servers. Also, all the eNB's are connected to the MME. The last element of the control plane is the PCRF, which sits in-between the PGWC and the HSS, and handles enforcing subscriber policies.

On the other hand, the user plane carries user data packets between the eNB/NSA gNB and the external WAN. There are two core user plane components that are the SGWU and the PGWU. The eNB's/NSA gNB's connect to the SGWU, which connects to the PGWU, which connects to the WAN. The separation of the control and the user planes physically, allows the deployment of multiple user plane servers, while keeping the control functionality centralized. This enables support for MEC use cases.

The 5G SA core contains the following functions: (i) AMF; (ii) SMF; (iii) UPF; (iv) AUSF; (v) NF Repository Function (NRF); (vi) UDM; (vii) Unified Data Repository (UDR); (viii) PCF; Network Slice Selection Function (NSSF); and (ix) Binding Support Function (BSF).

The 5G SA core uses a SBA and SBI to interconnect its modules. As shown in Figure 2.8, in this scenario the control plane functions are configured to register with the NRF, and then, the NRF helps them discover the other core functions. Besides that, the AMF handles connection and mobility management. The UDM, AUSF and UDR carry out similar operations as the 4G HSS, generating SIM authentication vectors and holding the subscriber profile. The session management is managed by the SMF. Also, the NSSF provides a way to select the network slice. Finally, the PCF is used for enforcing subscriber policies.

In this case, the 5G SA core user plane only contains one function, the UPF, which carries user data packets between the gNB and the external WAN. It connects back to the SMF too.

All of the previous components have config files to help the users deploy their own setup. Each config file (with the exception of the SMF and the UPF) contains the component's IP bind addresses/local interface names and the IP addresses/DNS names of the other components it needs to connect to.

2.3.2.3. free5GC

free5GC [50] is an open-source project for 5G mobile core networks. The main goal of this project is to implement the 5GC defined in 3GPP from Release 15 [9]. They have divided this task into three main stages. In the first one they migrated

from the 4G Evolved Packet Core to the 5GC Service-Based Architecture that supported NSA 5G. Then, in the second stage, they implemented the SA 5GC functions and features. And finally, in the last stage, which is still under development, their aim is to develop a fully operational 5GC. The free5GC source code can be found at [51].

2.4. Related work on virtualization tools for building distributed cellular networks

This section describes some papers and projects found in the literature, which covers topics that are very similar or follow a common interest, as the one discussed in this Master Thesis.

The virtualization and cloudification of the mobile network have been a hot research topic in the recent times, especially when it comes to highly distributed networking systems. In this respect, containerization has played a key role in this objective. The authors of [52] define this key step especially in what regards agility as: "*The containerization seems to be the adequate approach to overcome the bottleneck caused by the Serving Gateway (SGW), as it could enable rapid deployment by scaling SGW instances based on workload*". Based on this idea, Figure 2.9 displays the cloudification of mobile network functions the authors defend, using Docker and twelve factors for enabling it. They also build a proof of concept of the scalability of SGW, comparing the performances of Kubernetes and Mesos-Marathon. The proof of concepts showed that a container-based approach is a viable option for achieving elasticity of future mobile networks.

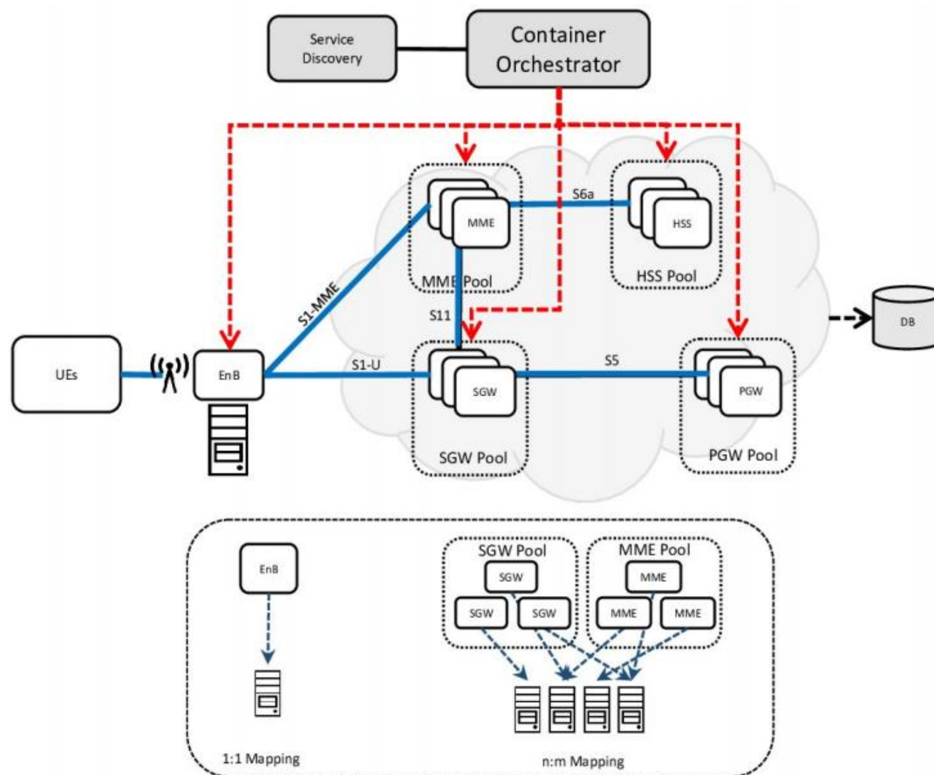


Figure 2.9: Cloudification design of mobile network. Image taken from [52].

Furthermore, in [53] and [54] a review of the current NFV management solutions and a definition of the cloud native toolbox in the context of NFV is presented. It is exposed that NFV technology is a promising attempt to solve the increase in the demand of the vital requirements for bandwidth, latency, and quality of service. Then, authors of [53] introduce an implementation of an open source Cloud Native VNF API design over the top layer, 5GaaS. As an application of the proposed design, the authors define the principles and describe from a standard perspective the feasibility of the prototype. They mentioned that, as part of their future work, it should address the application of Kubernetes orchestration at the VIM layer and to update the CN-VNF framework with better support for the RAN API. In addition to that, in [54], a container-based design of a virtual evolved packet core, based on the OAI software package, is presented. They successfully containerized (and virtualized) the EPC component functions into two separate Docker containers: (i) the control plane container, for virtual home subscriber server and virtual mobility management entity; and (ii) the data plane container, for virtual serving and packet data network gateway. This paper also proposes an algorithm called Specifically Assigned Cores (SAC) to achieve a better utilization of CPU cores. Their preliminary results show that SAC outperforms the default scheme, Randomly Assigned Cores (RAC), in terms of lower CPU load and less packet loss. The authors point out the superiority of SAC over RAC is amplified with the traffic level.

Moreover, two different sets of testbeds for cloud based 5G networks are analyzed in [55] and [56] to shape 5G technology as a flexible, scalable, and demand-oriented network. Paper [55] introduces a novel testbed called 5GIIK, which provides implementation, management, and orchestration across all network domains and different access technologies. 5GIIK is one of the most comprehensive testbeds because it provides additional features and capabilities such as slice provision dynamicity, real-time monitoring. On the other hand, authors of [56] display a 5G mobile network testbed with a virtualized and orchestrated structure using containers. It is focused on integration to artificial intelligence applications. The presented testbed uses open-source technologies to deploy and orchestrate the VNFs to flexibly create various mobile network scenarios.

Furthermore, taking into consideration the virtualization of the RAN, two very interesting papers have been found in the literature. The first one [57], aims to ease the integration of satellite components in forthcoming 5G systems (SatCloudRAN). Authors give special attention to the design, by considering the split and placement of virtualized and non-virtualized functions, while taking into account the characteristics of the transport links between both kinds of functions. They assess how virtualization and softwarization technologies, such as NFV and SDN can deliver part of the satellite gateway functionalities as virtual network functions and can achieve a flexible and programmable control and management of the satellite infrastructure. The authors of the second paper [58] focus their attention to the reduction of the handover in virtualized cloud RAN. They explained that in order to meet the challenging 5G capacity requirements, operators are densifying their cellular networks by deploying additional small cells to cover hot spots, and such an increase in the number and density of cells may result in excessive numbers of handovers. To avoid that, a handover reduction

mechanism is implemented in a Cloud Radio Access Network (CRAN). There, the digital unit of a conventional BS is separated from the radio unit and moved to the cloud for better mobility management and cost saving.

Due to the importance of virtualized infrastructures on the achievement of the requirements of 5G networks and beyond, many research groups and projects in the literature focus their efforts, not only in the advancement of the containerized paradigm, but also to analyze the security and robustness of the available frameworks. In [59], the importance of a secure framework for virtualized networks is pointed out. They indicate the open research issues and future research directions of 5G security and trust in the context of virtualized networking and SDN. A framework of security and trust focusing on solving 5G network security issues is proposed.

Finally, the Open-VERSO [60] project aims to achieve a generic hardware based platform, which will allow advanced networks to get deployed on demand. The platform is a computing environment designed to allow the hardware to operate on real time while being orchestrated from the cloud. It was created to demonstrate and evaluate the viability of an infrastructure based on the “Open RAN” concept and its integration in the cloud. Furthermore, it was conceived to demonstrate the viability and performance of key technologies for the evolution of networks beyond 5G. It has demonstrated the viability and performance of key technologies for the evolution of networks beyond 5G.

CHAPTER 3. RESEARCH WORK

3.1. Introduction

For the development of this Master Thesis the srsLTE and Open5GS solutions have been chosen to deploy the RAN and the core of our network respectively. Those open-source initiatives have been selected due to the features they have available and offer, that cover all the essentials of this deployment. Also, both solutions have a very active community behind them and a very updated, clear and detailed documentation, so in order to solve any kind of problem, there is plenty of information online.

The final aim of this project is to set up a fully virtualized, containerized and distributed open-source-based network that can be deployed on different nodes, separating RAN and core network functions, or even the modules of the core network itself if required on a cellular network. Based on this deployment, this work aims to analyze the performance of the network and the virtualized infrastructure when varying different network parameters. The road to fulfill this goal has been started with a first phase consisting on achieving connectivity in a baremetal deployment, using only one computer, with the aim of verifying that the basic deployment of the open-source modules behaves as expected. The second stage consisted in the deployment of the network using Kubernetes to automate the deployment and the management of the containers. This deployment has been performed in two different manners: (i) a single worker node cluster containing the RAN and the core network; and (ii) the separation of the core and RAN logic into two different worker nodes. Finally, this setup has been compared with existing projects following the same deployment using Docker containers, without a container orchestration platform [61].

3.2. Baremetal Deployment

This deployment has the most basic setup and it is composed of the next hardware components: (i) two computers with Ubuntu 20.04; (ii) an USRP B210 [62] with USB 3.0 connected to the computer running srsLTE; (iii) a sysmoUSIM [63] and (iv) a HUAWEI LTE USB Stick [64]. Figure 3.1 exhibits the setup connected.

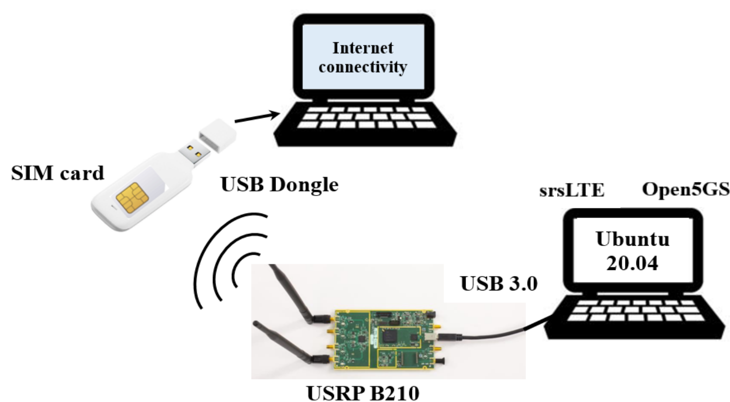


Figure 3.1: Baremetal deployment setup.

3.2.1. Installing software and dependencies

This section includes the installation of: (i) the Ettus driver to manage the USRP; (ii) the srsLTE software and dependencies for the RAN; and (iii) the Open5GS and Open5GS Web-UI software and dependencies, which will form the CN and the UE registration interface respectively.

First of all, it is important to check if the computer recognizes the USRP. To do so, in the command terminal run:

```
uhd_usrp_probe
```

If the console output shows an error or no devices connected, a new command is needed to download Ettus driver, otherwise the connection was successful:

```
./usr/lib/uhd/utils/uhd_images_downloader.py
```

Then, the installation process of srsLTE and Open5Gs with all their dependencies can be followed in ANNEX I.

Open5GS is composed of several modules besides the core functions, such as the Open5GS Web-UI and a MongoDB database. The database will be created with the Open5GS installation command. The Web-UI allows users to register and store the details of the SIM cards. In this Master Thesis, programmable sysmoISIM-SJA2 SIM cards, which has been properly programmed and configure beforehand using the pysim software package [65] with the card data supplied by the vendor. More information about this process can be found in [66]. The srsLTE source code can be found at [67] and the Open5GS source code at [49]

3.2.2. Setup and configuration

Once everything is properly connected, the order of setting srsLTE and Open5GS up is not vital, and it can be interchangeable. In this work, the configuration files of srsLTE are set up in the first place, as follows:

```
~/.config/srsran/enb.conf
```

To get access to that file, and edit the configuration file, the following commands must be executed:

```
sudo -i
nano ~/.config/srsran/enb.conf

[enb]
enb_id = 0x19B
mcc = 001   ----> Put your MCC information
mnc = 03    ----> Put your MNC information
mme_addr = 127.0.1.100
gtp_bind_addr = 127.0.1.1
s1c_bind_addr = 127.0.1.1
n_prb = 50   ---> Number of Physical Resource Blocks (PRB) assigned
```

The MCC and MNC codes are part of the IMSI of the SIM card. The IP addresses shown belong to the local domain, due to the deployment being done in one computer. The number of Physical Resource Blocks (PRB) used can be also configured. This value will change in the performance evaluation shown in the next Chapter.

To configure Open5GS, some modifications have to be done in two different files. The first one is the file "*mme.yaml*", that is stored in:

```
/etc/open5gs/mme.yaml
```

The changes to be done are: (i) set the S1AP IP address; (ii) set the Public Land Mobile Network ID (PLMN), which are the MCC and the MNC; and (iii) set the Tracking Area Code (TAC). Once it has been properly setup, the file should look like the following:

```
mme:
    s1ap:
      - addr: 127.0.1.100 #mme_addr of the enb.conf file of the srsLTE
    gtpc:
      - addr: 127.0.0.2
    gummei:
      plmn_id:
        mcc: 001  ----> Put your MCC information
        mnc: 03   ----> Put your MNC information
      mme_gid: 2
      mme_code: 1
    tai:
      plmn_id:
        mcc: 001  ----> Put your SIM information
        mnc: 03   ----> Put your SIM information
      tac: 7      ----> Put your Tracking Area Code
```

The second file that has to be modified is the "*sgwu.yaml*" to set the GTP-U IP address. It is stored in:

```
/etc/open5gs/sgwu.yaml
```

It should look like the following:

```
sgwu:
    pfcp:
      - addr: 127.0.0.6
    gtpu:
      - addr: 127.0.1.100  #mme_addr of the enb.conf file of the srsLTE
```

After making those modifications in the previous config files, the Open5gs daemons must be restarted by running to make the changes effective:

```
sudo systemctl restart open5gs-mmed
sudo systemctl restart open5gs-sgwud
```

Once both system services are restarted, the subscriber information (the SIM) has to be registered. To do so, the Web-UI can be access at "*http://localhost:3000*" with the following credentials:

```
Username: admin
Password: 1423
```

To add a subscriber, some data is required: (i) the IMSI; (ii) the authentication key (K) and (iii) the derived operator code (OPc). To finish the registration, an APN has to be configured (name and type). Figure 3.2 displays a successful registration.

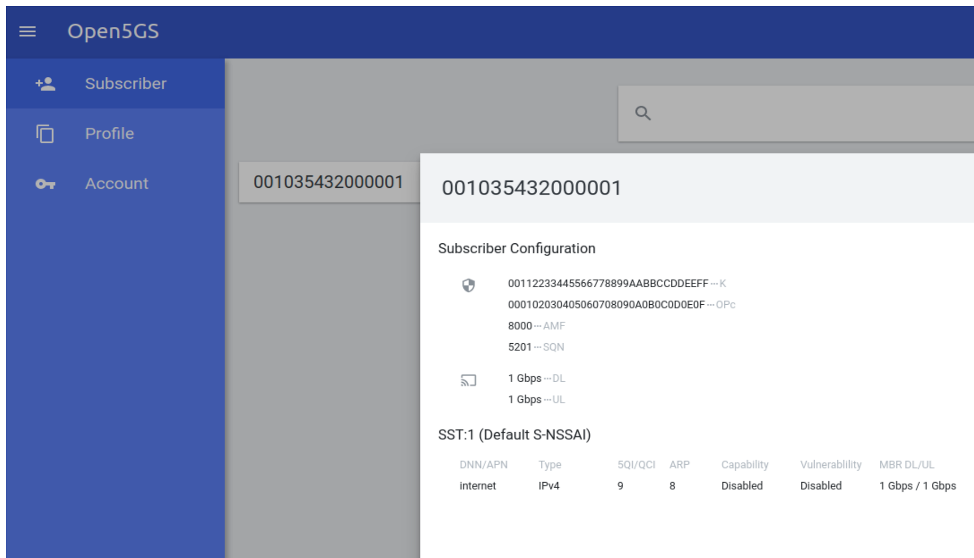


Figure 3.2: Subscriber registered in the Web-UI.

In order of the UE to have WAN connectivity, a route has to be added by enabling forwarding and adding a NAT rule:

```
    ### Enable IPv4/IPv6 Forwarding

    sudo sysctl -w net.ipv4.ip_forward=1
    sudo sysctl -w net.ipv6.conf.all.forwarding=1

    ### Add NAT Rule

    sudo iptables -t nat -A POSTROUTING -s 10.45.0.0/16 ! -o ogstun -
j MASQUERADE
    sudo ip6tables -t nat -A POSTROUTING -s 2001:230:cafe::/48 ! -o
ogstun -j MASQUERADE
```

It is very important to point out that these commands are **not persistent**, so they have to be reintroduced each time the computer is restarted.

3.2.3. Get the deployment running

As soon as all the configurations are finished, it is time to launch srsLTE (only srsLTE because the system services of Open5GS are already running, since they have been restarted in the setup section). To run srsENB, this statement must be typed in the command console:

```
    sudo srsenb
```

To achieve connectivity in the UE, the SIM card can be introduced in a regular off-the-shelf phone or via a USB dongle. Then, a new APN has to be added by introducing the name and the apn (same as name), that corresponds to the one

registered in the Web-UI. Leave the remaining fields by default. At that moment, enable the APN and the UE gets connected, as shown in Figure 3.3.

```
javi@kubernetes-worker3:~$ sudo srsenb
---  Software Radio Systems LTE eNodeB  ---

Reading configuration file /root/.config/srsran/enb.conf...
WARNING: cpu0 scaling governor is not set to performance mode. Realtime processing could be compromis
ed. Consider setting it to performance mode before running the application.

Built in RelWithDebInfo mode using 21.04.0.

Opening 1 channels in RF device=default with args=default
[INFO] [UHD] linux; GNU C++ version 9.3.0; Boost_107100; UHD_3.15.0.0-release
[INFO] [LOGGING] Fastpath logging disabled at runtime.
Opening USRP channels=1, args: type=b200,master_clock_rate=23.04e6
[INFO] [UHD RF] RF UHD Generic instance constructed
[INFO] [B200] Detected Device: B210
[INFO] [B200] Operating over USB 3.
[INFO] [B200] Initialize CODEC control...
[INFO] [B200] Initialize Radio control...
[INFO] [B200] Performing register loopback test...
[INFO] [B200] Register loopback test passed
[INFO] [B200] Performing register loopback test...
[INFO] [B200] Register loopback test passed
[INFO] [B200] Asking for clock rate 23.040000 MHz...
[INFO] [B200] Actually got clock rate 23.040000 MHz.
Setting frequency: DL=2680.0 Mhz, UL=2560.0 MHz for cc_idx=0 nof_prb=50

==== eNodeB started ===
Type <t> to view trace
RACH:  tti=10131, cc=0, preamble=18, offset=1, temp_crnti=0x46
User 0x46 connected
```

Figure 3.3: Baremetal connectivity achieved.

Figure 3.3 depicts that the setup has been properly configured and that User 0x46 has been registered correctly and it can connect to the internet without any problems. Notice that srsRAN uses the Radio Network Temporary Identifier (RNTI) as user identifier, which may change every time the user is disconnected.

If for any reason there is a need to remove either Open5GS or the Web-UI packages, is possible to do it by running the following commands:

```
      sudo apt-get purge open5gs
      sudo apt-get autoremove
      sudo rm -Rf /var/log/open5gs
      curl  -fsSL  https://open5gs.org/open5gs/assets/webui/uninstall  |
sudo -E bash -
```

3.2.4. Problems found

Three main problems have been faced during this deployment, which are not completely documented in the project websites:

- The incorrect selection of the mme_s1ap, sgwu_gtpu and enb_mme_addr IP addresses, due to using different local IP's. This caused that the modules of each software could not find each other. This problem was solved by selecting the same IP address in all of the three.
- Not taking into account that and the addition of the NAT rule and the port forwarding configuration were not persistent, so one day the deployment was working and the next one was not, without any modifications.

- The incorrect creation of the APN (not using the exact same name as the one in adding the subscriber), so as shown in Figure 3.4, the UE was not able to get connected.

```
[INFO] [B200] Detected Device: B210
[INFO] [B200] Operating over USB 3.
[INFO] [B200] Initialize CODEC control...
[INFO] [B200] Initialize Radio control...
[INFO] [B200] Performing register loopback test...
[INFO] [B200] Register loopback test passed
[INFO] [B200] Performing register loopback test...
[INFO] [B200] Register loopback test passed
[INFO] [B200] Asking for clock rate 23.040000 MHz...
[INFO] [B200] Actually got clock rate 23.040000 MHz.
Setting frequency: DL=2680.0 Mhz, UL=2560.0 MHz for cc_idx=0

==== eNodeB started ===
Type <t> to view trace
RACH:  tti=2771, cc=0, preamble=34, offset=0, temp_crnti=0x46
Disconnecting rnti=0x46.
RACH:  tti=311, cc=0, preamble=31, offset=1, temp_crnti=0x47
Disconnecting rnti=0x47.
RACH:  tti=7971, cc=0, preamble=5, offset=1, temp_crnti=0x48
Disconnecting rnti=0x48.
RACH:  tti=5331, cc=0, preamble=14, offset=1, temp_crnti=0x49
Disconnecting rnti=0x49.
RACH:  tti=2691, cc=0, preamble=40, offset=1, temp_crnti=0x4a
Disconnecting rnti=0x4a.
```

Figure 3.4: Failure on connectivity due to wrong APN.

3.3. Kubernetes-based Deployment

As mentioned in the introduction section, two different K8s setups have been deployed: a two nodes (master and a worker) setup and a three nodes (master and two workers) setup. Most of the steps taken to deploy and configure the K8s cluster are very similar, but there are a few differences between both deployments that are going to be explained. This deployment is composed of the following hardware components: (i) three/four computers with Ubuntu 20.04; (ii) an USRP B210 [63] with USB 3.0 connected to the computer; (iii) a HUAWEI LTE USB Stick [64]; (iv) a D-Link DGS 108 Switch [68] and (v) a sysmoUSIM [63]. Figure 3.5 shows both setups connected. In the two nodes deployment (blue) both srsLTE and Open5GS run in the worker node 1. On the other hand, in the three nodes setup (red), even if the Open5GS software still runs in the worker node 1, srsLTE is executed in the worker node 2. Take into consideration that the USRP must be connected to the node that deploys the RAN. The third node contains the master K8s node.

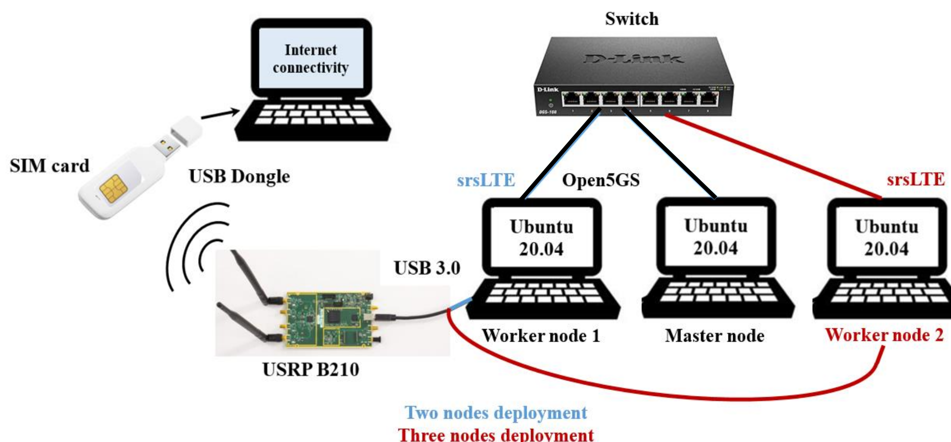


Figure 3.5: Two and three nodes deployments.

3.3.1. Installing software and dependencies

Before starting to set the K8s cluster, some modifications have to be made in all the nodes: (i) setting the computer hostnames; (ii) installing Docker; (iii) disabling swap and enabling IP forwarding; and (iv) installing kubectl, kubelet and kubeadm**.** This process can be followed in the ANNEX II.

3.3.2. Kubernetes cluster setup

For both cases, the setup and the configuration of the cluster is very similar. The only difference is that, instead of adding only one worker node, two worker nodes are added.

Kubernetes uses the CNI to interact with networking providers like Calico. In this case, the Tigera operator is going to be used to install Calico. The operator provides lifecycle management for Calico exposed via the Kubernetes API. The process of setting up the K8s cluster can be followed in ANNEX III.

3.3.3. Design and creation of the Docker images

In order to deploy either the RAN or the CN in the K8s cluster, a Docker image containing the logic of each software is needed. Due to the existing images found in Docker Hub [69] (public repository for Docker images) were too specific, just defined for their own UE, new Docker images were needed.

3.3.3.1. srsLTE Docker image

After researching many Docker images to see how they faced different problems of networking I was having, it was decided to base the Docker image on an existing open-source project [70], that was solving most of them. In order to make the deployment work correctly for our scenario, some modifications and improvements were made:

- Create a new config file to make the deployment configurable for any UE. Without this file, each time a new UE was required, (i) a new Docker image had to be built; or (ii) the corresponding pod had to be accessed to edit the UE configuration and restart the service, making any change less agile and flexible.
- Update the "*launcher.sh*" and the "*conf/enb.conf*" files to include the configurable feature that the config file allows. This lets the user make the deployment for any particular UE.
- Changed the Dockerfile, adding some commands to install all the dependencies needed that were not included before and to load all the files that the modifications mentioned previously required.

This Docker image can be found at [71]. The content of all the files forming the srsLTE Docker image can be found in ANNEX IV: srsLTE.

3.3.3.2. Open5GS Docker image

As in the previews case, after researching many Docker images to see how other initiatives proposed their solutions, the same repository as the one for srsRAN was used as reference for building the Open5GS image [72], because of the quality of his documentation and that the previous srsRAN Docker image was based in his other container. The modifications done are:

- Create a new config file to make the deployment configurable for any UE, as in the srsLTE Docker image.
- This deployment does not support IPv6 yet, so every line in all the modules that included this feature were commented. This is not mandatory for the image to work, but removes all the errors that used to appear on the console.
- While deploying the original container, a problem of shared libraries arose. This problem was caused when more than one module of the 5GC tried to access the same resources at the same time. It was solved by adding the following command in the Dockerfile:

```
RUN sh -c "echo /open5gs/install/lib/x86_64-linux-gnu > /etc/ld.so.conf.d/open5gs.conf
```

This Docker image can be found at [73]. The content of all the files forming the Open5GS Docker image can be found in ANNEX IV: Open5GS.

3.3.4. Design the descriptor file

The descriptor file is in charge of deploying the different pods, services and containers needed for the proper functioning of the project. The file is divided below into manageable parts to be able to explain it in detail.

```
kind: Pod
metadata:
    name: epc                       #Name of the Pod
    labels:
        app: epc
spec:
    containers:
    - name: open5gs                 #Container Nº 1
        image: javipalomares/open5gs:latest
        env:                        #Environmental variables
        - name: mcc
            value: "001"
        - name: mnc
            value: "03"
        - name: tac
            value: "7"
        securityContext:
            privileged: true
    - name: open5gs-webui           #Container Nº 2
        image: snslab/open5gs-webui:latest
    - name: mongodb                 #Container Nº 3
        image: mongo
```

```
        - name: mongo-express          #Container Nº 4
            image: mongo-express
            env:
            - name: ME_CONFIG_MONGODB_SERVER
                value: "localhost"
        nodeSelector:
            IDname: kworker1           #Node label
```

This part defines a pod labeled and called ***epc***. This pod contains a total of four containers:

- The first one is called ***open5gs***. It is based on the last version of the Open5GS Docker image created and explained in the previous section (*javipalomares/open5gs:latest).* After that, the environmental variables to configure it (mcc, mnc and tac) must be instantiated. Then, to give the container root privileges, the *securityContext* is set to true. This gives the container access to all the ports and resources of the system.

- The second container is called ***open5gs-webui***. It is based on the last version of the Docker image taken as reference (*snslab/open5gs-webui:latest*). This container holds the logic of the Open5GS Web-UI, where the subscribers are registered. No more declarations have to be introduced in this container.

- The third and fourth containers are called ***mongodb*** and ***mongo-express*** respectively. They are based on the last versions of the Docker images of mongo and mongo-express. They are used to store the subscribers list and also, mongo-express lets the host view a graphical interface of the content of the mongo database. In the last one, a parameter has to be introduced to configure where the host can access the already mentioned interface, that is set to localhost.

The last parameter that has to be introduced is the node in which this Pod has to be deployed. As mentioned at the beginning of this section, the CN logic is deployed at the worker-node1. To be able to make the assignment, the kubernetes-workers have to be labeled. This procedure will be explained in the next section. The kubernetes-worker1 is labeled kworker1 and through the *nodeSelector* the assignment is achieved. *nodeSelector* is a field that specifies the node in which the pod is chosen to run.

The motive to define the four previous containers in the same Pod is to avoid networking problems between them.

The next part defines another pod labeled and named ***srsenb***. Inside of it, only one container, also called ***srsenb,*** is defined. It is based on the last version of the srsLTE Docker image created and explained in the previous section (*javipalomares/srslte:latest*). After that, the environmental variables to configure the container (enb_mcc, enb_mnc, enb_prb, empower_pod_addr) are instantiated. Then, as in the previous case, to give the container root privileges, the *securityContext* is set to true. As mentioned at the beginning of this section, the RAN logic is going to be deployed in the worker-node2. So, once the node is labeled kworker2, the pod is assigned to it.

```
kind: Pod
metadata:
    name: srsenb
    labels:
        app: srsenb
spec:
    containers:
        - name: srsenb              #Container Nº1
            image: javipalomares/srslte:latest
            env:                    #Environmental variables
            - name: enb_mcc
                value: "001"
            - name: enb_mnc
                value: "03"
            - name: enb_id
                value: "0x19B"
            - name: enb_prb
                value: "75"
            - name: empower_pod_addr
                value: "127.0.0.1"
            securityContext:
                privileged: true
    nodeSelector:
        IDname: kworker2            #Node label
```

In the scenario of having only one worker node, the only thing that needs to be changed is to assign the core and the RAN containers to the label of your node and everything will be deployed in that node.

The following part defines a service named ***epc-mongo-express-service*** that is connected to the pod called ***epc***. As explained in Section 2.2.2, a service is an abstract way to expose an application running on a set of pods. The type of service is defined, in this case is *NodePort*, that is used to expose the service on each Node's IP at a static port. The port command exposes the K8s service on the specified port within the cluster. The *targetPort* indicates the port on which the service will send requests to. The port that is exposed is called ***web-ui***, it is accessible in port 8081 and exposed externally in port 30000. The connecting protocol is TCP. This port gives access to the mongo express interface with the list of subscribers.

```
kind: Service
metadata:
    name: epc-mongo-express-service
spec:
    selector:
        app: epc                    #Connect to this pod
    type: NodePort                  #Type of the service
    ports:
    - name: web-ui
        protocol: TCP               #Connecting protocol
        port: 8081
        targetPort: 8081                  #Exposed port
        nodePort: 30000             #NodePort assigned static port
```

Finally, the last part defines a service named ***epc-open5gs-webui-service*** that is connected to the pod called ***epc***. The port exposed is called ***web-ui***, it is

accessible in port 3000 and it is exposed externally in port 30001. The connecting protocol is TCP. This port gives access to the Open5GS Web-UI, where the subscribers have to be registered.

```
kind: Service
metadata:
    name: epc-open5gs-webui-service
spec:
    selector:
        app: epc                        #Connect to this pod
    type: NodePort                      #Type of the service
    ports:
    - name: web-ui
        protocol: TCP                   #Connecting protocol
        port: 3000
        targetPort: 3000                      #Exposed port
        nodePort: 30001                 #NodePort assigned static port
```

The complete descriptor file can be found in ANNEX V: Descriptor file.

3.3.5. Get the deployment running

Once the setup is properly physically connected and set up as explained in Section 3.3.1 and Section 3.3.2, it is time to label each worker node in orderto assign the pods described in the descriptor file. To do so, the following commands must be run in the master console:

```
kubectl label nodes kubernetes-worker1 IDname=kworker1
kubectl label nodes kubernetes-worker2 IDname=kworker2
```

After that, the descriptor file to deploy the containers and services can be launched by introducing the following command:

```
kubectl apply -f K8s_deployment.yaml
```

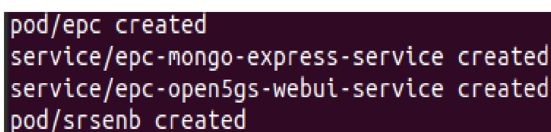

```
pod/epc created
service/epc-mongo-express-service created
service/epc-open5gs-webui-service created
pod/srsenb created
```

Figure 3.6: Output of the launched descriptor.

Figure 3.6 shows the pods and the services have been properly created. Then, it can be accessing from the master web browser by typing type <IP>:3000. The value of IP can be got by running any of the next commands:

```
kubectl get pods -o wide   or   kubectl get service
```

When using the first command, the IP address of a pod called ***epc*** has to be retrieved. On the other hand, when using the second command, the IP of the service named ***epc-open5gs-webui-service*** must be checked. Both IP addresses redirect the user to the Open5GS Web-UI web page. The usage of the second option is recommended due to its persistence against pod failure.

To register the UE in the Open5GS Web-UI, the same steps as in Section 3.2.2 have to be followed, but using the <IP>:3000 instead of localhost:3000. Once the subscriber is added, the APN can be activated and the UE connected. To check that everything has been connected properly the following commands can be used:

- To get the logs of the Open5GS software, the following steps can be taken as a reference:

```
kubectl log epc open5gs
```

Figure 3.7 shows that the MME service has been initiated and connected through the port 36412 of the S1-MME. Furthermore, it displays that the eNB-S1 connection is accepted and added. Also, the UE is detected and connected, and the number of MME-sessions is increased to 1.

```
[app] INFO: MME initialize...done (../src/mme/app-init.c:33)
[gtp] INFO: gtp_server() [127.0.0.2]:2123 (../lib/gtp/path.c:31)
[gtp] INFO: gtp_connect() [127.0.0.3]:2123 (../lib/gtp/path.c:58)
[mme] INFO: s1ap_server() [127.0.1.100]:36412 (../src/mme/s1ap-sctp.c:56)
[diam] INFO: CONNECTED TO 'hss.localdomain' (SCTP,soc#14): (../lib/diameter/commo
n/logger.c:108)
[mme] INFO: eNB-S1 accepted[127.0.1.1]:54329 in s1_path module (../src/mme/s1ap-s
ctp.c:108)
[mme] INFO: eNB-S1 accepted[127.0.1.1] in master_sm module (../src/mme/mme-sm.c:1
72)
[mme] INFO: [Added] Number of eNBs is now 1 (../src/mme/mme-context.c:1798)
[mme] INFO: InitialUEMessage (../src/mme/s1ap-handler.c:223)
[mme] INFO: [Added] Number of eNB-UEs is now 1 (../src/mme/mme-context.c:3172)
[mme] INFO:     ENB_UE_S1AP_ID[1] MME_UE_S1AP_ID[1] TAC[7] CellID[0x19b01] (../sr
c/mme/s1ap-handler.c:358)
[mme] INFO: [001035432000001] Unknown UE by IMSI (../src/mme/mme-context.c:2279)
[mme] INFO: [Added] Number of MME-UEs is now 1 (../src/mme/mme-context.c:2124)
[emm] WARNING: [] Attach request (../src/mme/emm-sm.c:198)
[emm] INFO:     IMSI[001035432000001] (../src/mme/emm-handler.c:172)
[mme] INFO: [Added] Number of MME-Sessions is now 1 (../src/mme/mme-context.c:318
```

Figure 3.7: Open5GS MME console output.

Figure 3.8 displays that the SGW-U service has been initiated and connected through the port 2152 of the S1-U. Furthermore, PFCP is associated, and the number of SGWU-sessions is increased to 1.

```
[app] INFO: SGW-U initialize...done (../src/sgwu/app.c:31)
[pfcp] INFO: pfcp_server() [127.0.0.6]:8805 (../lib/pfcp/path.c:31)
[gtp] INFO: gtp_server() [127.0.0.6]:2152 (../lib/gtp/path.c:31)
[pfcp] INFO: ogs_pfcp_connect() [127.0.0.3]:8805 (../lib/pfcp/path.c:59)
[sgwu] INFO: PFCP associated (../src/sgwu/pfcp-sm.c:168)
[sgwu] INFO: UE F-SEID[CP:0x1 UP:0x1] (../src/sgwu/context.c:143)
[sgwu] INFO: [Added] Number of SGWU-Sessions is now 1 (../src/sgwu/context.c:148)
```

Figure 3.8: Open5GS SGW-U console output.

Figure 3.9 depicts that the SGW-C service has been initiated and connected through the port 2123. Also, PFCP is associated, and the number of SGWC-sessions is increased to 1.

```
[app] INFO: SGW-C initialize...done (../src/sgwc/app.c:31)
[gtp] INFO: gtp_server() [127.0.0.3]:2123 (../lib/gtp/path.c:31)
[pfcp] INFO: pfcp_server() [127.0.0.3]:8805 (../lib/pfcp/path.c:31)
[pfcp] INFO: ogs_pfcp_connect() [127.0.0.6]:8805 (../lib/pfcp/path.c:59)
[sgwc] INFO: PFCP associated (../src/sgwc/pfcp-sm.c:172)
[sgwc] INFO: [Added] Number of SGWC-UEs is now 1 (../src/sgwc/context.c:209)
[sgwc] INFO: [Added] Number of SGWC-Sessions is now 1 (../src/sgwc/context.c:865)
[gtp] INFO: gtp_connect() [127.0.0.4]:2123 (../lib/gtp/path.c:58)
```

Figure 3.9: Open5GS SGW-C console output.

- To get the logs of the srsLTE software:

```
kubectl log srsenb
```

Figure 3.10 displays that the eNB has been properly configured and that the User 0x46 has been registered correctly and it can connect to the internet without any problems.

```
Opening 1 channels in RF device=default with args=default
[INFO] [UHD] linux; GNU C++ version 9.2.1 20200304; Boost_107100; UHD_3.15.0.0-2build5
[INFO] [LOGGING] Fastpath logging disabled at runtime.
[INFO] [UHD RF] RF UHD Generic instance constructed
[INFO] [B200] Detected Device: B210
[INFO] [B200] Operating over USB 3.
[INFO] [B200] Initialize CODEC control...
[INFO] [B200] Initialize Radio control...
[INFO] [B200] Performing register loopback test...
[INFO] [B200] Register loopback test passed
[INFO] [B200] Performing register loopback test...
[INFO] [B200] Register loopback test passed
[INFO] [B200] Asking for clock rate 23.040000 MHz...
[INFO] [B200] Actually got clock rate 23.040000 MHz.
Opening USRP channels=1, args: type=b200,master_clock_rate=23.04e6
Setting frequency: DL=2680.0 Mhz, UL=2560.0 MHz for cc_idx=0 nof_prb=50
13:32:38.063822 [COMN   ] [D] [    0] RxSockets: socket fd=13 has been registered.
13:32:38.064001 [COMN   ] [D] [    0] RxSockets: socket fd=14 has been registered.

==== eNodeB started ===
Type <t> to view trace
Closing stdin thread.
RACH:  tti=6591, cc=0, preamble=0, offset=1, temp_crnti=0x46
User 0x46 connected
```

Figure 3.10: srsLTE eNB console output.

The ports shown in, Figure 3.8 and Figure 3.9 are the same ports used to interconnect the Open5GS 5G NSA architecture shown in Figure 2.8. Adding that to the connectivity achievement, it can be stated that the setup has been deployed correctly.

3.3.6. Problems found

Four main problems have been faced during this deployment:

- While deploying the Open5GS CN container, an error of shared libraries arose and it was solved as explained in Section 3.3.3.2.
- The assignation of IPs done by the switch to the nodes is not permanent so, in various occasions, the cluster stopped working because the addresses had changed and the cluster had to be deployed all over again.

It only happened once, so the solution was to deploy the cluster again with the new addresses by adding them to the "*etc/hosts*" file as explained in Section 3.3.1.

- Due to the fact that the nodes that compose the cluster are physically connected to the switch, sometimes they got disconnected from it and the deployment stopped functioning even if the pods seem to be in a "Running" status. This behavior can be observed in Figure 3.11. If the disconnection was brief, the Calico services and pods that controlled the networking between nodes could get restarted automatically. If the disconnection occurred overnight, the only solution that was found was to remove and recreate the previous processes manually.

- Networking problems such as no connection between the containers on different pods (srsLTE and Open5GS) appeared. The creation of K8s services was necessary to expose the necessary ports for the pods to see connect correctly.

```
NAME                                       READY   STATUS
calico-kube-controllers-7b4657785d-gwvvm   1/1     Running
calico-node-fkp4g                          0/1     Running
calico-node-wh7dr                          0/1     Running
```

Figure 3.11: Calico pod running error.

3.4. Docker-based Deployment

In order to compare the behavior of Docker containers and analyze the performance of other existing open-source projects, a Docker-based project is selected for deployment and analysis [61]. This section deploys a single node setup using an existing Docker project and the following hardware components: (i) two computers with Ubuntu 20.04; (ii) an USRP B210 [62] with USB 3.0 connected to the computer that deploys the RAN; (iii) a sysmoUSIM [63]; and (iv) a HUAWEI LTE USB Stick [64]. displays the setup connected.

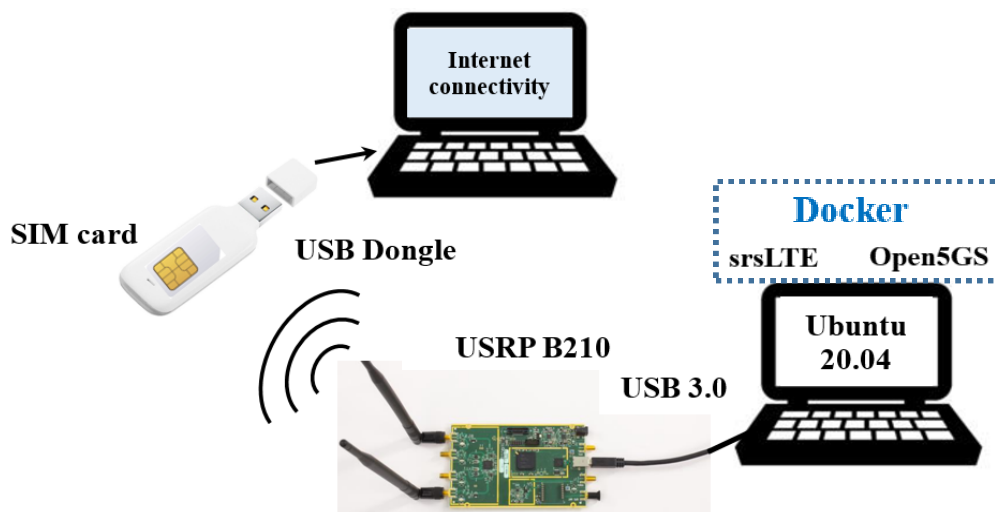


Figure 3.12: Docker deployment setup.

3.4.1. Installing software and dependencies

In this setup, there are two mandatory software requirements: (i) docker-ce and (ii) docker-compose. To install them, the following commands are required:

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
sudo apt-get install docker-compose
```

After that, the repository can be cloned to build the base Docker image of Open5GS:

```
git clone https://github.com/herlesupreeth/docker_open5gs
cd docker_open5gs/base
docker build -t docker_open5gs .

cd ../ims_base
docker build -t docker_kamailio .
```

3.4.2. Setup and configuration

To properly configure the network, the following parameters in the "*.env*" file for the particular scenario must be edited:

```
MCC         --> First three digits of the IMSI of the SIM card.
MNC         --> Next two digits after the MCC of your IMSI.
DOCKER_HOST_IP    --> IP address of the host running the Docker setup.
SGWU_ADVERTISE_IP --> Change this value to the DOCKER_HOST_IP only if the eNB is not running in the same host.
UPF_ADVERTISE_IP --> Like the SGWU_ADVERTISE_IP, change this value to the DOCKER_HOST_IP only if the eNB is not running in the same host.
```

Furthermore, the TAC parameter in the "*srslte/rr.conf*" file must be edited to match the corresponding Tracing Area Code (TAC). Also, If the eNB is not running in the same host as the CN, the additional steps are required:

```
Under the "mme" section in the nsa-deploy.yaml file, uncomment the following part:

    # ports:
    #   - "36412:36412/sctp"

Under the "amf" section in the nsa-deploy.yaml file, uncomment the following part:

    # ports:
    #   - "38412:38412/sctp"

Under the "sgwu" section in the nsa-deploy.yaml file, uncomment the following part:

    # ports:
    #   - "2152:2152/udp"
```

These ports have to be exposed in order to assure there is connectivity between the modules of the Open5GS core, as shown in Figure 2.8.

On the other hand, If the eNB is not running in the same host as the CN, make sure that the host running the eNB has the static route to the SGWU container or the UE will not find the core. It can be added by running:

```
ip r add <SGWU_CONTAINER_IP> via <SGWU_ADVERTISE_IP>
```

3.4.3. Get the deployment running

Once the previous modifications have been made, the following commands are of guide to deploy the network. Notice that it is needed to run the core and the RAN commands in separate terminals:

```
cd ..
set -a
source .env

#Build and run the Open5GS NSA network in terminal 1
sudo docker-compose build
sudo docker-compose -f nsa-deploy.yaml up

# Build and run srsRAN eNB in terminal 2
sudo docker-compose -f srsenb.yaml build
sudo docker-compose -f srsenb.yaml up -d && sudo docker attach srsenb
```

This creates a Docker container for each of the modules that are part of the Open5GS CN (amf, ausf, hss, mme, sgwu, sgwc, etc.) and one for the srsenb. This can be checked by running:

```
sudo docker ps
```

To register the UE in the Open5GS Web-UI, the same steps as in Section 3.2.2 have to be followed, but using the <DOCKER_HOST_IP>:3000 instead of localhost:3000. Once the subscriber is added, the UE can be connected and the APN activated. Figure 3.13 shows the output of the srsenb running command. Even though the UE achieves constant internet connectivity, it connects and disconnects continuously from the network. For now, it is a valid solution, but this will represent a problem in the network evaluation of the next chapter. This issue has been reported to the maintainers of the project.

```
User 0x160 connected
RACH:  tti=5611, cc=0, preamble=28, offset=9, temp_crnti=0x162
Disconnecting rnti=0x162.
RACH:  tti=5881, cc=0, preamble=37, offset=1, temp_crnti=0x163
User 0x163 requesting RRC Reestablishment as 0x160. Cause: otherFailure
Disconnecting rnti=0x160.
User 0x163 connected
RACH:  tti=7181, cc=0, preamble=34, offset=1, temp_crnti=0x164
User 0x164 requesting RRC Reestablishment as 0x163. Cause: otherFailure
Disconnecting rnti=0x163.
User 0x164 connected
RACH:  tti=8421, cc=0, preamble=7, offset=1, temp_crnti=0x165
User 0x165 requesting RRC Reestablishment as 0x164. Cause: otherFailure
Disconnecting rnti=0x164.
User 0x165 connected
RACH:  tti=10081, cc=0, preamble=13, offset=1, temp_crnti=0x166
User 0x166 requesting RRC Reestablishment as 0x165. Cause: otherFailure
Disconnecting rnti=0x165.
User 0x166 connected
RACH:  tti=621, cc=0, preamble=43, offset=1, temp_crnti=0x167
User 0x167 requesting RRC Reestablishment as 0x166. Cause: otherFailure
Disconnecting rnti=0x166.
```

Figure 3.13: Intermittent connectivity achieved with Docker.

CHAPTER 4. PERFORMANCE EVALUATION AND ACHIEVED RESULTS

4.1. Methodology

This chapter will be focusing on an analysis of the resources performance of the three deployments, taking special attention to the results achieved in the fully virtualized open-source-based network. The setup utilized for the baremetal, Kubernetes and Docker deployments are represented in Figure 3.1, Figure 3.5 and Figure 3.12, respectively. The specifications of the computers used in all the scenarios are: an Intel Core i7-6500 2,5GHz processor with 16 GB of RAM. Table 4.1 summarizes a set of experiments that are going to be carried out for each deployment. It also specifies the aimed measurement and the parameters that will be changed for each case.

Table 4.1: List of the experiments with their parameters.

# Experiment	# PRBs	Distances (m)	BW (Mbps)	Measurement
1	25	[1, ..., 20]	[1, ..., 150]	Throughput
2	50	[1, ..., 20]	[1, ..., 150]	Throughput
3	75	[1, ..., 20]	[1, ..., 150]	Throughput
4	25, 50, 75	1	[1, ..., 150]	Throughput
5	25, 50, 75	3	[1, ..., 150]	Throughput
6	25, 50, 75	6	[1, ..., 150]	Throughput
7	25, 50, 75	10	[1, ..., 150]	Throughput
8	25, 50, 75	15	[1, ..., 150]	Throughput
9	25, 50, 75	20	[1, ..., 150]	Throughput
10	25, 50, 75	1, 3	1	Resources
11	25, 50, 75	1, 6	25	Resources
12	25, 50, 75	1	50	Resources
13	25, 50, 75	1	75	Resources
14	25, 50, 75	1	100	Resources
15	25, 50, 75	1	150	Resources
16	50	1	100	Time to restart
17	75	3	75	Time to restart

The tests have been carried out using the IPERF3 [74] software, which is a tool that allows generating data streams for both Transport Control Protocol (TCP) and User Datagram Protocol (UDP), to measure network performance. For the evaluation of the deployments, only UDP streams are going to be used, to be able to change the transmission speed parameter.

In the deployed networks, the IPERF3 software has to be installed on both terminals: the UE and the node that deploys the eNB. A phone is being used as UE and acts as an IPERF3 client and the eNB acts as the server. In order to do so, run the following commands:

```
#Server:
iperf3 -s -B 192.168.11.194 -p 5000 -i 0.25 > n_15mPRB50.txt
```

```
#Client:
-u -c 192.168.11.194 -t 60 -i 0.25 -4 -p 5000 -O 1 -b 25M
```

The meaning of the flags of the server command are: (i) the “s” indicates that IPERF3 is going to be used as a server; (ii) the “B” is followed by the IP address of the server; (iii) the “p” represents the port that the server is going to be exposed; and (iv) the “i” represents the granularity of the measurements (the unit is the second). The last parameter indicates the name of the file where the data is going to be stored. For the sake of this work and to make it easier for the future managing and analysis of the data, the next structure has been followed:

“***n**_distance**PRB**NumberofPRB.txt*” ---> n_15mPRB50.txt

The meaning of the flags of the client command are: (i) the “u” indicates UDP data streams are going to be used; (ii) the “c” shows that IPERF3 is going to be used as a client and it is followed by the IP address of the server; (iii) the “t” represents the number of measurements of the transmission; (iv) the “i” represents the granularity of the measurements (the unit is the second); (v) the “4” indicates that iPv4 addresses are going to be used; (vi) the “p” represents the port that the client is going to be accessing; (vii) the “O” indicates the number of initial seconds that are going to be omitted, to avoid inaccuracies related to the setting up of the connection (is a real connection and it can’t change the state instantly); and (viii) the “b” represents the speed of the transmitted data.

To acquire the computational resources utilized by each module of Open5GS and srsLTE, a different command had to be run in each of the deployments:

- For the Baremetal deployment, the following command saves in the “*top-5iterations.txt”* file, five iterations of the system resources:

```
top -b -n 5 > top-5iterations.txt
```

- For the Docker and K8s deployments, the following command saves each second the system resources in the “*resources.txt”* file:

```
sudo docker stats > resources.txt
```

The measurements have been executed five times, to provide better accuracy to the evaluation and to show the confidence interval at 95%, in each of the following scenarios:

- Different deployments: Baremetal, Docker, two nodes Kubernetes and three nodes Kubernetes.
- Number of PRBs on the eNB: 25, 50 and 75.
- Distances of the UE form the RAN: 1m, 3m, 6m, 10m, 15m and 20m.
- Transmission speed of the packet streams: 1Mbps, 25Mbps, 50Mbps, 75Mbps, 100Mbps and 150Mbps.

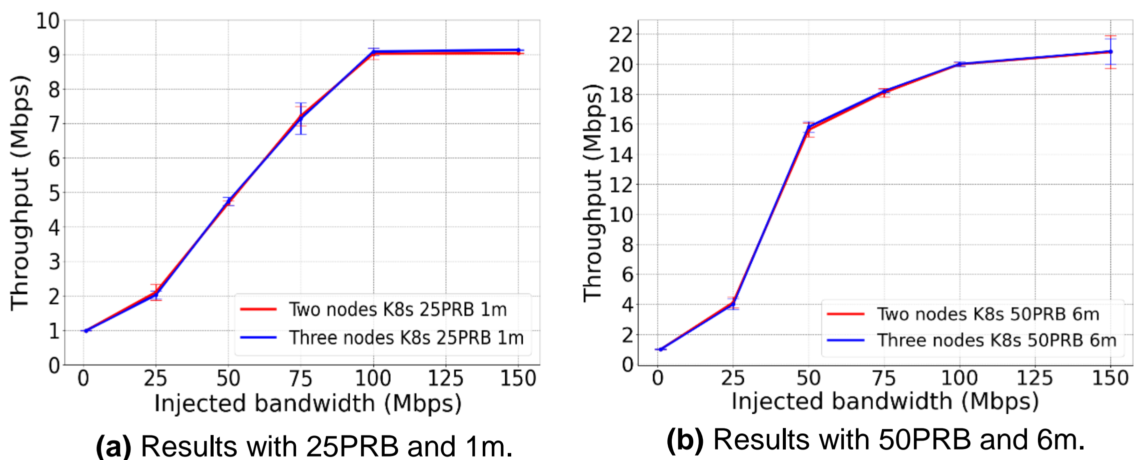
$N_{measurements} = 5 * 4 * 3 * 6 * 6 = 2160$ $\qquad N_{lines\ of\ data} = 2160 * 60 = 129600$

Furthermore, in addition to the previous number, other measurements have been made to test the resource usage and the relaunch time of the system, when

forcing failure in a part of the deployment. Over 2300 measurements and 130000 lines of data have been obtained in total. In order to extract evaluations and comparisons of the raw data, some cleaning scripts have been necessary. This code is shown in the ANNEX VI. Also the scripts used to plot the results can be found in that annex.

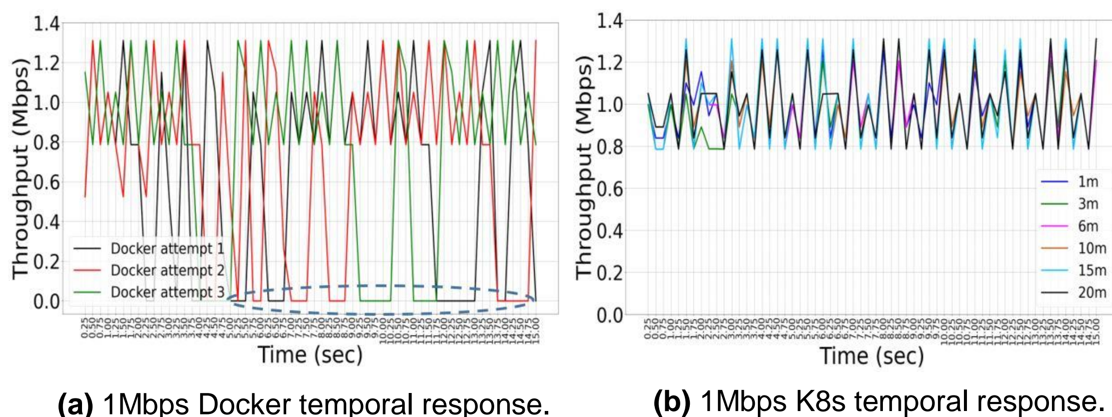
4.2. Results discussion

Before discussing the obtained results, it is necessary to indicate a few things. The first one is that, due to the similarity of the data gathered from the two-node K8s and the three-node K8s deployments, only the three nodes setup results will be shown due to space constraints in this document. These similarities can be appreciated as an example in Figure 4.1, where it is shown the throughput comparison of both setups, for two different set of conditions (distance an number of PRB).



(a) Results with 25PRB and 1m. **(b)** Results with 50PRB and 6m.

Figure 4.1: Results comparison between two and three nodes in K8s.



(a) 1Mbps Docker temporal response. **(b)** 1Mbps K8s temporal response.

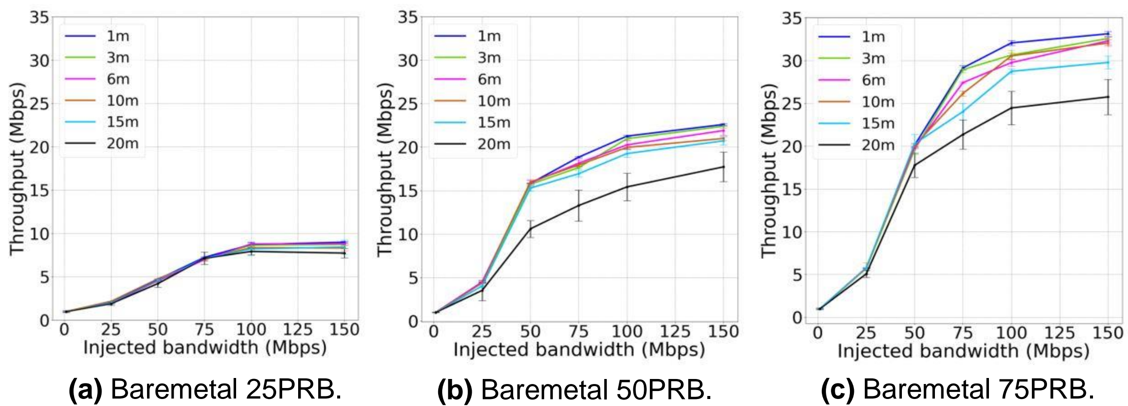
Figure 4.2: Comparison between the temporal response of Docker and K8s.

The second point to take into account is that, as mentioned in Section 3.4.3, the Docker deployment used to compare with the Kubernetes setup designed in this Master Thesis, has an error, which makes the UE get connected and

disconnected continuously. This makes the average throughput of the Docker setup decrease in comparison with the baseline case. This behavior can be clearly seen in Figure 4.2, that compares the temporal response of the Docker and the Kubernetes deployment while injecting a bandwidth of 1 Mbps.

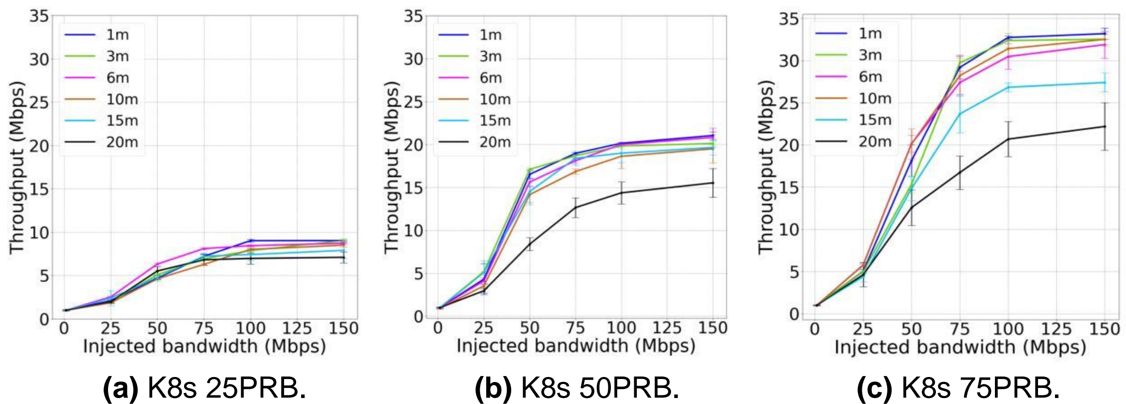
4.2.1. Measuring throughput

In experiments 1 to 9 the average throughput has been measured using the values that IPERF3 provides at the end of each test. It automatically averages the throughput of the whole connection. Experiments 1, 2, and 3 compare the effects of distance and number of PRB in the output, on each deployment, as shown in Figure 4.3, Figure 4.4 and Figure 4.5, respectively.



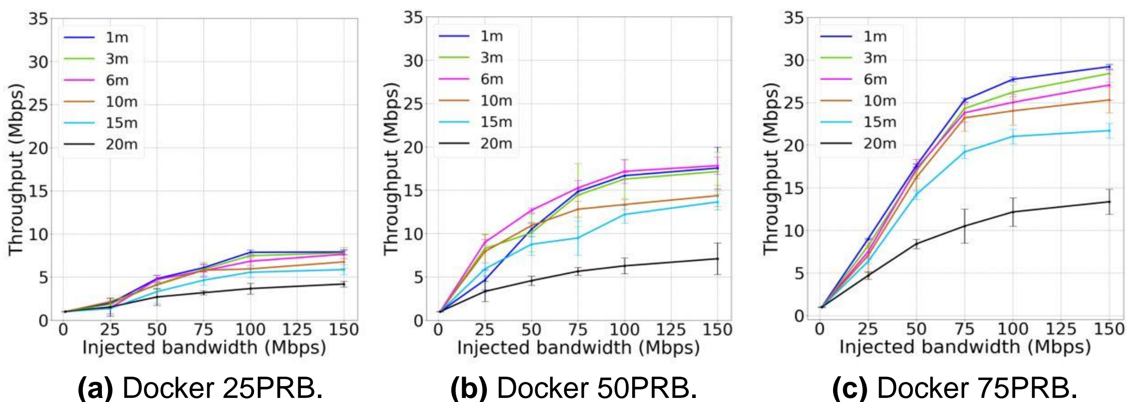
(a) Baremetal 25PRB. **(b)** Baremetal 50PRB. **(c)** Baremetal 75PRB.

Figure 4.3: Baremetal throughput comparison at diff. distances with each PRB.



(a) K8s 25PRB. **(b)** K8s 50PRB. **(c)** K8s 75PRB.

Figure 4.4: K8s throughput comparison at different distances with each PRB.

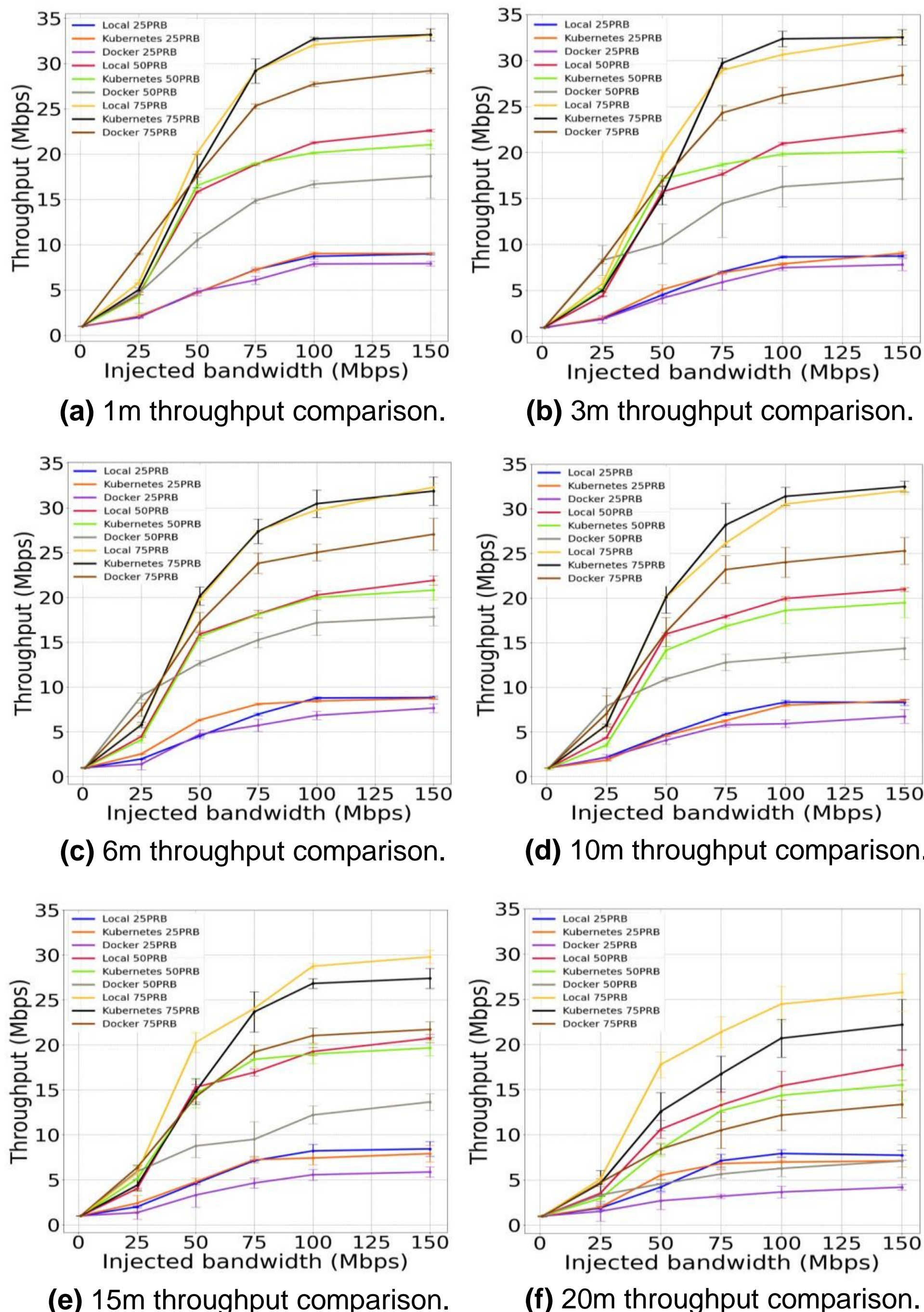


(a) Docker 25PRB. **(b)** Docker 50PRB. **(c)** Docker 75PRB.

Figure 4.5: Docker throughput comparison at diff. distances with each PRB.

By analyzing the results, it is readily observed in Figure 4.4, the impact that the parameters distance and number of PRB have in the throughput of the Kubernetes deployment. As it could be deduced, the more resources are allocated (higher number of PRB), the better throughput is obtained. On the other hand, it is noticed that the longer the distance between the UE and the RAN, the lower the throughput. Furthermore, in longer distances, the confidence interval at 95% gets increased. This means that the value of the throughput fluctuates more with regard of the average. These behaviors are due to interferences and free space losses. As displayed in Figure 4.3 and, the other deployments follow the same tendencies as the ones commented in the K8s deployment.

Moreover, in experiments 4 to 9 the average throughput of the three deployments is tested using distance as the comparing factor.



(a) 1m throughput comparison.

(b) 3m throughput comparison.

(c) 6m throughput comparison.

(d) 10m throughput comparison.

(e) 15m throughput comparison.

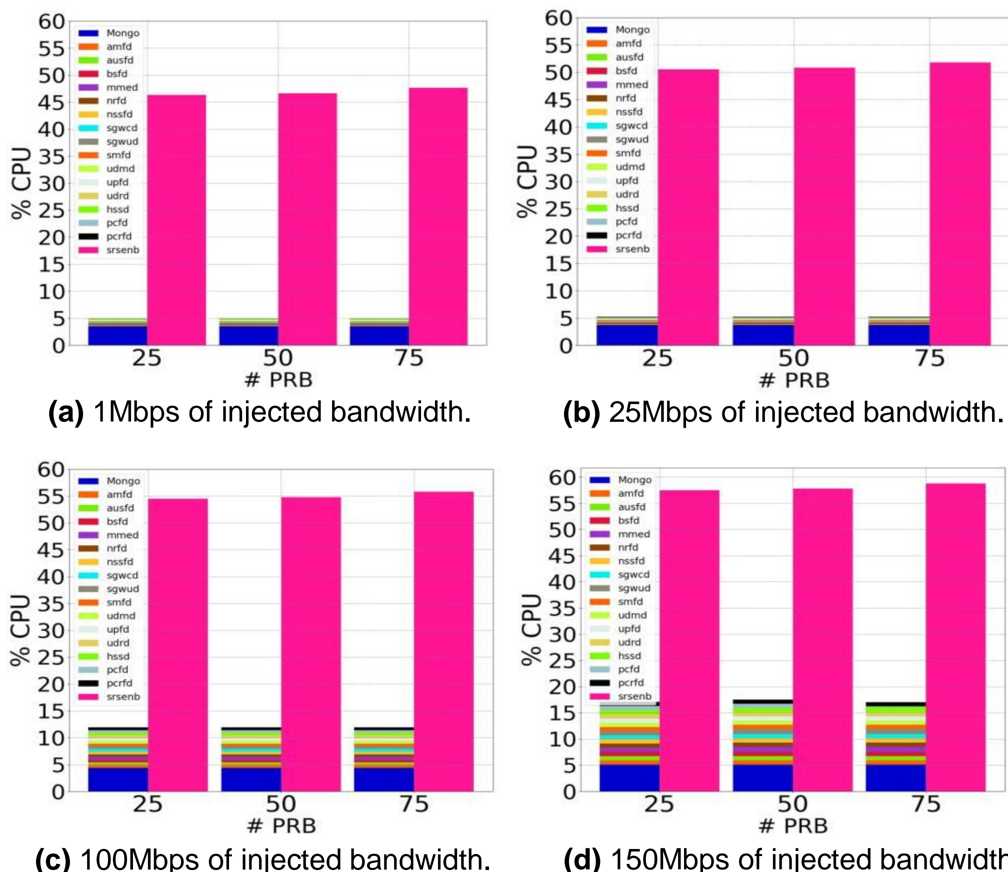
(f) 20m throughput comparison.

Figure 4.6: Comparison between deployments and PRBs at diff. distances.

Figure 4.6 displays six subfigures that represent a summary of the average throughput in all the deployments, while changing the value of the parameter number of PRB, at different distances. Furthermore, the tendencies obtained in experiments 1 to 3 are observed as well, such as the impact of: (i) the number of PRB; and (ii) the distance between UE and RAN. This method of representing the information provides more trends in the results. By evaluating the gathered data from the K8s deployment against the rest of the setups, the first tendency that can be clearly appreciated in any of the subfigures, is different levels of achieved throughput. Comparing it to the baremetal setup, there is no actual improvement on the throughput. Their tendencies and values are very similar. In contrast with that, the achieved throughput compared with the Docker deployment, is much higher. This is due to the previously mentioned connectivity error. The difference becomes more obvious as the value of the number of PRB and distance increases. On the other hand, the Docker deployment has higher throughput at sorter areas, in the range of 0 to 25 Mbps of injected bandwidth. Lastly, as in the previous experiments, Figure 4.6 (f) shows how the fluctuation of the throughput with regard of the average, increases at larger distances (values of the 95% confidence interval are larger).

4.2.2. Measuring resource consumption

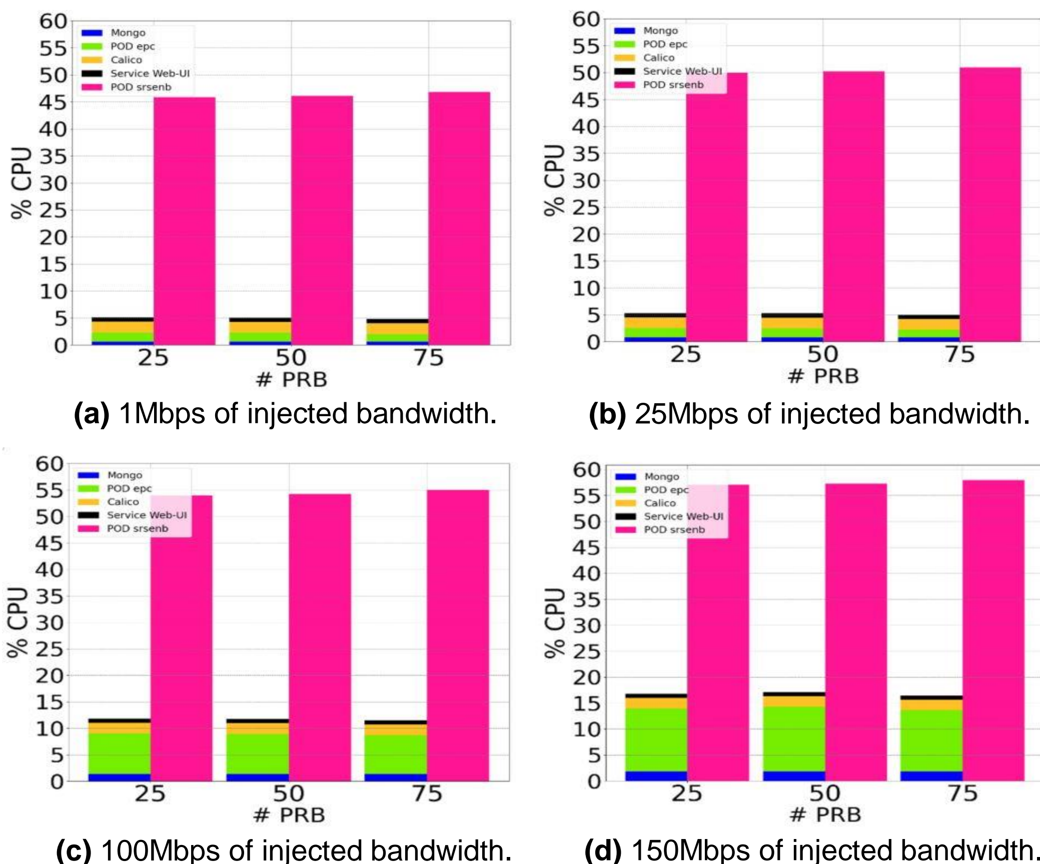
In experiments 10 to 15 the computational resources utilized by each module of Open5GS and srsLTE has been measured. Mainly, these experiments aim to compare the impact of the injected bandwidth and the number of PRB in the resources utilized on each deployment. Finally, some of the measurements have been repeated at different distances, to check their impact on the resources.



(a) 1Mbps of injected bandwidth. **(b)** 25Mbps of injected bandwidth.

(c) 100Mbps of injected bandwidth. **(d)** 150Mbps of injected bandwidth.

Figure 4.7: Comparison of baremetal CPU resources at different bandwidths.



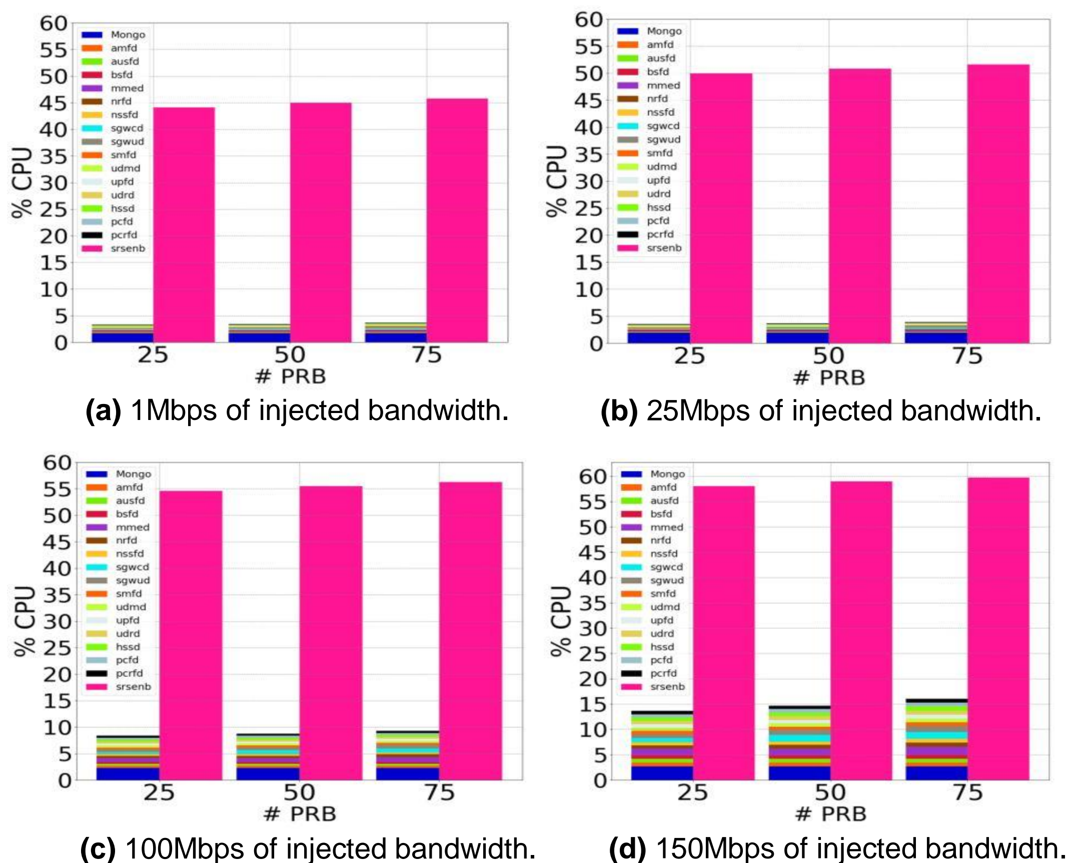
(a) 1Mbps of injected bandwidth.

(b) 25Mbps of injected bandwidth.

(c) 100Mbps of injected bandwidth.

(d) 150Mbps of injected bandwidth.

Figure 4.8: Comparison of K8s CPU resources at different bandwidths.



(a) 1Mbps of injected bandwidth.

(b) 25Mbps of injected bandwidth.

(c) 100Mbps of injected bandwidth.

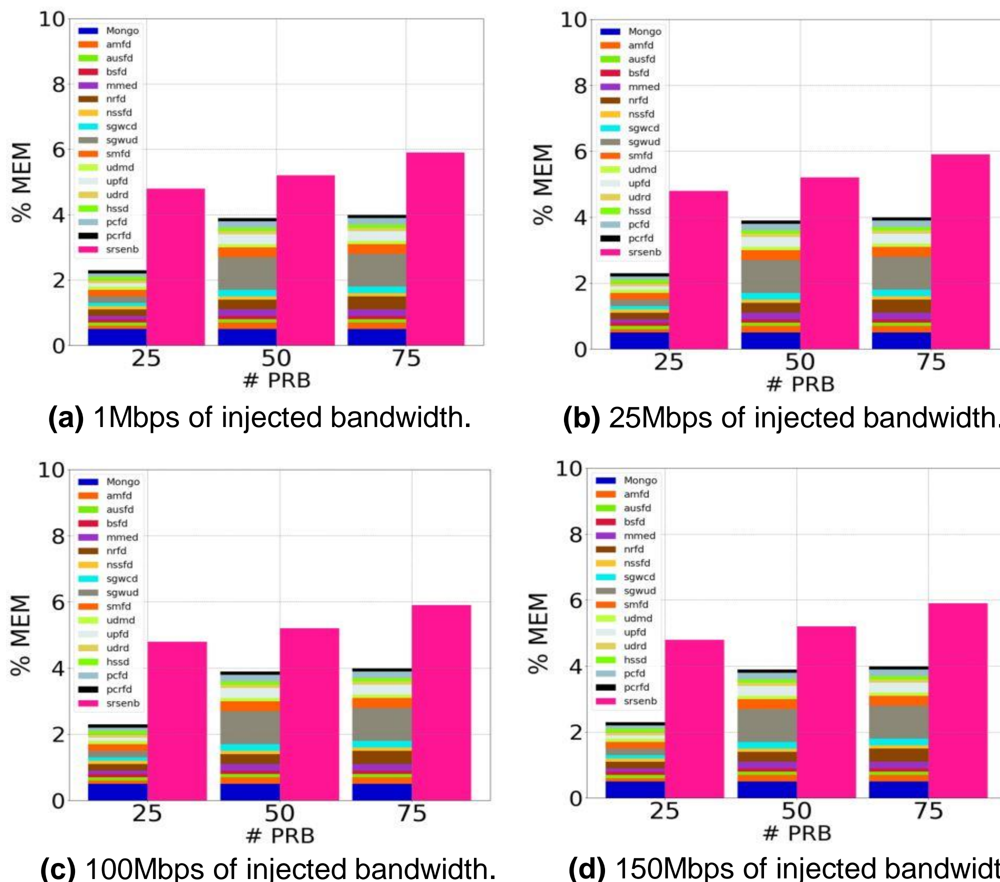
(d) 150Mbps of injected bandwidth.

Figure 4.9: Comparison of Docker CPU resources at different bandwidths.

Figure 4.7, Figure 4.8 and Figure 4.9 show the usage of CPU resources of the baremetal, the Kubernetes and the Docker deployments respectively, at four different injected bandwidths. After evaluating the results, the following conclusions have been achieved in the K8s deployment:

- The "*srsenb"* module requires much more resources than all the other modules combined, reaching almost the 60% of the CPU capacity at high injected bandwidths. This tendency is replicated in the rest of the setups.
- The number of PRB has a slight impact on the resources of the "*srsenb"* module and almost no repercussion in the rest of the elements. This behavior is repeated in the baremetal deployment, but not in the Docker setup. Figure 4.9 shows that, at higher injected bandwidths the impact of the number of PRB on the core modules is noticeable.
- The parameter that has the biggest impact in the used resources of all the modules, is the injected bandwidth. The higher it gets, the more resources are needed. This tendency is also replicated in the rest of the setups.
- The "*epc"* pod contains all the core functions and as it was expected, it consumes as much as all the baremetal and Docker contributions of the CN functions added together. Globally, under the same conditions, the same percentage of CPU is used in the three deployments.

The usage of the memory resources of the baremetal, the Kubernetes and the Docker deployments, at four different injected bandwidths, has been measured and are shown in Figure 4.10, Figure 4.11 and Figure 4.12 respectively.



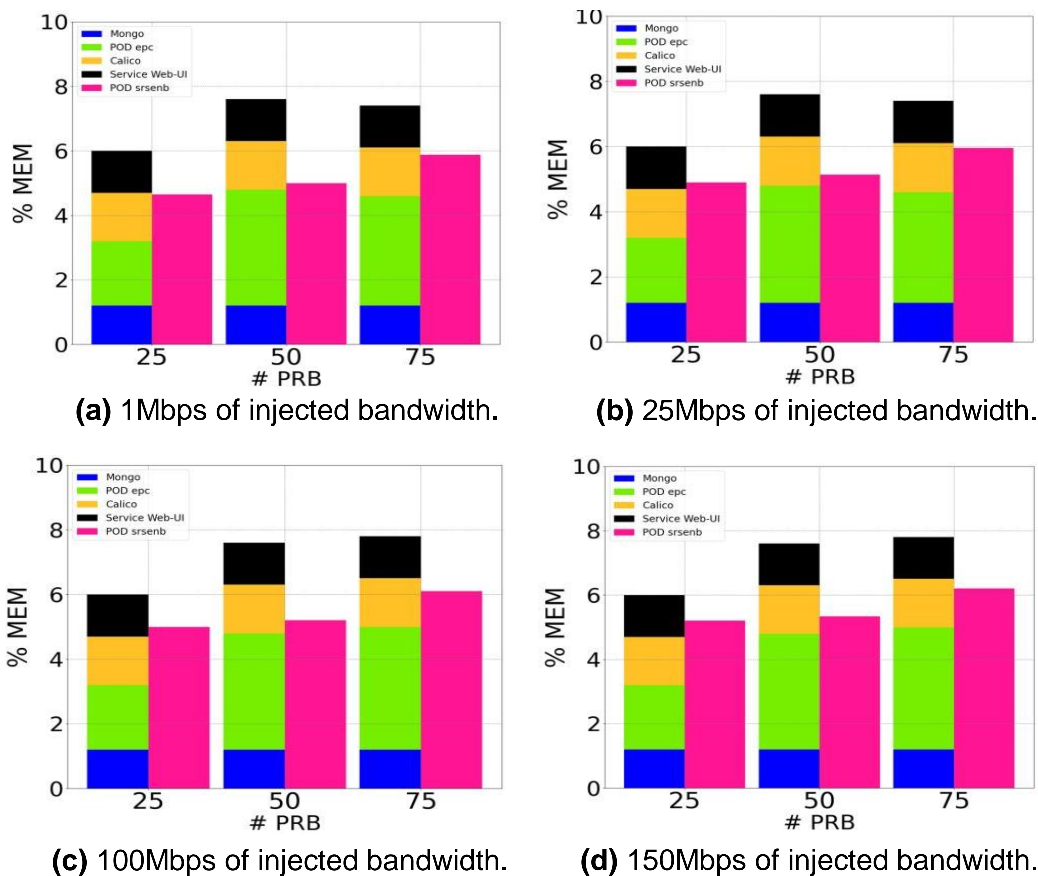
(a) 1Mbps of injected bandwidth.

(b) 25Mbps of injected bandwidth.

(c) 100Mbps of injected bandwidth.

(d) 150Mbps of injected bandwidth.

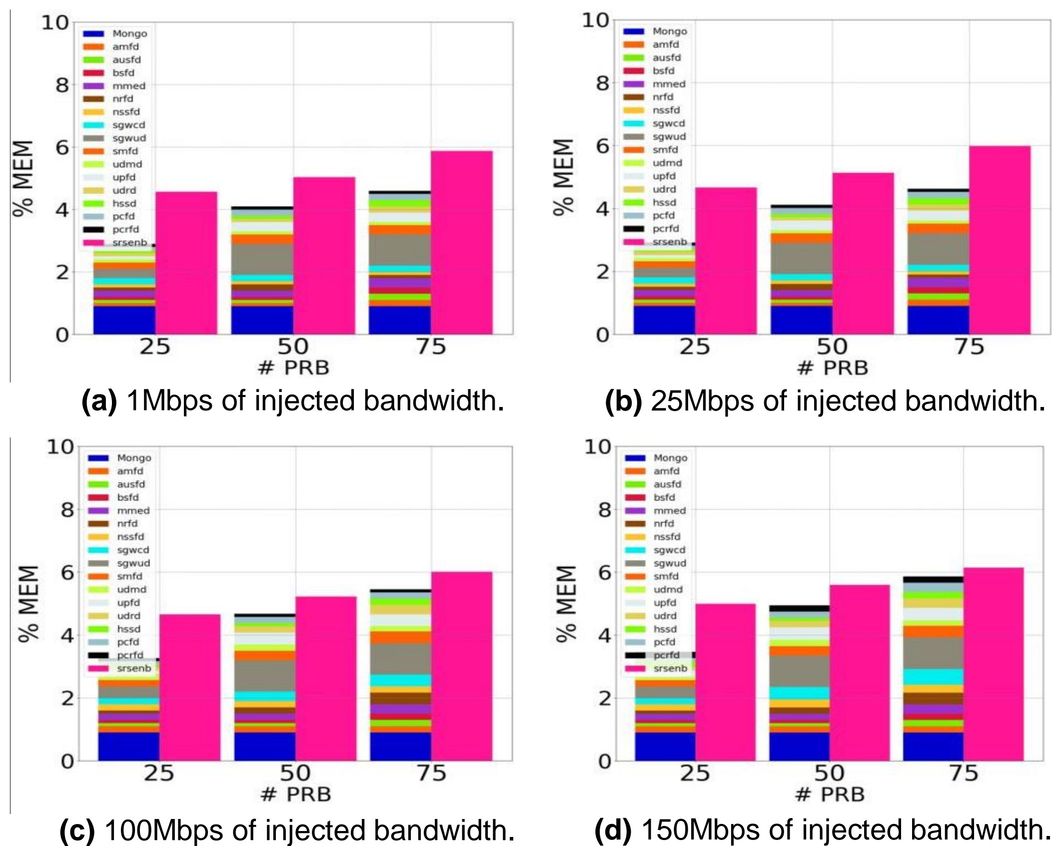
Figure 4.10: Comparison of baremetal MEM resources at diff. bandwidths.



(a) 1Mbps of injected bandwidth. (b) 25Mbps of injected bandwidth.

(c) 100Mbps of injected bandwidth. (d) 150Mbps of injected bandwidth.

Figure 4.11: Comparison of K8s MEM resources at different bandwidths.



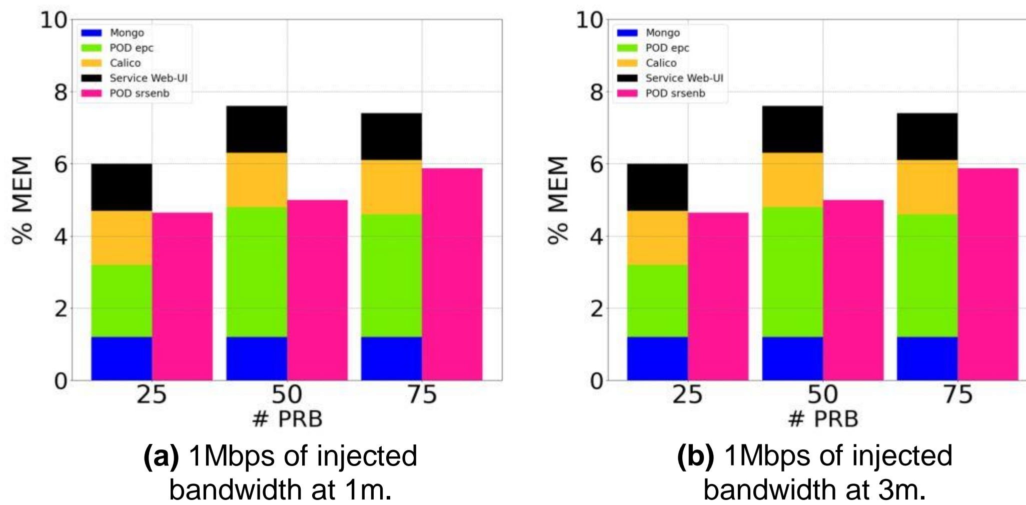
(a) 1Mbps of injected bandwidth. (b) 25Mbps of injected bandwidth.

(c) 100Mbps of injected bandwidth. (d) 150Mbps of injected bandwidth.

Figure 4.12: Comparison of Docker MEM resources at different bandwidths.

In contrast with the results achieved in the CPU resources experiments, the memory usage is more related to the number of PRB than to the injected bandwidth. Furthermore, the magnitude of the "*srsenb*" module is no longer dominant in terms of memory usage, even though it still has the biggest contribution. Finally, the K8s deployments is the most memory consuming of the three deployments by 4% in average, due to the memory cost of the deployment of the Kubernetes cluster. This extra 4% can be considered negligible when comparing it to the flexibility and ease to deploy that the K8s setup provides. Moreover, this effect is visible because there is no memory allocation on demand and all the modules are always working. A future line of work is to implement it and test the energy efficiency of the deployment.

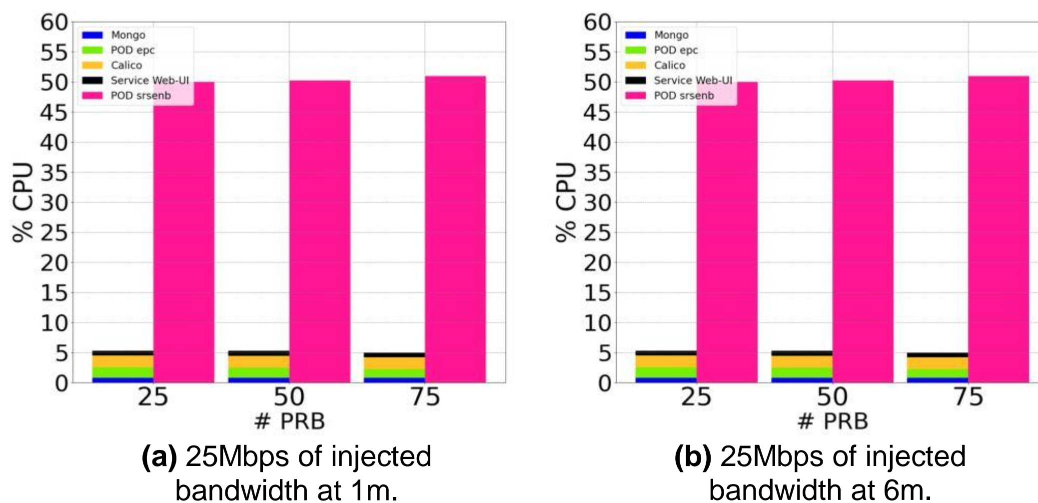
To prove that the distance between the RAN and the UE does not affect either the CPU or the memory recourses, the same experiment has been executed at different distances locking the rest of the parameters in the K8s deployment. Figure 4.13 and Figure 4.14 show that the distance between the UE and the RAN does not alter the usage of the system resources.



(a) 1Mbps of injected bandwidth at 1m.

(b) 1Mbps of injected bandwidth at 3m.

Figure 4.13: Comparison of K8s MEM resources at different distances.



(a) 25Mbps of injected bandwidth at 1m.

(b) 25Mbps of injected bandwidth at 6m.

Figure 4.14: Comparison of K8s CPU resources at different distances.

4.2.3. Measuring forced reconnecting time

The aim of experiments 16 and 17 is to observe how the system responds to a failure in any of its modules and if it is capable of solving that error and keep functioning. This is especially relevant in distributed environments where a virtual network function can suffer some temporary unavailability. These experiments aim to measure the time to reestablish the connection in those cases where the modules are able to reconnect after a failure. To do so, some of the CN modules have been forced to failed.

Figure 4.15 presents the reconnecting attempts of the Kubernetes deployment. If any of the modules that are related to the registration and storage of the UE fails, the system cannot regain connectivity. This behavior can be observed in the attempts 2 and 5 of Figure 4.15. This is due to the K8s descriptor file did not include volumes to storage the sensible information of the UE and once one of these modules fails, the information gets lost. Kubernetes restarts the container with a clean state. A volume is a directory on disk or in another container that storages information. A future line of work consists on implementing these volumes in the deployment file, to prevent this from happening again. Otherwise, two different times of reconnection can be observed: *t* and *t+10*. This is due to the working mode of the *"srsenb"* module. If the connection is lost, this module tries to reconnect every ten seconds, and repeats a reconnection attempt in case of failure.

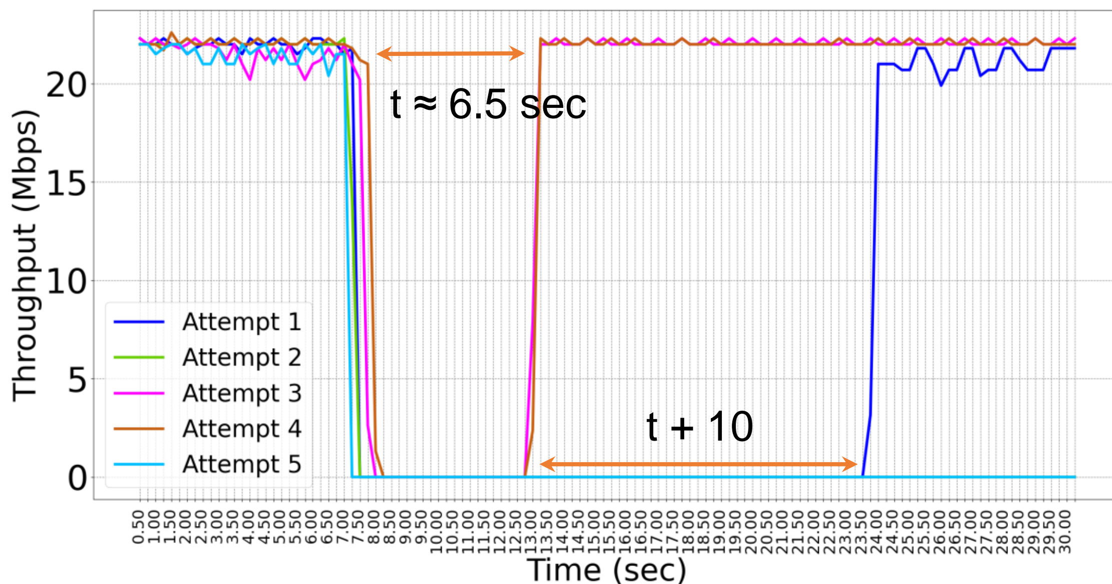


Figure 4.15: K8s forced reconnection time.

Figure 4.16 displays the attempts of making the baremetal system fail. All of them were solved and the reconnection was achieved. Two different times of reconnection can be observed in Figure 4.16: *t'* and *t'+10*. This is due to the already mentioned behavior of the *"srsenb"* module. The reconnection is always achieved due to all the modules that form the baremetal deployment are daemons that relaunch each time they fail. Otherwise, these modules and the entire network would have stayed down.

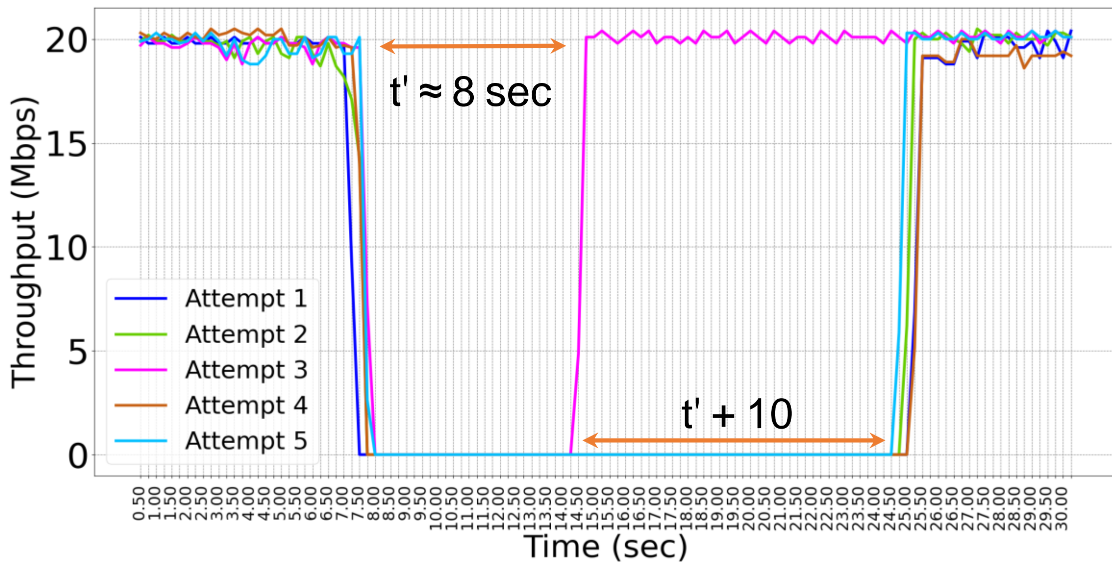


Figure 4.16: Baremetal forced reconnection time.

The response of the Docker deployment is very similar to the previous one. In this occasion, instead of been daemons, each module forming the CN and the RAN are Docker containers, which restart every time they fail. The same structure of time reconnecting can be observed in Figure 4.17.

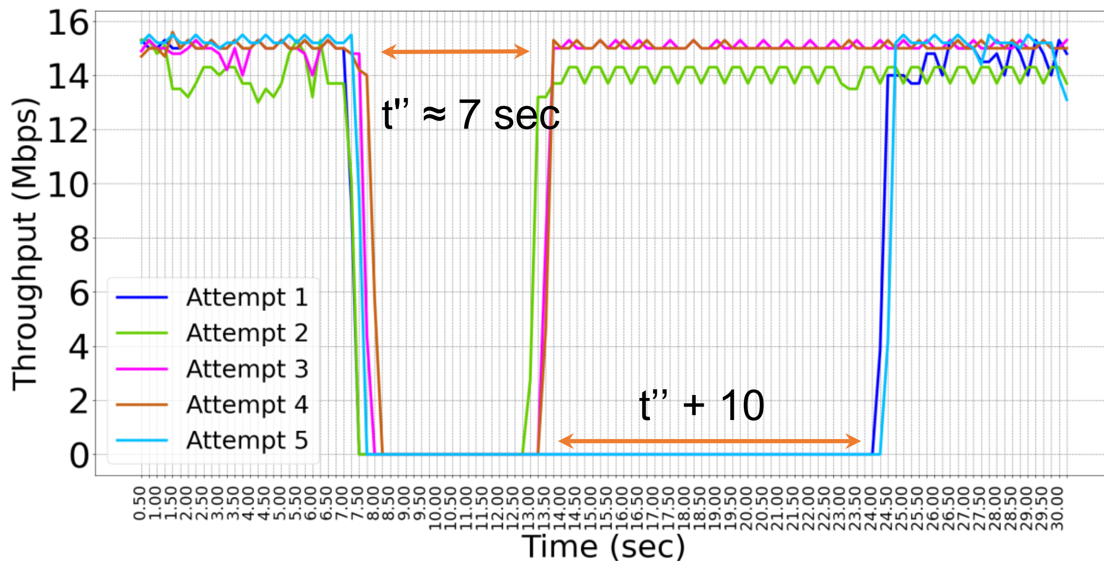


Figure 4.17: Docker forced reconnection time.

By comparing the reconnection time t, t' and t'' of Figure 4.15, Figure 4.16 and Figure 4.17 respectively, some differences on the waiting time until reconnection can be observed. Ordering the deployments from slowest to fastest: baremetal (8 seconds), Docker (7 seconds) and Kubernetes (6.5 seconds). This result can be explained due the last two are containerized and virtualized deployments. Moreover, K8s is the fastest ought to the Calico networking controller.

Lastly, the reconnection time has been measured at different distances, number of PRB and injected bandwidth, but any of those parameters affect the reconnection time.

CONCLUSIONS AND FUTURE WORKS

5.1. Conclusions of the work

As explained in the objectives section, the aim of this project has been to set up a fully virtualized, containerized and distributed open-source-based network that can be deployed on different nodes, separating RAN and core network functions on different nodes. This work has analyzed the performance of the network and the virtualized infrastructure while varying different network parameters and perform a comparison with a state-of-the-art deployment based on Docker tools. To this purpose, this Master thesis was divided in different phases. Firstly, to achieve connectivity in a baremetal deployment, to verify the behavior of the open-source modules in a basic deployment. The second stage consisted on the design of the descriptors and containers required for the deployment of the network using Kubernetes in order to automate the deployment and the management of the aforementioned containers. Lastly, the K8s setup has been compared with existing state-of-the-art projects following the same deployment using Docker containers, without a container orchestration platform.

As is natural, some problems have been faced in the understanding and deploying of some of the software and hardware equipment, but after consulting related works and documentation, all the problems were resolved satisfactorily. Due to this project aims to explain the operation of each of the already mentioned deployments, a section dedicated to these problems has been added to every setup.

From the representation of the data in the section results discussion, some conclusions have been achieved:

- The module *“srsenb”* of the RAN consumes the most memory and CPU resources on every deployment.

- The number of PRB has a direct impact on the system throughput. The more PRB are allocated, the higher the throughput. Moreover, the usage of memory resources is more related to the number of PRB, but the usage of CPU resources is more related to injected bandwidth.

- On average, the same percentage of CPU is used in the three deployments. But the K8s deployment has a 4% more usage of system memory resources than the rest. That can be considered negligible when comparing it to the flexibility, low setting time and ease to deploy that the K8s setup provides.

- Kubernetes orchestration platform facilitates the configuration and deployment of the virtualized network, having the scenario ready in a few seconds. Also, has the fastest reconnecting time in case of an internal failure, due to the cluster networking.

While performing the tests, a connectivity problem was found in the docker-based project that was being used to compare with the K8s deployment. That error was reducing the capabilities of the setup due to the reconnection errors. This issue has been reported to the maintainers of the project.

The principal contributions of this Master Thesis are: (i) the creation of two Docker images that can set up a K8s fully virtualized, containerized and distributed open-source-based network, as well as the Kubernetes descriptors required for the automated deployment of the network on a fully configurable manner (i.e., the network can be deployed on different nodes, separating RAN and CN functions in seconds); (ii) the introduction of a programmable feature in the deployment, to let the end user register its own UE; and (iii) the analysis of the resource performance of the three deployments, giving more emphasis to the K8s scenario.

A forthcoming paper entitled "*Design and evaluation of a kubernetes-based system for distributed open-source cellular networks*" including the main results of this work is being prepared for submission to the IEEE Wireless Communications and Networking Conference to be held in Austin, USA (April 10-13, 2022).

5.2. Future lines of development and research

Taking as a reference the contributions, the conclusions of the analysis and the issues identified in this Master Thesis, it is possible to establish several future lines of development and research.

As it was mentioned in Section 4.2.3, the first improvement that this project has to execute is to add volumes to the Kubernetes deployment. This will solve the issue of disconnection in case of failure of any module related with the storage or registration of the UE since they will persistently storage the UE information and once the failed module is restarted, it will retrieve the necessary information from the volume and regain connectivity. Moreover, as mentioned in Section 4.2.3, to get an energy and resources advantage on the network virtualization, an implementation of an smart resource allocation function has to be investigated. This would assign each module the necessary resources on demand, instead of been running all the time.

One more future line of work would be related to an extension of the analysis performed. On the one hand, it is worthy studying the impact of an increasing number of UEs, together with the already covered amount of traffic in the user plane. On the other hand, in addition to the resource usage of the virtualization infrastructure, it would be interesting to test the energy consumption of the deployment before and after the virtualization.

Another future line of work in this regard would be to test the deployment for 5G NSA and 5G SA equipment's, to verify that the same performance holds. In addition to that, a much broader repertoire (beyond 5) of repetitions of each

experiment must be done. Moreover, to allow full assurance of the results, the use of statistical methods would be a tool to explore.

Furthermore, in this project a containerized separation of the CN and the RAN has been executed, but if needed, a separation of each core function into a container, or in the form of microservices, can be implemented. This would have the advantage of an easy deployment on demand of the functions. Moreover, after researching the state of the art of virtualized networks, some solutions such as [48], defend that the separation the logic of the SGW and PGW of the 4G CN can solve the bottleneck caused by those modules. Extrapolating this idea to the 5GC, creates a great future line of work, which is to test the performance of separating the UPF module form the CN and deploy it with the MEC platform to handle the traffic.

Lastly, to test how the deployment behaves when extrapolated to serverless scenarios on top of K8s, would be interesting.

5.3. Sustainability considerations

This project presents a fully virtualized open-source-based network. In terms of economic impact, the ability to launch the network modules on demand, will translate into a reduction of the resources, that will have an impact in the energy consumption. This will reduce the cost of actual RAN and CN. Moreover, in terms of social impact, the containerization of the software will facilitate the deployment for new users and save them time and effort when trying to integrate or update modules or when deploying the setup. Finally, as shown in [75], in terms environmental impact, the reduction of energy consumption will translate not only in economic impact, but also in a greener world.

5.4. Ethical and security considerations

As it has been mentioned before, this project presents a fully virtualized open-source-based deployment. Ethically, this means that this Master Thesis is part of a worldwide community that works together to accomplish a common goal. Moreover, this software cannot be licensed by any company and anyone can use. Furthermore, this deployment can be run in a generic hardware, avoiding the bad implications of depending on the prices of a few big companies. Taking into account ethical considerations, this technology has no negative contributions. Moreover, in terms of security, a global concern has arisen from the technological community due to the vulnerabilities that a fully virtualized deployment may encounter that were not possible in non-virtualized setups. By researching the state of the art, some papers have been found around this topic. The work in [76] analyzes the impact that an attack to a system component under the responsibility of a given stakeholder may yield to a completely different player in a complex and virtualized system such as the 5G infrastructure. Finally, to solve these threats, authors in [77] propose a secure and trustworthy framework for virtualized networks and software-defined networking.

ACRONYMS

3GPP	3rd Generation Partnership Project
5G-AN	5G Access Network
5GC	5G Core
AF	Application Function
AMF	Access and Mobility Management Function
APN	Access Point Name
AUSF	Authentication Server Function
BSF	Binding Support Function
CI/CD	Continuous Integration/Continuous Deployment
CN	Core Network
CNI	Container Network Interface
CNM	Container Network Model
CUPS	Control User Plane Separation
eMBB	enhanced Mobile Broadband
eNB	eNodeB
EPC	Evolved Packet Core
E-UTRAN	Evolved UMTS Terrestrial Radio Access Network
gNB	Next Generation NodeB
HSS	Home Subscriber Server
IMSI	International Mobile Subscriber Identity
IoT	Internet of Things
K8s	Kubernetes
LTE	Long Term Evolution
MANO	NFV Management and Orchestration
MCC	Mobile Country Code
MEC	Multi-access Edge Computing
MIB	Master Information Block
MIMO	Multiple Input, Multiple Output
MME	Mobile Management Entity
mMTC	massive Machine Type Communication
MNC	Mobile Network Code
MSISDN	Mobile Subscriber Integrated Services Digital Network
NAT	Network Address Translation
NF	Network Function
NFV	Network Function Virtualization
NFVI	NFV Infrastructure
NR	New Radio
NRF	NF Repository Function
NSA	Non Stand Alone
NSSF	Network Slice Selection Function
OAI	Open Air Interface

OFDMA	Orthogonal Frequency Division Multiple Access
OP	Operator Code
OS	Operating System
PCF	Policy Control Function
PCRF	Policy and Charging Rules Function
PDN	Packet Data Network
PGW	Packet Data Network Gateway
PGWC	Packet Gateway Control Plane
PGWU	Packet Gateway User Plane
PLMN	Public Land Mobile Network
PRB	Physical Resource Block
QoS	Quality of Service
RAN	Radio Access Network
RAT	Radio Access Technology
RIC	RAN Intelligent Controller
SA	Stand Alone
SBA	Service Based Architecture
SBI	South-Bound Interface
SCFDMA	Single Carrier Frequency Division Multiple Access
SDN	Software Defined Network
SD-RAN	Software-Defined RAN
SGW	Serving Gateway
SGWC	Serving Gateway Control Plane
SGWU	Serving Gateway User Plane
SIB	System Information Block
SIM	Subscriber Identity Module
SMF	Session Management Control Function
srsRAN	Software Radio Systems RAN
TAC	Tracking Area Code
TCP	Transport Control Protocol
UDM	Unified Data Management
UDP	User Datagram Protocol
UDR	Unified Data Repository
UE	User Equipment
UPF	User Plane Function
URLLC	Ultra Reliable Low Latency
VIM	Virtualized Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Functions
VNFM	VNF Manager
VNFO	NFV Orchestrator

ANNEX I: INSTALLING SRSLTE AND OPEN5GS

This section explains the installation process of srsLTE and Open5GS with all the necessary dependencies in Ubuntu 20.04. To do so, the following commands in the console terminal must be run:

```
# Installing srsLTE

sudo add-apt-repository ppa:softwareradiosystems/srsran
sudo apt-get update
sudo apt-get install srsran –y
```

Once the srsLTE process is completed, MongoDB has to be installed for Open5GS and Open5GS Web-UI to work. The next commands are used to install all the software's with the necessary dependencies:

```
# Installing MongoDB

sudo apt-get update
sudo apt-get install mongodb

# Installing Open5GS

sudo apt-get install software-properties-common
sudo add-apt-repository ppa:open5gs/latest
sudo apt-get update
sudo apt-get install open5gs

# Installing the Open5GS Web-UI

curl -fsSL https://deb.nodesource.com/setup_14.x | sudo -E bash -
sudo apt install nodejs
curl  -fsSL  https://open5gs.org/open5gs/assets/webui/install  |
sudo -E bash -
```

ANNEX II: NODES PREPARATION TO DEPLOY THE K8S CLUSTER

This section will be focused on the installation of the necessary software and dependencies required to deploy the K8s cluster. Before starting to set the K8s cluster, some modifications have to be made in **all the nodes**:

- Set hostnames: ensure that all of the nodes have a unique hostname. In this scenario, the hostnames are kubernetes-master, kubernetes-worker1 and kubernetes-worker2. The following command can be used in each node to set the host names:

```
sudo hostnamectl set-hostname <chosen_name>
```

  The changes will not be noticeable in the terminal until it is restarted. Then, some entries have to be added into the "*/etc/hosts*" file. Those entries are:

```
<IP address master>          chosen_name_master
<IP address worker1>         chosen_name_worker1
<IP address worker2>         chosen_name_worker2
```

  In this project are:

```
10.43.79.43                  kubernetes-master
10.43.79.1                   kubernetes-worker1
10.43.79.7                   kubernetes-worker2
```

- Install ssh (optional): ssh is a tunneling tool to manage all the nodes from the master console terminal and facilitates the management of the cluster from a single node. To get it install, the following commands are required:

```
sudo apt-get install shh
sudo systemctl enable –-now ssh
```

  Once, it is installed in all the nodes, the following command can be used to get access and control over the other nodes:

```
shh <user_name>@<chosen_name_workerX>
For this project: shh javi@kubernetes-worker1
```

- Installing Docker: Connect (or do it manually) to each node and run the following commands:

```
sudo apt-get update
sudo apt-get install docker.io
```

  Now the Docker service can be enabled and its status can be verified:

```
sudo systemclt enable docker.service --now
sudo systemctl status docker
```

If everything has been done correctly, it should be active and running.

- Disable swap and enable IP forwarding: Kubernetes will refuse to work if the system used is using swap memory. Before proceeding any further, it has to be ensured that the master and worker nodes have swap memory disabled with this command:

```
sudo swapoff -a
```

This will disable swap memory until the system reboots. To make this change persistent it is needed to edit the "*/etc/fstab*" file and comment the *"/swap"* line.

To enable IP forwarding permanently, the file *"/etc/sysctl.conf*" must be edited to uncomment the "*net.ipv4.ip_forward=1"* line. To make sure it worked, the following command must be run:

```
sudo sysctl -p
Console output: net.ipv4.ip_forward = 1
```

- Install kubectl, kubelet and kubeadm: to finish the installation process, the following commands must be introduced in each node:

```
sudo apt-get install -y apt-transport-https curl

curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add

sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"

sudo apt update

sudo apt install -y kubelet kubeadm kubectl
```

ANNEX III: DEPLOYMENT OF THE K8S CLUSTER

This annex explains the procedure to deploy the Kubernetes cluster in detailed.

All the following commands are run in the master node:

- Initialize the K8s cluster, assigning the pod network an IP range:

```
sudo kubeadm init
```

The console returns a set of commands and an identification token for the worker nodes to join the cluster. Those commands are required to configure kubectl and to get the token. This token is a private key that the master node can share with other worker nodes to authenticate them and attach them to the K8s cluster. The console output must be similar to:

```
Commands:

mkdir -p $HOME/.kube`
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config`
sudo chown $(id -u):$(id -g) $HOME/.kube/config`

Token:

kubeadm join 10.43.79.43:6443 --token b4sfnc.53ifyuncy017cnqq --
discovery-token-ca-cert-hash
sha256:5078c5b151bf776c7d2395cdae08080faa6f82973b989d29caaa4d58c28d0e4
```

- The Tigera Calico operator can be installed by running:

```
kubectl create -f https://docs.projectcalico.org/manifests/
tigera-operator.yaml`
```

- After that, Calico itself can be installed by creating the necessary custom resource:

```
kubectl create -f https://docs.projectcalico.org/manifests/
custom-resources.yaml
```

- Once this is done, the previous noted token must be introduced in all worker nodes to include them into the cluster:

```
sudo kubeadm join 10.43.79.43:6443 --token
b4sfnc.53ifyuncy017cnqq --discovery-token-ca-cert-hash
sha256:5078c5b151bf776c7d2395cdae08080faa6f82973b989d29caaa4d58c28d0e4
```

To check that the deployment of the cluster has been done correctly and that the worker nodes have been well added to the cluster, the next two commands must be run, analyzing the obtained output:

```
kubectl get nodes -o wide
```

Table Annex.1: K8s cluster nodes information.

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP
kubernetes-master	Ready	control-plane,master	4m	v1.21.2	10.43.79.43
kubernetes-worker1	Ready	<none>	46s	v1.21.2	10.43.79.1
kubernetes-worker2	Ready	<none>	59s	v1.21.2	10.43.79.7

```
kubectl get pods -n calico-system -o wide
```

Table Annex.2: Calico pods information.

NAME	READY	STATUS	RESTARTS	IP	NODE
calico-kube-controllers-7f58dbcbbd-zn8qw	1/1	Running	0	10.43.79.43	kubernetes-master
calico-node-7s4gw	1/1	Running	0	10.43.79.43	kubernetes-master
calico-node-nvnph	1/1	Running	1	10.43.79.7	kubernetes-worker2
calico-node-xsx2g	1/1	Running	0	10.43.79.1	kubernetes-worker1
calico-typha-b6f4d48cd-48rbl	1/1	Running	1	10.43.79.1	kubernetes-worker1
calico-typha-b6f4d48cd-9nvpw	1/1	Running	0	10.43.79.7	kubernetes-worker2
calico-typha-b6f4d48cd-zqmf7	1/1	Running	0	10.43.79.43	kubernetes-master

If all the nodes in Table 3.1 have the status "*Ready*" and all the pods of Table 3.2 have the status "*Running*", the cluster has been properly created.

ANNEX IV: DOCKER IMAGES

This section presents the content of the necessary files for the creation of both Docker images: srsLTE and Open5GS.

srsLTE

Dockerfile:

This file creates the Docker image.

```
FROM ubuntu:20.04
MAINTAINER Javier Palomares <japato.96@gmail.com>
ENV DEBIAN_FRONTEND noninteractive
USER root

# Dependencies needed for the UHD driver for the USRP B210
RUN apt-get update && \
   apt-get -yq install cmake git iputils-ping nano libfftw3-dev libmbedtls-dev libboost-
program-options-dev libconfig++-dev libsctp-dev libuhd-dev usbutils iproute2

# Fetching empower-enb-agent
RUN git clone https://github.com/5g-empower/empower-enb-agent.git
RUN cd empower-enb-agent && \
   cmake -DCMAKE_BUILD_TYPE=Release . && \
   make && \
   make install

# Fetching srsRAN
RUN git clone https://github.com/5g-empower/srsRAN.git && \
   cd srsRAN && \
   git checkout agent && \
   mkdir build && \
   cd build && \
   cmake ../ && \
   make

# Running the image needed for the UHD driver for the USRP B210
RUN ./usr/lib/uhd/utils/uhd_images_downloader.py

ADD conf/enb.conf /etc/srsran/
ADD conf/drb.conf /etc/srsran/
ADD conf/rr.conf /etc/srsran/
ADD conf/sib.conf /etc/srsran/

# Add Kubernetes config, setup and launch scripts
ADD dns_replace.sh /
ADD config.sh /
ADD launcher.sh /

# Run the launcher script
ENTRYPOINT ["/launcher.sh"]
```

dns_replace.sh:

This code assigns dynamically the IP addresses of the mme_addr, gtp_bind_addr and s1c_bind_addr, to make the connections within the RAN and with the CN.

```
if [ -z "$empower_pod_addr" ]; then

  while [ -z $(getent hosts runtime-service | awk '{ print $1 }') ]
  do
    echo "Waiting for the 5G-EmPOWER Runtime to come up..."
    sleep 10
  done

  echo "5G-EmPOWER Runtime service found"
  EMPOWER_POR_ADDR=$(getent hosts empower-service | awk '{ print $1 }')

else

  EMPOWER_POR_ADDR=$empower_pod_addr

fi

if [ -z "$epc_pod_addr" ]; then

  while [ -z $(getent hosts epc-service | awk '{ print $1 }') ]
  do
    echo "Waiting for the EPC to come up..."
    sleep 10
  done

  echo "EPC service found"
  EPC_POD_ADDR=$(getent hosts epc-service | awk '{ print $1 }')

else
  EPC_POD_ADDR=$epc_pod_addr
fi

if [ -z "$local_pod_addr" ]; then
  LOCAL_POD_ADDR=$(ip route get 1 | awk '{print $(NF-2);exit}')
else
  LOCAL_POD_ADDR=$local_pod_addr
fi

#Assignation of the new values to the variables
sed -i 's/ENB_ID_REPLACE/'$enb_id'/g' /etc/srsran/enb.conf
sed -i 's/EPC_REPLACE/'"$EPC_POD_ADDR"'/g' /etc/srsran/enb.conf
sed -i 's/LOCAL_REPLACE/'$LOCAL_POD_ADDR'/g' /etc/srsran/enb.conf
sed -i 's/EMPOWER_REPLACE/'$EMPOWER_POR_ADDR'/g' /etc/srsran/enb.conf
```

config.sh:

This file provides the configurable feature.

```
#!/bin/bash

#Defines new variables that store the values introduced in the descriptor (.yaml)
N_ENB_MCC=$enb_mcc
N_ENB_MNC=$enb_mnc
N_ENB_PRB=$enb_prb

#Changes the value of the previous parameter to the new one
sed -i 's/ENB_MCC/'$N_ENB_MCC'/g' /etc/srsran/enb.conf
sed -i 's/ENB_MNC/'$N_ENB_MNC'/g' /etc/srsran/enb.conf
sed -i 's/ENB_PRB/'$N_ENB_PRB'/g' /etc/srsran/enb.conf
```

launcher.sh:

This file executes the RAN.

```
#!/bin/bash

_term() {
  echo "Caught SIGTERM signal!"
  kill -TERM "$child"
}

trap _term SIGTERM

env

#Launches the previous scripts and the command to start the srsENB
./config.sh
./dns_replace.sh
./srsRAN/build/srsenb/src/srsenb &

child=$!

wait "$child"
```

conf/enb.conf:

The full file it is too long, so only the parts that have been modified from the original “*srsLTE/enb.conf*” file is going to be shown:

```
#####################################################################
#                 srsENB configuration file
#####################################################################

#####################################################################
# Agent configuration
#
# address:              Controller IP address (default 127.0.0.1)
# port:                 Controller port (default: 5533)
# delay:                Hello period (default: 2000)
#
#####################################################################
[agent]
address = EMPOWER_REPLACE
port = 5533
delay = 2000

#####################################################################
# eNB configuration
#
# enb_id:               20-bit eNB identifier.
# mcc:                  Mobile Country Code
# mnc:                  Mobile Network Code
# mme_addr:             IP address of MME for S1 connnection
# gtp_bind_addr:        Local IP address to bind for GTP connection
# gtp_advertise_addr:   IP address of eNB to advertise for DL GTP-U Traffic
# s1c_bind_addr:        Local IP address to bind for S1AP connection
# n_prb:                Number of Physical Resource Blocks (6,15,25,50,75,100)
# tm:                   Transmission mode 1-4 (TM1 default)
# nof_ports:            Number of Tx ports (1 port default, set to 2 for TM2/3/4)
#
#####################################################################
# Parameters that have been introduced to make the RAN configurable
[enb]
enb_id = ENB_ID_REPLACE
mcc = ENB_MCC
mnc = ENB_MNC
mme_addr = EPC_REPLACE
gtp_bind_addr = LOCAL_REPLACE
s1c_bind_addr = LOCAL_REPLACE
n_prb = ENB_PRB
#tm = 4
#nof_ports = 2
```

Open5GS

Dockerfile:

This file creates the Docker image.

```
FROM ubuntu:20.04
MAINTAINER Javier Palomares <japato.96@gmail.com>
ENV DEBIAN_FRONTEND noninteractive
USER root

# Dependencies for the Open5gs
RUN apt-get update && \
    apt-get upgrade -y && \
    apt-get install -y --no-install-recommends \
        python3-pip python3-setuptools python3-wheel \
        ninja-build build-essential flex \
        bison git iputils-ping \
        nano libsctp-dev libgnutls28-dev \
        libgcrypt-dev libssl-dev libidn11-dev \
        libmongoc-dev libbson-dev libyaml-dev \
        libnghttp2-dev libmicrohttpd-dev libcurl4-gnutls-dev \
        meson netcat iproute2 \
        wget unzip iptables

# Fetching Open5GS
RUN git clone https://github.com/open5gs/open5gs && cd /open5gs && meson build --
prefix=`pwd`/install && ninja -C build

RUN cd /open5gs/build && ninja install

# Coping configuration files needed
COPY conf/* /open5gs/install/etc/open5gs/

# Solve the shared libraries problem
RUN sh -c "echo /open5gs/install/lib/x86_64-linux-gnu > /etc/ld.so.conf.d/open5gs.conf"
RUN ldconfig

# Add Kubernetes config, setup and launch scripts
ADD setup.sh /
ADD launcher.sh /
ADD config.sh /

# Run the launcher script
ENTRYPOINT ["/launcher.sh"]
```

config.sh:

This file provides the configurable feature.

```
#!/bin/bash

#Defines new variables that store the values introduced in the descriptor (.yaml)
N_MCC=$mcc
N_MNC=$mnc
N_TAC=$tac

#Changes the value of the previous parameter to the new one
sed -i 's/MCC/'$N_MCC'/g' /open5gs/install/etc/open5gs/mme.yaml
sed -i 's/MNC/'$N_MNC'/g' /open5gs/install/etc/open5gs/mme.yaml
sed -i 's/TAC/'$N_TAC'/g' /open5gs/install/etc/open5gs/mme.yaml
```

conf/mme.yaml:

The full file it is too long, so only the parts that have been modified from the original "*Open5GS/mme.yaml*" file is going to be shown:

```
mme:
  freeDiameter: /open5gs/install/etc/freeDiameter/mme.conf
  s1ap:
  gtpc:
    - addr: 127.0.0.2
  gummei:
    plmn_id:
      mcc: MCC
      mnc: MNC
    mme_gid: 2
    mme_code: 1
  tai:
    plmn_id:
      mcc: MCC
      mnc: MNC
    tac: TAC
  security:
      integrity_order : [ EIA2, EIA1, EIA0 ]
      ciphering_order : [ EEA0, EEA1, EEA2 ]
  network_name:
      full: Open5GS
  mme_name: open5gs-mme0
```

launcher.sh:

This file executes the CN.

```
#!/bin/bash

_term() {
  echo "Caught SIGTERM signal!"
  kill -TERM "$child"
}

trap _term TERM

echo -e "\n\n---------- ENV VARIABLES ----------"
env

#Launches the config script
./config.sh

until nc -z localhost 27017
do
    echo "waiting for mongodb to come up..."
    sleep 2
done

sleep 10

#Launches the setup script
/setup.sh

#Launches all the modules of Open5GS
/open5gs/install/bin/open5gs-mmed -D
/open5gs/install/bin/open5gs-sgwcd -D
/open5gs/install/bin/open5gs-smfd -D
/open5gs/install/bin/open5gs-amfd -D
/open5gs/install/bin/open5gs-sgwud -D
/open5gs/install/bin/open5gs-upfd -D
/open5gs/install/bin/open5gs-hssd -D
/open5gs/install/bin/open5gs-pcrfd -D
/open5gs/install/bin/open5gs-nrfd -D
/open5gs/install/bin/open5gs-ausfd -D
/open5gs/install/bin/open5gs-udmd -D
/open5gs/install/bin/open5gs-pcfd -D
/open5gs/install/bin/open5gs-nssfd -D
/open5gs/install/bin/open5gs-bsfd -D
/open5gs/install/bin/open5gs-udrd
```

ANNEX V: KUBERNETES DESCRIPTOR FILE

This annex contains the file used to create the two-nodes and tree-nodes K8s deployment explained in this Master Thesis.

K8s_deployment.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: epc
  labels:
    app: epc
spec:
  containers:
  - name: open5gs
    image: javipalomares/open5gs:latest
    env:
    - name: mcc
      value: "001"
    - name: mnc
      value: "03"
    - name: tac
      value: "7"
    securityContext:
      privileged: true
  - name: open5gs-webui
    image: snslab/open5gs-webui:latest
  - name: mongodb
    image: mongo
  - name: mongo-express
    image: mongo-express
    env:
    - name: ME_CONFIG_MONGODB_SERVER
      value: "localhost"
  nodeSelector:
    IDname: kubernetes-worker1
---
apiVersion: v1
kind: Service
metadata:
  name: epc-mongo-express-service
spec:
  selector:
    app: epc
  type: NodePort
  ports:
  - name: web-ui
    protocol: TCP
    port: 8081
    targetPort: 8081
    nodePort: 30000
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: epc-open5gs-webui-service
spec:
  selector:
    app: epc
  type: NodePort
  ports:
  - name: web-ui
    protocol: TCP
    port: 3000
    targetPort: 3000
    nodePort: 30001
---
apiVersion: v1
kind: Pod
metadata:
  name: srsenb
  labels:
    app: srsenb
spec:
  containers:
    - name: srsenb
      image: javipalomares/srslte:latest
      env:
      - name: enb_mcc
        value: "001"
      - name: enb_mnc
        value: "03"
      - name: enb_id
        value: "0x19B"
      - name: enb_prb
        value: "75"
      - name: empower_pod_addr
        value: "127.0.0.1"
      securityContext:
        privileged: true
  nodeSelector:
    IDname: kubernetes-worker2
```

ANNEX VI: CODE TO CLEAN AND PLOT THE RESULTS

In this annex is going to be included the used code to clean, format and plot the raw data extracted from Iperf3.

Throughput average

extract_average.py:

```
#File to extract the average throughput data
import pandas as pd


def format_line(text):
    text = text.replace("[  6]   ", "")
    text = text.replace("- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -", "")
    text = text.replace("[  5]   ", "")
    text = text.replace("  sec ", "")
    text = text.replace(" MBytes ", "")
    text = text.replace(" Mbits/sec ", "")
    text = text.replace(" Kbits/sec ", "")
    text = text.replace(" ms ", "")
    text = text.replace("  receiver", "")
    text = text.replace("  ", " ")
#    text = text.replace(".", ",")
    return text


def convert(lst):
    return lst[0].split()


def transform_csv(path, name):
    transfer = []
    bit_rate_real = []
    jitter = []

    df = pd.DataFrame()

    with open('{}/{}.txt'.format(path, name), 'r') as in_file:
        for myline in in_file:
            lines = convert([myline])
            transfer.append(lines[1])
            bit_rate_real.append(lines[2])
            jitter.append(lines[3])


        df['Transfer(KBytes)'] = transfer
        df['Bitrate_real(Mbits/sec)'] = bit_rate_real
        df['Jitter(ms)'] = jitter
```

```
        df.to_csv('{}/{}.csv'.format(path, name))


def create_file(deployment, folder_origin, origin, destination, format_dest):
    myLines = []
    listNum = []
    cont = 0
    with open('Data/{}/{}/{}.txt'.format(deployment, folder_origin, origin), 'r') as myfile:
        for myline in myfile:
            myLines.append(myline.rstrip('\n'))
    for element in myLines:
        if element == "- - - - - - - - - - - - - - - - - - - - - - - - -":
            listNum.append(cont)
        cont += 1

    with open('Results/{}/Average/{}/{}.{}'.format(deployment, folder_origin, destination,
format_dest), 'w+') as data:
        for num in listNum:
            formated_line = format_line(myLines[num + 2])
            data.write(formated_line + "\n")

    transform_csv('Results/{}/Average/{}/'.format(deployment, folder_origin),
'{}'.format(destination))


deploy = ["Local", "Kubernetes", "Kubernetes2"]
PRB = [25, 50, 75]
dist = [1, 3, 6, 10, 15, 20]

for i in deploy:
    for j in PRB:
        for k in dist:
            create_file(i, "{}PRB".format(j), "n_{}mPRB{}".format(k, j), "{}m".format(k), "txt")
```

clean_average.py:

```
#File to clean the average throughput data
import numpy as np
import pandas as pd
import scipy.stats


# Set all in Mbps
def check_Mbps(array):
    for i in range(0, len(array)):
        if isinstance(array[i], str):
            array[i] = float(array[i])
        if array[i] > 900:
            array[i] = array[i] / 1000
    return array


# Average the measurements and get the 0,95 confidence interval
```

```
def get_mean_confidence_interval(data, average, sup, inf):
    cont = 0
    for i in range(0, int(len(data) / 5)+1):
        m, se = np.mean(data[i * cont:i * cont + 5]), scipy.stats.sem(data[i * cont:i * cont +
5])
        h = se * scipy.stats.t.ppf((1 + 0.95) / 2., len(data) - 1)
        cont += 1

        average.append(m)
        sup.append(m + h)
        inf.append(m - h)

    return average, sup, inf


def store_values(deploy, n_PRB, dist, average, sup, inf):
    df['Average_{}_{}PRB_{}m'.format(deploy, n_PRB, dist)] = average
    df['Sup_{}_{}PRB_{}m'.format(deploy, n_PRB, dist)] = sup
    df['Inf_{}_{}PRB_{}m'.format(deploy, n_PRB, dist)] = inf

    df.to_csv('Results/Clean data/{}/clean_average2.csv'.format(deploy))


def obtain_data(deploy, n_PRB, dist):
    average = []
    sup = []
    inf = []

    data = pd.read_csv("Results/{}/Average/{}PRB/{}m.csv".format(deploy, n_PRB, dist))
    bitRate = data['Bitrate_real(Mbits/sec)']

    # Set all in Mbps
    bitRate = check_Mbps(bitRate)

    bitRate = bitRate[0:29]
    average, sup, inf = get_mean_confidence_interval(bitRate, average, sup, inf)
    store_values(deploy, n_PRB, dist, average, sup, inf)


deployment = ["Local", "Kubernetes", "Kubernetes2"]
PRB = [25, 50, 75]
dist = [1, 3, 6, 10, 15, 20]

for i in deployment:
    df = pd.DataFrame()
    for j in PRB:
        for k in dist:
            obtain_data(i, j, k)
```

plot_avera ge.py:

```
import pandas as pd
import matplotlib.pyplot as plt


def plot_Same_PRB(deployment):
    colors = ['#0000FF', '#66CD00', '#FF00FF', '#C76114', '#00BFFF', 'k']
    cnt_plots = 0

    for k in num_PRB:
        plt.figure(figsize=(15, 10))
        cnt_plots += 1
        cont = 0
        for s in distances:
            data_plot = data['Average_{}_{}PRB_{}m'.format(deployment, k, s)]
            plt.plot(range_plot, data_plot, colors[cont], linewidth=4, label='{}m'.format(s))
            data_error = data['Sup_{}_{}PRB_{}m'.format(deployment, k, s)] -
data['Average_{}_{}PRB_{}m'.format(deployment, k, s)]
            plt.errorbar(range_plot, data_plot, linewidth=2, marker="o", yerr=data_error,
fmt=(colors[cont]), capsize=10)
            #plt.title('{} deployment with {}PRB'.format(deployment, k))
            plt.grid(color='tab:gray', linestyle='--', linewidth=0.5)
            plt.xlabel('Throughput introduced (Mbps)', fontsize=22)
            plt.xticks(range(0, 151, 25), fontsize=20)
            plt.ylabel('throughput (Mbps)', fontsize=22)
            plt.yticks(range(0, 11), fontsize=20)
            plt.legend(loc=2, prop={'size': 15})
            cont += 1
        plt.show()


def plot_Same_Distance(deployment):
    colors = ["#1E90FF", "r", "k"]
    cnt_plots = 0

    for i in distances:
        plt.figure(figsize=(15, 10))
        cnt_plots += 1
        cont = 0
        for j in num_PRB:
            data_plot = data['Average_{}_{}PRB_{}m'.format(deployment, j, i)]
            data_error = data['Sup_{}_{}PRB_{}m'.format(deployment, j, i)]-
data['Average_{}_{}PRB_{}m'.format(deployment, j, i)]
            plt.errorbar(range_plot, data_plot, linewidth=2, marker="o", yerr=data_error,
fmt=colors[cont], capsize=10)
            plt.plot(range_plot, data_plot, colors[cont], linewidth=4, label='{}PRB'.format(j))
            #plt.title('{} deployment at {}m'.format(deployment, i))
            plt.grid(color='tab:gray', linestyle='--', linewidth=0.5)
            plt.xlabel('Throughput introduced (Mbps)', fontsize=22)
            plt.xticks(range(0, 151, 25), fontsize=20)
            plt.ylabel('Throughput (Mbps)', fontsize=22)
            plt.yticks(range(0, round(max(data_plot))+1, 5), fontsize=20)
            plt.legend(loc=2, prop={'size': 15})
```

```
            cont += 1
        plt.show()


def plot_comparing_deployments(prb, dist):
    colors = ["#1E90FF", "r", "k"]

    cont = 0
    data1 = pd.read_csv("Results/Clean data/Local/clean_average.csv")
    data2 = pd.read_csv("Results/Clean data/Kubernetes/clean_average.csv")
    data3 = pd.read_csv("Results/Clean data/Kubernetes2/clean_average.csv")

    data_local_plot = data1['Average_Local_{}PRB_{}m'.format(prb, dist)]
    data_local_error = data1['Sup_Local_{}PRB_{}m'.format(prb, dist)] - data1[
        'Average_Local_{}PRB_{}m'.format(prb, dist)]
    data_k8s_plot = data2['Average_Kubernetes_{}PRB_{}m'.format(prb, dist)]
    data_k8s_error = data2['Sup_Kubernetes_{}PRB_{}m'.format(prb, dist)] - data2[
        'Average_Kubernetes_{}PRB_{}m'.format(prb, dist)]
    data_k8s2_plot = data3['Average_Kubernetes2_{}PRB_{}m'.format(prb, dist)]
    data_k8s2_error = data3['Sup_Kubernetes2_{}PRB_{}m'.format(prb, dist)] - data3[
        'Average_Kubernetes2_{}PRB_{}m'.format(prb, dist)]

    list_plot = [data_local_plot, data_k8s_plot, data_k8s2_plot]
    list_error = [data_local_error, data_k8s_error, data_k8s2_error]

    plt.figure(figsize=(15, 10))
    for i in range(0, 3):

        plt.errorbar(range_plot, list_plot[i], linewidth=2, marker="o", yerr=list_error[i],
fmt=colors[cont], capsize=10)
        plt.plot(range_plot, list_plot[i], colors[cont], linewidth=4, label='{} {}PRB
{}m'.format(deploys[i], prb, dist))
        cont += 1

        #plt.title('Comparing deployments')
        plt.grid(color='tab:gray', linestyle='--', linewidth=0.5)
        plt.xlabel('Throughput introduced (Mbps)', fontsize=22)
        plt.xticks(range(0, 151, 25), fontsize=20)
        plt.ylabel('Throughput (Mbps)', fontsize=22)
        plt.yticks(range(0, 11), fontsize=20)
        plt.legend(loc=2, prop={'size': 15})
    plt.show()


deploys = ["Local", "Kubernetes", "Kubernetes2"]
num_PRB = [25, 50, 75]
range_plot = [1, 25, 50, 75, 100, 150]
distances = [1, 3, 6, 10, 15, 20]

for i in deploys:
    data = pd.read_csv("Results/Clean data/{}/clean_average.csv".format(i))
    plot_Same_PRB(i)
    # plot_Same_Distance(i)

# plot_comparing_deployments(25, 1)
```

Tempor al

Scripts to extract, clean and plot the temporal representation of the data.

extract_temporal.py:

```
def format_line(text):
    text = text.replace("[  5]   ", "")
    text = text.replace("[  5]  ", "")
    text = text.replace("[  6]  ", "")
    text = text.replace("  sec ", "")
    text = text.replace(" MBytes ", "")
    text = text.replace(" KBytes ", "")
    text = text.replace(" Bytes ", "")
    text = text.replace(" Mbits/sec ", "")
    text = text.replace(" Kbits/sec ", "")
    text = text.replace(" bits/sec ", "")
    text = text.replace(" ms ", "")
    text = text.replace("  receiver", "")
    text = text.replace("  ", " ")
    text = text.replace("[ ID] Interval      Transfer   Bitrate     Jitter  Lost/Total Datagrams",
"")
    text = text.replace("[SUM] 0.0-15.2 sec 1 datagrams received out-of-order", "")
    text = text.replace("[SUM] 0.0-15.3 sec 1 datagrams received out-of-order", "")
    text = text.replace("WARNING: Size of data read does not correspond to offered
length", "")
    text = text.replace("- - - - - - - - - - - - - - - - - - - - - - - - -", "")
    text = text.replace("-----------------------------------------------------------", "")
    text = text.replace("Server listening on 5000", "")
    #    text = text.replace(".", ",")
    return text


def create_file(deployment, folder_origin, origin, destination, format_dest):
    myLines = []

    with open('Data/{}/{}/{}.txt'.format(deployment, folder_origin, origin), 'r') as myfile:
        for myline in myfile:
            myLines.append(myline.rstrip('\n'))

    with open('Results/{}/Temporal/{}/{}.{}'.format(deployment, folder_origin, destination,
format_dest), 'w+') as data:
        for element in myLines:
            not_found = 0
            for word in bad_words:
                if element.__contains__(word):
                    break
                else:
                    not_found += 1

                if not_found == len(bad_words):
                    formated_line = format_line(element)
                    data.write(formated_line + "\n")
```

```
bad_words = ["Accepted", "omitted", "local"]
deploy = ["Local", "Kubernetes", "Kubernetes2"]
PRB = [25, 50, 75]
dist = [1, 3, 6, 10, 15, 20]

for i in deploy:
    for j in PRB:
        for k in dist:
            create_file(i, "{}PRB".format(j), "n_{}mPRB{}".format(k, j), "{}m".format(k), "txt")
```

clean_temporal.py:

```
import pandas as pd


def convert(lst):
    return lst[0].split()


def transform_csv(path, file):
    cont = 0
    n_parse = 0
    df = pd.DataFrame()
    interval = []
    transfer = []
    bit_rate_real = []
    jitter = []

    with open('{}/{}m.txt'.format(path, file), 'r') as in_file:
        for myline in in_file:
            if myline == "\n":
                continue
            else:
                if cont == 61:
                    n_parse += 1
                    df['{} Interval(sec)'.format(n_parse)] = interval[0:60]
                    df['{} Transfer(KBytes)'.format(n_parse)] = transfer[0:60]
                    df['{} Bitrate_real(Mbits/sec)'.format(n_parse)] = bit_rate_real[0:60]
                    df['{} Jitter(ms)'.format(n_parse)] = jitter[0:60]

                    interval = []
                    transfer = []
                    bit_rate_real = []
                    jitter = []
                    cont = 0

                else:
                    lines = convert([myline])
                    interval.append(lines[0])
                    transfer.append(lines[1])
                    bit_rate_real.append(lines[2])
                    jitter.append(lines[3])
                    cont += 1
```

```
        df.to_csv('{}/{}m.csv'.format(path, file))
bad_words = ["Accepted", "omitted", "local"]
deploy = ["Local", "Kubernetes", "Kubernetes2"]
PRB = [25, 50, 75]
dist = [1, 3, 6, 10, 15, 20]

for i in deploy:
    for j in PRB:
        for k in dist:
            transform_csv('Results/{}/Temporal/{}PRB/'.format(i, j), k)
```

plot_temporal.py:

```
import pandas as pd
import matplotlib.pyplot as plt


def plot_dist(data, mbps, dep):

    colors = ['#0000FF', 'g', '#FF00FF', '#C76114', '#00BFFF', 'k']
    for rb in num_PRB:
        cont = 0
        plt.figure(figsize=(15, 10))
        for i in distances:
            data_plot = data['Temporal_{}_{}PRB_{}m'.format(dep, rb, i)]
            data_error = data['Temporal_{}_{}PRB_{}m'.format(dep, rb, i)] - \
                        data['Sup_{}_{}PRB_{}m'.format(dep, rb, i)]
            #plt.errorbar(range_plot, data_plot, linewidth=2, marker="o", yerr=data_error,
fmt=colors[cont], capsize=10)
            plt.plot(range_plot, data_plot, colors[cont], linewidth=3, label='{}m'.format(i))
            plt.title('{}Mbps {} deployment with {}PRB'.format(mbps, dep, rb))
            plt.grid(color='tab:gray', linestyle='--', linewidth=0.5)
            plt.xlabel('Time (sec)', fontsize=22)
            plt.yticks(fontsize=15)
            plt.xticks(range_plot, labels, rotation=90, fontsize=12,
horizontalalignment='center')
            plt.ylabel('Throughput (Mbps)', fontsize=22)
            plt.legend(loc=1, prop={'size': 20})
            cont += 1
        plt.show()


deploys = ["Local", "Docker", "Kubernetes"]
num_PRB = [25, 50, 75]
range_plot = range(0, 60, 1)
distances = [1, 3, 6, 10, 15, 20]
mbps = [1, 25, 50, 75, 100, 150]
labels = []
sum = 0

for i in range(0, 60):
    labels.append('{:.2f}'.format(sum+0.25))
    sum += 0.25


for dep in deploys:
    for i in mbps:
        data = pd.read_csv("Results/Clean
data/{}/clean_temporal_{}Mbps.csv".format(dep, i))
        plot_dist(data, i, dep)
```

Resources

Scripts to extract, clean and plot the resources usage data.

extract_resources.py:

```
import os
import pandas as pd

containers = ['mme', 'sgwu', 'amf', 'sgwc', 'upf', 'bsf', 'udr', 'ausf', 'pcf', 'smf', 'udm', 'nssf',
'webui', 'hss',
              'pcrf', 'mongo', 'nrf']

for i in range(25, 76, 25):
    os.system('cat Data/Local/resources/{}_resources_local.txt | grep open5gs > '
              'Results/Local/Resources/{}_clean_open5gs_resources.txt'.format(i, i))

    os.system('cat Data/Local/resources/{}_resources_local.txt | grep srsenb > '
              'Results/Local/Resources/{}_clean_srsenb_resources.txt'.format(i, i))

    os.system('cat Data/Local/resources/{}_resources_local.txt | grep mongodb > '
              'Results/Local/Resources/{}_clean_mongo_resources.txt'.format(i, i))

    for cont in containers:
        os.system('cat Data/Docker/resources/{}_resources_docker.txt | grep {} > '
                  'Results/Docker/Resources/{}_clean_open5gs-{}_resources.txt'.format(i,
cont, i, cont))

    os.system('cat Data/Docker/resources/{}_resources_docker.txt | grep srsenb > '
              'Results/Docker/Resources/{}_clean_srsenb_resources.txt'.format(i, i))

    os.system('cat Data/Kubernetes/resources/{}_resources_kub.txt | grep open5gs > '
              'Results/Kubernetes/Resources/{}_clean_open5gs_resources.txt'.format(i, i))

    os.system('cat Data/Kubernetes/resources/{}_resources_kub.txt | grep srsenb > '
              'Results/Kubernetes/Resources/{}_clean_srsenb_resources.txt'.format(i, i))


def convert(lst):
    return lst[0].split()
def transform_csv(path, name, case):

    if case == 'Local':
        PR = []
        NI = []
        RES = []
        VIRT = []
        SHR = []
        CPU = []
        MEM = []

        df = pd.DataFrame()

        with open('{}/{}.txt'.format(path, name), 'r') as in_file:
```

```
        for myline in in_file:
            if myline == "\n":
                continue
            else:
                lines = convert([myline])
                PR.append(lines[2])
                NI.append(lines[3])
                VIRT.append(lines[4])
                RES.append(lines[5])
                SHR.append(lines[6])
                CPU.append(lines[8])
                MEM.append(lines[9])

        df['PR'] = PR
        df['NI'] = NI
        df['VIRT'] = VIRT
        df['RES'] = RES
        df['SHR'] = SHR
        df['%CPU'] = CPU
        df['%MEM'] = MEM

        df.to_csv('{}/Final/{}.csv'.format(path, name))
    else:
        CPU = []
        NI = []
        RES = []
        VIRT = []
        SHR = []
        CPU = []
        MEM = []


cases = ['mongo', 'open5gs', 'srsenb']

for i in range(25, 76, 25):
    for nom in cases:
        transform_csv('Results/Local/Resources', '{}_clean_{}_resources'.format(i, nom),
'Local')
```

plot_resources.py:

```
import numpy as np
import pan das as pd
import matplotlib.pyplot as plt


def plot_CPU_Resources(deploy, j, s):
    cont = 0
    CPU_mongo = data['CPU_mongo_{}PRB'.format(j)]
    CPU_srsenb = data['CPU_srsenb_{}PRB'.format(j)]
    CPU_amfd = data['CPU_amfd_{}PRB'.format(j)]
    CPU_ausfd = data['CPU_ausfd_{}PRB'.format(j)]
    CPU_bsfd = data['CPU_bsfd_{}PRB'.format(j)]
    CPU_mmed = data['CPU_mmed_{}PRB'.format(j)]
    CPU_nrfd = data['CPU_nrfd_{}PRB'.format(j)]
    CPU_nssfd = data['CPU_nssfd_{}PRB'.format(j)]
    CPU_sgwcd = data['CPU_sgwcd_{}PRB'.format(j)]
    CPU_sgwud = data['CPU_sgwud_{}PRB'.format(j)]
    CPU_smfd = data['CPU_smfd_{}PRB'.format(j)]
    CPU_udmd = data['CPU_udmd_{}PRB'.format(j)]
    CPU_upfd = data['CPU_upfd_{}PRB'.format(j)]
    CPU_udrd = data['CPU_udrd_{}PRB'.format(j)]
    CPU_hssd = data['CPU_hssd_{}PRB'.format(j)]
    CPU_pcfd = data['CPU_pcfd_{}PRB'.format(j)]
    CPU_pcrfd = data['CPU_pcrfd_{}PRB'.format(j)]

    # labels = ['25', '50', '75']
    labels = ['{}'.format(j)]
    width = 0.3  # the width of the bars

    bar1 = np.arange(len(labels))
    bar2 = []
    list_other = list(np.add(CPU_mongo, CPU_amfd))

    for k in bar1:
        bar2.append(k + width)

    plt.bar(bar1, CPU_mongo, width, label='Mongo', color=colors[cont])
    cont += 1
    plt.bar(bar1, CPU_amfd, width, bottom=CPU_mongo, label='amfd',
color=colors[cont])
    cont += 1
    plt.bar(bar1, CPU_ausfd, width, bottom=list_other, label='ausfd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_ausfd)
    cont += 1
    plt.bar(bar1, CPU_bsfd, width, bottom=list_other, label='bsfd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_bsfd)
    cont += 1
    plt.bar(bar1, CPU_mmed, width, bottom=list_other, label='mmed', color=colors[cont])
    list_other = add_fragment(list_other, CPU_mmed)
    cont += 1
    plt.bar(bar1, CPU_nrfd, width, bottom=list_other, label='nrfd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_nrfd)
    cont += 1
```

```
    plt.bar(bar1, CPU_nssfd, width, bottom=list_other, label='nssfd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_nssfd)
    cont += 1
    plt.bar(bar1, CPU_sgwcd, width, bottom=list_other, label='sgwcd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_sgwcd)
    cont += 1
    plt.bar(bar1, CPU_sgwud, width, bottom=list_other, label='sgwud',
color=colors[cont])
    list_other = add_fragment(list_other, CPU_sgwud)
    cont += 1
    plt.bar(bar1, CPU_smfd, width, bottom=list_other, label='smfd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_smfd)
    cont += 1
    plt.bar(bar1, CPU_udmd, width, bottom=list_other, label='udmd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_udmd)
    cont += 1
    plt.bar(bar1, CPU_upfd, width, bottom=list_other, label='upfd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_upfd)
    cont += 1
    plt.bar(bar1, CPU_udrd, width, bottom=list_other, label='udrd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_udrd)
    cont += 1
    plt.bar(bar1, CPU_hssd, width, bottom=list_other, label='hssd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_hssd)
    cont += 1
    plt.bar(bar1, CPU_pcfd, width, bottom=list_other, label='pcfd', color=colors[cont])
    list_other = add_fragment(list_other, CPU_pcfd)
    cont += 1
    plt.bar(bar1, CPU_pcrfd, width, bottom=list_other, label='pcrfd', color=colors[cont])
    cont += 1

    plt.bar(bar2, CPU_srsenb, width, label='srsenb', color=colors[cont])

    plt.grid(color='tab:gray', linestyle='--', linewidth=0.5)
    plt.ylabel('% CPU', fontsize=22)
    plt.xlabel('Nº of PRB', fontsize=22)
    plt.title('{} deployment of {}Mbps'.format(deploy, s), fontsize=20)
    plt.xticks(bar1 + width / 2, labels, fontsize=20)
    plt.yticks(fontsize=20)
    plt.legend(loc=2, prop={'size': 15})


def add_fragment(old_list, fragment):
    old_list = list(np.add(old_list, fragment))
    return old_list


def plot_MEM_Resources(deploy, j, s):
    cont = 0
    MEM_mongo = data['MEM_mongo_{}PRB'.format(j)]
    MEM_srsenb = data['MEM_srsenb_{}PRB'.format(j)]
    MEM_amfd = data['MEM_amfd_{}PRB'.format(j)]
    MEM_ausfd = data['MEM_ausfd_{}PRB'.format(j)]
    MEM_bsfd = data['MEM_bsfd_{}PRB'.format(j)]
    MEM_mmed = data['MEM_mmed_{}PRB'.format(j)]
```

```
    MEM_nrfd  = data['MEM_nrfd_{}PRB'.format(j)]
    MEM_nssfd = data['MEM_nssfd_{}PRB'.format(j)]
    MEM_sgwcd = data['MEM_sgwcd_{}PRB'.format(j)]
    MEM_sgwud = data['MEM_sgwud_{}PRB'.format(j)]
    MEM_smfd = data['MEM_smfd_{}PRB'.format(j)]
    MEM_udmd = data['MEM_udmd_{}PRB'.format(j)]
    MEM_upfd = data['MEM_upfd_{}PRB'.format(j)]
    MEM_udrd = data['MEM_udrd_{}PRB'.format(j)]
    MEM_hssd = data['MEM_hssd_{}PRB'.format(j)]
    MEM_pcfd = data['MEM_pcfd_{}PRB'.format(j)]
    MEM_pcrfd = data['MEM_pcrfd_{}PRB'.format(j)]

    labels = ['{}'.format(j)]
    width = 0.3  # the width of the bars

    bar1 = np.arange(len(labels))
    bar2 = []
    list_other = list(np.add(MEM_mongo, MEM_amfd))

    for k in bar1:
        bar2.append(k + width)

    plt.bar(bar1, MEM_mongo, width, label='Mongo', color=colors[cont])
    cont += 1
    plt.bar(bar1, MEM_amfd, width, bottom=MEM_mongo, label='amfd',
color=colors[cont])
    cont += 1
    plt.bar(bar1, MEM_ausfd, width, bottom=list_other, label='ausfd', color=colors[cont])
    list_other = add_fragment(list_other, MEM_ausfd)
    cont += 1
    plt.bar(bar1, MEM_bsfd, width, bottom=list_other, label='bsfd', color=colors[cont])
    list_other = add_fragment(list_other, MEM_bsfd)
    cont += 1
    plt.bar(bar1, MEM_mmed, width, bottom=list_other, label='mmed',
color=colors[cont])
    list_other = add_fragment(list_other, MEM_mmed)
    cont += 1
    plt.bar(bar1, MEM_nrfd, width, bottom=list_other, label='nrfd', color=colors[cont])
    list_other = add_fragment(list_other, MEM_nrfd)
    cont += 1
    plt.bar(bar1, MEM_nssfd, width, bottom=list_other, label='nssfd', color=colors[cont])
    list_other = add_fragment(list_other, MEM_nssfd)
    cont += 1
    plt.bar(bar1, MEM_sgwcd, width, bottom=list_other, label='sgwcd',
color=colors[cont])
    list_other = add_fragment(list_other, MEM_sgwcd)
    cont += 1
    plt.bar(bar1, MEM_sgwud, width, bottom=list_other, label='sgwud',
color=colors[cont])
    list_other = add_fragment(list_other, MEM_sgwud)
    cont += 1
    plt.bar(bar1, MEM_smfd, width, bottom=list_other, label='smfd', color=colors[cont])
    list_other = add_fragment(list_other, MEM_smfd)
    cont += 1
    plt.bar(bar1, MEM_udmd, width, bottom=list_other, label='udmd', color=colors[cont])
```

```
    list_other = add_fragment(list_other, MEM_udmd)
    cont += 1
    plt.bar(bar1, MEM_upfd, width, bottom=list_other, label='upfd', color=colors[cont])
    list_other = add_fragment(list_other, MEM_upfd)
    cont += 1
    plt.bar(bar1, MEM_udrd, width, bottom=list_other, label='udrd', color=colors[cont])
    list_other = add_fragment(list_other, MEM_udrd)
    cont += 1
    plt.bar(bar1, MEM_hssd, width, bottom=list_other, label='hssd', color=colors[cont])
    list_other = add_fragment(list_other, MEM_hssd)
    cont += 1
    plt.bar(bar1, MEM_pcfd, width, bottom=list_other, label='pcfd', color=colors[cont])
    list_other = add_fragment(list_other, MEM_pcfd)
    cont += 1
    plt.bar(bar1, MEM_pcrfd, width, bottom=list_other, label='pcrfd', color=colors[cont])
    cont += 1

    plt.bar(bar2, MEM_srsenb, width, label='srsenb', color=colors[cont])

    plt.grid(color='tab:gray', linestyle='--', linewidth=0.5)
    plt.ylabel('% MEM', fontsize=22)
    plt.xlabel('Nº of PRB', fontsize=22)
    plt.xticks(bar1 + width / 2, labels, fontsize=20)
    plt.yticks(range(0, 11), fontsize=20)
    plt.title('{} deployment of {}Mbps'.format(deploy, s))
    plt.legend(loc=2, prop={'size': 15})


deploys = ["Local", "Docker”, "Kubernetes"]
num_PRB = [25, 50, 75]
range_plot = [1, 25, 50, 75, 100, 150]
colors = ['#0000CD', '#FF6103', '#76EE00', '#DC143C', '#9932CC', '#8B4500',
'#FFC125', '#00EEEE',
          '#8B8878', '#FF6103', '#C0FF3E', '#E0EEEE', '#E3CF57', '#7CFC00',
'#9AC0CD', '#030303', '#FF1493']

for i in deploys:
    for s in range_plot:
        data = pd.read_csv("Results/Clean data/{}/clean_resources_{}Mbps.csv".format(i,
s))

        for j in num_PRB:
            plt.figure(figsize=(15, 10))

            plot_CPU_Resources(i, j, s)
            #plot_MEM_Resources(i, j, s)
            plt.show()
```

Forced disconnection

Scripts to extract, clean and plot the forced disconnection data.

extract_disc.py:

```
def format_line(text):
    text = text.replace("[  5]   ", "")
    text = text.replace("[  5]  ", "")
    text = text.replace("[  6]  ", "")
    text = text.replace("  sec ", "")
    text = text.replace(" MBytes ", "")
    text = text.replace(" KBytes ", "")
    text = text.replace(" Bytes ", "")
    text = text.replace(" Mbits/sec ", "")
    text = text.replace(" Kbits/sec ", "")
    text = text.replace(" bits/sec ", "")
    text = text.replace(" ms ", "")
    text = text.replace("  receiver", "")
    text = text.replace("  ", " ")
    text = text.replace("[ ID] Interval      Transfer   Bitrate     Jitter  Lost/Total Datagrams",
"")
    text = text.replace("[SUM] 0.0-15.2 sec 1 datagrams received out-of-order", "")
    text = text.replace("[SUM] 0.0-15.3 sec 1 datagrams received out-of-order", "")
    text = text.replace("WARNING: Size of data read does not correspond to offered
length", "")
    text = text.replace("- - - - - - - - - - - - - - - - - - - - - - - - -", "")
    text = text.replace("-----------------------------------------------------------", "")
    text = text.replace("Server listening on 5000", "")
    #    text = text.replace(".", ",")
    return text


def create_file(deployment, file):
    myLines = []

    with open('Data/{}/forced_dis/{}.txt'.format(deployment, file), 'r') as myfile:
        for myline in myfile:
            myLines.append(myline.rstrip('\n'))

    with open('Results/{}/Forced_dis/forced_dis.txt'.format(deployment), 'w+') as data:
        for element in myLines:
            not_found = 0
            for word in bad_words:
                if element.__contains__(word):
                    break
                else:
                    not_found += 1

                if not_found == len(bad_words):
                    formated_line = format_line(element)
                    data.write(formated_line + "\n")
```

```
bad_words = ["Accepted", "omitted", "local"]
deploy = ["Local", "Docker", "Kubernetes"]

for i in deploy:
    create_file(i, 'f_1mPRB50_25M')
```

clean_disc.py:

```
import pandas as pd


def convert(lst):
    return lst[0].split()


def transform_csv(file, i):
    cont = 0
    num_file = 1
    df = pd.DataFrame()
    interval = []
    transfer = []
    bit_rate_real = []
    jitter = []

    with open('{}'.format(file), 'r') as in_file:
        for myline in in_file:
            if myline == "\n":
                continue
            else:
                if cont == 121:
                    df['Interval(sec)'] = interval[cont-121:cont-1]
                    df['Transfer(KBytes)'] = transfer[cont-121:cont-1]
                    df['Bitrate_real(Mbits/sec)'] = bit_rate_real[cont-121:cont-1]
                    df['Jitter(ms)'] = jitter[cont-121:cont-1]
                    df.to_csv('Results/Clean data/{}/clean_forced_disc{}.csv'.format(i,
num_file))

                    interval = []
                    transfer = []
                    bit_rate_real = []
                    jitter = []
                    cont = 0
                    num_file += 1

                else:
                    lines = convert([myline])
                    interval.append(lines[0])
                    transfer.append(lines[1])
                    bit_rate_real.append(lines[2])
                    jitter.append(lines[3])
                    cont += 1


bad_words = ["Accepted", "omitted", "local"]
```

```
deploy = ["Local", "Docker", "Kubernetes"]


for i in deploy:
    transform_csv('Results/{}/Forced_dis/forced_dis.txt'.format(i), i)
```

plot_disc.py:

```
import pandas as pd
import matplotlib.pyplot as plt


def plot_disc(data, dep, cont):
    data_plot = data['Bitrate_real(Mbits/sec)']

    plt.plot(range_plot, data_plot, colors[cont], linewidth=4, label='Attempt {}'.format(i))
    plt.title('{} deployment'.format(dep))
    plt.grid(color='tab:gray', linestyle='--', linewidth=0.5)
    plt.xlabel('Time (sec)', fontsize=22)
    plt.xticks(range_plot, labels, rotation=90, fontsize=12, horizontalalignment='center')
    plt.yticks(fontsize=15)
    plt.ylabel('Throughput (Mbps)', fontsize=22)
    plt.legend(loc=3, prop={'size': 15})


deploys = ["Local", "Docker", "Kubernetes"]
colors = ['#0000FF', '#66CD00', '#FF00FF', '#C76114', '#00BFFF', 'k']
range_plot = range(0, 120, 1)
labels = []
sum = 0

for i in range(0, 120):
    labels.append('{:.2f} '.format(sum+0.25))
    #labels.append('')
    sum += 0.25


for dep in deploys:
    cont = 0
    plt.figure(figsize=(15, 10))

    for i in range(1, 6):
        data = pd.read_csv("Results/Clean data/{}/clean_forced_disc{}.csv".format(dep, i))
        plot_disc(data, dep, cont)
        cont += 1
    plt.show()
```

REFERENCES

[1] 3GPP, "Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Link Control (RLC) protocol specification (Release 8), Rep. TS 36.322," Sophia Antipolis, 2010.

[2] 3GPP, "Evolved Universal Terrestrial Radio Access (E-UTRA); Base Station (BS) conformance testing (Release 10), Rep. TS 36.141," Sophia Antipolis, 2011.

[3] C. Cox, "An introduction to LTE. LTE, LTE-Advanced, SAE, VoLTE and 4G mobile communications," *John Wiley & Sons,* pp. 53-66, 2014.

[4] H. Yeh and H. Mutahir Abdul, "Hadamard SCFDMA-A modified uplink transmission scheme with low PAPR and SER," Proceedings of Annual IEEE Systems Conference (SysCon), 2015, pp. 711-715, doi: 10.1109/SYSCON.2015.7116834.

[5] H. Yin and S. Alamouti, "OFDMA: A Broadband Wireless Access Technology," 2006 IEEE Sarnoff Symposium, 2006, pp. 1-4, doi: 10.1109/SARNOF.2006.4534773.

[6] E. Hajlaoui, A. Zaier, A. Khlifi, J. Ghodhbane, M. B. Hamed and L. Sbita, "4G and 5G technologies: A Comparative Study," Proceedings of International Conference on Advanced Technologies for Signal and Image Processing (ATSIP), 2020, pp. 1-6, doi: 10.1109/ATSIP49331.2020.9231605.

[7] Source: “LTE Network Architecture”, [Online]. Available: https://www.tutorialspoint.com/lte/lte_network_architecture.htm. Accessed: 01/09/2021

[8] 3GPP, "Evolved Universal Terrestrial Radio Access (E-UTRA); Requirements on User Equipment’s (UE’s) supporting a release-independent frequency band (Release 14), Rep. TS 36.307," Sophia Antipolis, 2017.

[9] 3GPP, "Physical layer; General description (Release 15), Rep. TS 38.201," Sophia Antipolis, 2018.

[10] Source: "ITU-R IMT 2020 requirements," ETRI graphic.

[11] 3GPP, "Technical Specification Group Services and System Aspects; Study on Architecture for Next Generation System (Release 14), Rep. TR 23.799," Sophia Antipolis, 2016.

[12] Source: “5G Core Network Functions” [Online]. Available: https://www.grandmetric.com/2018/03/02/5g-core-network-functions/. Accessed: 01/09/2021

[13] Source: “5G NR Deployment Scenarios” [Online]. Available: https://www.rfwireless-world.com/Terminology/5G-NR-deployment-scenarios-or-modes.html. Accessed: 01/09/2021

[14] Antony Franklin, A., Tambe, S.D. Multi-access edge computing in cellular networks. *CSIT* 8, 85–92 (2020). https://doi.org/10.1007/s40012-020-00276-6

[15] A. Filali, A. Abouaomar, S. Cherkaoui, A. Kobbane and M. Guizani, "Multi-Access Edge Computing: A Survey," in IEEE Access, vol. 8, pp. 197017-197046, 2020, doi: 10.1109/ACCESS.2020.3034136.

[16] VENTRE, Pier Luigi et al. SDN Architecture and Southbound APIs for IPv6 Segment Routing Enabled Wide Area Networks. . 2018. DOI: 10.1109/TNSM.2018.2876251

[17] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," in Proceedings of the IEEE, vol. 103, no. 1, pp. 14-76, Jan. 2015, doi: 10.1109/JPROC.2014.2371999.

[18] "5G-EmPOWER," [Online]. Available: https://github.com/5g-empower/5g-empower.github.io/wiki. Accessed: 01/09/2021

[19] "Odin controller," [Online]. Available: https://github.com/Wi5/odin-wi5-controller. Accessed: 01/09/2021

[20] "FlexRAN", [Online]. Available: https://mosaic5g.io/flexran/. Accessed: 01/09/2021

[21] O-RAN Alliance, "O-RAN Use Cases and Deployment Scenarios, White Paper," 2020.

[22] Y. Bo, W. Xingwei, L. Keqin , D. k. Sajal and H. Min, "A comprehensive survey of Network Function Virtualization," *Computer Networks,* vol. 133, pp. 212-262, 2018.

[23] ETSI, "Architectural Framework, GS NFV 002 V1.2.1," Sophia Antipolis, 2014.

[24] Kratzke, Nane & Quint, Peter-Christian. (2017). Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study. Journal of Systems and Software. 126. 1-16. 10.1016/j.jss.2017.01.001.

[25] “LXD”, [Online]. Available: https://wiki.archlinux.org/title/LXD. Accessed: 01/09/2021

[26] “Windows Containers”, [Online]. Available: https://docs.microsoft.com/en-us/virtualization/windowscontainers/. Accessed: 01/09/2021

[27] “Docker”, [Online]. Available: https://www.docker.com/. Accessed: 01/09/2021

[28] “Kubernetes”, [Online]. Available: https://kubernetes.io/. Accessed: 01/09/2021

[29] "CNI Specification", [Online]. Available: https://github.com/containernetworking/cni/blob/master/SPEC.md. Accessed: 01/09/2021

[30] "Apache Mesos", [Online]. Available: https://github.com/apache/mesos/blob/master/docs/cni.md. Accessed: 01/09/2021

[31] "Cloud Foundry", [Online]. Available: https://github.com/cloudfoundry-attic/guardian-cni-adapter. Accessed: 01/09/2021

[32] Source: "CNI" [Online]. Available: https://thenewstack.io/container-networking-landscape-cni-coreos-cnm-docker/. Accessed: 01/09/2021

[33] "Contiv Networking", [Online]. Available: https://github.com/contiv/netplugin?utm_source=thenewstack&utm_medium=website&utm_campaign=platform. Accessed: 01/09/2021

[34] "Project Calico", [Online]. Available: https://docs.projectcalico.org/getting-started/kubernetes/. Accessed: 01/09/2021

[35] "Weave", [Online]. Available: https://github.com/weaveworks/weave?utm_source=thenewstack&utm_medium=website&utm_campaign=platform. Accessed: 01/09/2021

[36] "OIA 5G RAN", [Online]. Available: https://openairinterface.org/oai-5g-ran-project/. Accessed: 01/09/2021

[37] Code source: "OIA 5G RAN", [Online]. Available: https://gitlab.eurecom.fr/oai/openairinterface5g/. Accessed: 01/09/2021

[38] Source: "OIA 5G RAN Documentation", [Online]. Available: https://openairinterface.org/oai-5g-ran-project/. Accessed: 01/09/2021

[39] "srsLTE", [Online]. Available: https://www.srslte.com/. Accessed: 01/09/2021

[40] Code source: "srsLTE", [Online]. Available: https://github.com/srsran/srsran. Accessed: 01/09/2021

[41] "free5GRAN", [Online]. Available: https://free5g.github.io/free5GRAN-documentation/index.html. Accessed: 01/09/2021

[42] Code source: "free5GRAN", [Online]. Available: https://github.com/free5G/free5GRAN. Accessed: 01/09/2021

[43] "5G-EMPOWER", [Online]. Available: https://github.com/5g-empower/5g-empower.github.io/wiki. Accessed: 01/09/2021

[44] Source: "5G-EMPOWER", [Online]. Available: https://github.com/5g-empower/5g-empower.github.io/wiki. Accessed: 01/09/2021

[45] "OIA 5G CN", [Online]. Available: https://openairinterface.org/oai-5g-core-network-project/. Accessed: 01/09/2021

[46] OIA 5G CN code source: https://gitlab.eurecom.fr/oai/cn5g. Accessed: 01/09/2021

[47] Source: “OIA-CN Documentation”, [Online]. Available: https://openairinterface.org/oai-5g-core-network-project/. Accessed: 01/09/2021

[48] “Open5GS”, [Online]. Available: https://open5gs.org/. Accessed: 01/09/2021

[49] Source: “Open5GS Documentation”, [Online]. Available: https://open5gs.org/open5gs/docs/guide/01-quickstart/ Accesed 01/09/2021

[50] “free5GC”, [Online]. Available: https://www.free5gc.org/. Accessed: 01/09/2021

[51] Source code “free5GC”, [Online]. Available: https://github.com/free5gc/free5gc. Accessed: 01/09/2021

[52] D. Luong, H. Thieu, A. Outtagarts and Y. Ghamri-Doudane, "Cloudification and Autoscaling Orchestration for Container-Based Mobile Networks toward 5G: Experimentation, Challenges and Perspectives," Proceedings of the IEEE 87th Vehicular Technology Conference (VTC Spring), 2018, pp. 1-7, doi: 10.1109/VTCSpring.2018.8417602.

[53] S. Imadali and A. Bousselmi, "Cloud Native 5G Virtual Network Functions: Design Principles and Use Cases," 2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2), 2018, pp. 91-96, doi: 10.1109/SC2.2018.00019.

[54] W. Lai, Y. Wang and K. Chiu, "Containerized Design and Realization of Network Functions Virtualization for a Light-Weight Evolved Packet Core Using OpenAirInterface," Proceedings of the Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), 2018, pp. 472-477, doi: 10.23919/APSIPA.2018.8659522.

[55] A. Esmaeily, K. Kralevska and D. Gligoroski, "A Cloud-based SDN/NFV Testbed for End-to-End Network Slicing in 4G/5G," Proceedings of the IEEE Conference on Network Softwarization (NetSoft), 2020, pp. 29-35, doi: 10.1109/NetSoft48620.2020.9165419.

[56] C. V. Nahum et al., "Testbed for 5G Connected Artificial Intelligence on Virtualized Networks," in IEEE Access, vol. 8, pp. 223202-223213, 2020, doi: 10.1109/ACCESS.2020.3043876.

[57] Ahmed, T., Dubois, E., Dupé, J.-B., Ferrús, R., Gélard, P., and Kuhn, N. (2018) Software-defined satellite cloud RAN. *Int. J. Satell. Commun. Network.*, 36: 108– 133. doi: 10.1002/sat.1206.

[58] X. Wang et al., "Handover reduction in virtualized cloud radio access networks using TWDM-PON fronthaul," in IEEE/OSA Journal of Optical Communications and Networking, vol. 8, no. 12, pp. B124-B134, December 2016, doi: 10.1364/JOCN.8.00B124.

[59] YAN, Zheng, Peng ZHANG a Athanasios V VASILAKOS. A security and trust framework for virtualized networks and software-defined networking.

Security and communication networks. Hindawi Limited, 2016, roč. 9, č. 16, s. 3059–3069. ISSN 1939-0114. DOI: 10.1002/sec.1243

[60] "Open-VERSO", [Online]. Available: https://www.openverso.org/en/open-verso/. Accessed: 01/09/2021

[61] "Docker project", [Online]. Available: https://github.com/herlesupreeth/docker_open5gs. Accessed: 01/09/2021

[62] "USRP B210", [Online]. Available: https://files.ettus.com/manual/page_usrp_b200.html

[63] "SIM Card", [Online]. Available: http://shop.sysmocom.de/products/sysmoISIM-SJA2. Accessed: 01/09/2021

[64] "HUAWEI LTE USB Stick", [Online]. Available: https://consumer.huawei.com/en/routers/e3372/specs/. Accessed: 01/09/2021

[65] "Pysim software package", [Online]. Available: http://git.osmocom.org/pysim/. Accessed: 01/09/2021

[66] "Pysim project", [Online]. Available: https://osmocom.org/projects/pysim/wiki. Accessed: 01/09/2021

[67] "srsLTE installation source code", [Online]. Available: https://docs.srslte.com/en/latest/general/source/1_installation.html#installation-from-source. Accesed 01/09/2021

[68] "D-Link DGS 108 Switch", [Online]. Available: https://images-eu.ssl-images-amazon.com/images/I/A1j3YykkauS.pdf. Accessed: 01/09/2021

[69] "Docker Hub", [Online]. Available: https://hub.docker.com/

[70] "Base srsLTE image", [Online]. Available: https://github.com/5g-empower/docker/tree/master/srsenb. Accessed: 01/09/2021

[71] "srsLTE Docker image", [Online]. Available: https://hub.docker.com/r/javipalomares/srslte. Accessed: 01/09/2021

[72] "Base srsLTE image", [Online]. Available: https://github.com/5g-empower/docker/tree/master/open5gs. Accessed: 01/09/2021

[73] "Open5GS Docker image", [Online]. Available: https://hub.docker.com/r/javipalomares/open5gs. Accessed: 01/09/2021

[74] "IPERF", [Online]. Available: https://iperf.fr/. Accessed: 01/09/2021

[75] Marotta, Antonio et al. On the energy cost of robustness for green virtual network function placement in 5G virtualized infrastructures. 2017.

[76] Suraci, Chiara et al. A stakeholder-oriented security analysis in virtualized 5G cellular networks. Computer networks (Amsterdam, Netherlands: 1999). Elsevier B.V, 2021, roč. 184, s. 107604–. ISSN 1389-1286. DOI: 10.1016/j.comnet.2020.107604

[77] Yan, Zheng, Peng ZHANG a Athanasios V VASILAKOS. A security and trust framework for virtualized networks and software-defined networking: Security and trust framework for virtualized networks and SDN. Security and communication networks. 2016, roč. 9, č. 16, s. 3059–3069. ISSN 1939-0114. DOI: 10.1002/sec.1243