Design and Evaluation of a K8s-based System for Distributed Open-Source Cellular Networks

Javier Palomares*‡, Estefanía Coronado*†, David Rincón‡, and Muhammad Shuaib Siddiqui*
*i2CAT Foundation, Barcelona, Spain; Email: {javier.palomares, estefania.coronado, shuaib.siddiqui}@i2cat.net
‡Department of Network Engineering, Universitat Politècnica de Catalunya (UPC), Castelldefels, Barcelona, Spain;
Email: javier.palomares@estudiantat.upc.edu, david.rincon@upc.edu
†Universidad de Castilla-La Mancha, Albacete, Spain; Email: estefania.coronado@uclm.es

***Abstract*—Virtualization in 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. This paper aims to study the viability and the performance of cloud-native infrastructures for hosting network functions. The selected frameworks implement both the 4G and the 5G stacks and their network functions. This work considers a variety of scenarios for enabling the deployment of a distributed and open-source cellular network: a baremetal setup, an all-docker-based setup and the proposed Kubernetes setup. Moreover, an analysis of the impact that the Radio Access Network (RAN) and the Core Network (CN) have on computational resource utilization is presented as the network conditions vary. The design proposed in this work has been validated and analyzed using the proposed prototype and testbed. This paper proposes a design to increase resource usage flexibility and performance and reduction of deployment time. The analysis of the gathered data reveals that the deployments of containerized cellular networks display better performance in terms of flexibility, low startup times, and ease of deployment while consuming the same resources as the non-containerized.**

***Index Terms*—Virtualization, NFV, SDN, Distributed Systems, 4G, 5G, Performance Evaluation.**

I. Introduction

In recent years, mobile networks have pursued to fulfill users' demands, including ultra-fast deployment, high speed, and reliability. Due to these needs, consecutive generations of enhanced communication networks have been deployed globally. Especially in a new fully distributed network paradigm where network functions are deployed in different locations and even provided by several software frameworks. Combining these requirements with the appearance of technologies, such as Software Defined Networks (SDN) [1] and Network Functions Virtualisation (NFV) [2], and recent virtualized infrastructure managers, encourage a focus on developing a more flexible architecture for mobile networks and examining their impact on the underlying infrastructure.

The cloud-native architecture [3] is a design methodology that utilizes cloud services, such as Elastic Compute Cloud (EC2), Simple Storage Service (S3), and Lambda from Amazon Web Services [4]. These services allow dynamic and agile application development techniques which take a modular approach to building, running, and updating software through a suite of cloud-based microservices. Each microservice is created to execute a particular function (implement, communicate, or run processes), which are often packaged into containers. Some benefits of containerization are: (i) portability, ease, and speed of deployment; (ii) fast scalability upon demand; and (iii) avoidance of incompatibility issues between the libraries and dependencies of the different components when updating or setting up the environment.

In the literature there exist works evaluating the impact that technologies such as SDN or NFV have in virtualized systems [5] [6], to solve the increase in demand on the vital requirements for bandwidth, latency, and quality of service. In [7] the authors provide a review focusing on the benefits of containerizing and distributing the Core Network (CN), while the work in [8] presents the impact of containerization and distribution of the Radio Access Network (RAN). By contrast, the main objective of this work is to enable the deployment of a distributed and configurable open-source solution for cellular networks where the CN and the RAN are containerized separately and deployed in different locations. Moreover, we aim to empirically analyze the impact that both containers have on the underlying hardware as the network conditions change. In this context, the contributions of this paper are as follows:

- First, we analyze how different network conditions affect the proposed Kubernetes-based deployment's infrastructure resources. Our prototype's capabilities are compared to: (i) a baseline setup, in which the original software frameworks are deployed directly to the Operating System, without any containerization; and (ii) an all-docker-based project [9] found in the researched state-of-the-art, which provides comparable functionality.
- Second, we compare and analyze the effects of distributing the software modules over several devices versus concentrating all modules on a single node.
- Third, the state-of-the-art works provide descriptors and Docker images that are exclusively suitable to their User Equipment (UE) (not generic). For that reason, we propose a set of descriptors that enable a configurable, automated, and distributed deployment (separating the RAN and CN functions across a Kubernetes (K8s) cluster). This programmable feature allows the end user to

perform a transparent registration of the UE, regardless of the scenario. The deployment of the network segments is enabled by two Docker images, which provide the necessary software modules to deploy an open-source-based network in seconds. In particular, the software implementation takes as a reference the srsRAN [10] and Open5GS [11] frameworks, as two of the most used open-source solutions for cellular networks.

The remainder of this paper is organized as follows. Section II reviews the related work on tools for building distributed cellular networks. Section III focuses on the research work, explaining the proposed design and configuration for the Kubernetes-based network deployment. Section IV presents the methodology used in the evaluation and discusses the achieved results. Conclusions and future lines of work are summarized in Section V.

II. Related Work

One of the key enablers for distributed networks is undoubtedly virtualization and containerization, whose role in networking is defined by the authors of [7] 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". They propose Docker to enable the cloudification of mobile network functions. They also build a proof of concept of the scalability of SGW, comparing the performances of Kubernetes and Mesos-Marathon. That showed that container-based approaches are a viable option for achieving elasticity on future mobile networks.

Moreover, two different testbeds for cloud-based 5G networks are analyzed in [12] and [13] to shape 5G technology as a flexible, scalable, and demand-oriented network. The work in [12] introduces a novel testbed called 5GIIK, which provides implementation, management, and orchestration across all network domains and different access technologies. It provides capabilities such as slice provision dynamicity and real-time monitoring. On the other hand, the authors of [13] display a 5G mobile network testbed with a virtualized and orchestrated structure. It uses containers to deploy and orchestrate VNFs to flexibly create various mobile network scenarios.

Taking into consideration the virtualization of the radio access network, the work in [8] aims to ease the integration of satellite components in forthcoming 5G systems (SatCloudRAN). The authors give special attention to the design, by considering the split and placement of virtualized and non-virtualized functions, while considering 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 related works mentioned focus their efforts on analyzing separately: (i) the impact that technologies such as SDN or NFV have in virtualized systems; (ii) the creation of different sets of testbeds for cloud-based 5G networks; (iii) the benefits of containerizing, and distributing the core part of the network; and (iv) the behavior of the virtualization of the RAN. This paper uses existing open-source frameworks and orchestration tools to enable the deployment of a containerized, distributed and configurable solution for cellular networks. This deployment has been performed in two different manners: (i) a single worker node cluster containing both, the RAN and the CN; and (ii) the separation of the CN and RAN logic into two different worker nodes. Due to this, we propose our own Docker images and K8s descriptor files for the deployment. The prototype has been tested on the proposed testbed, and an in-depth analysis has been performed on the impact the RAN and the CN have on resource utilization (radio and virtualized infrastructure) as the network conditions vary. To the best of our knowledge, no prior research has been conducted in the literature discussing the impact on resource consumption and the network performance of containerizing, and distributing both the CN and RAN parts of the network.

III. Design of the Setup and Configuration Files Required for the Deployment

The main aim of this paper is to enable a configurable, distributed and containerized open-source-based network that could be deployed on multiple nodes, keeping the RAN and CN functions separated. Based on this deployment, an analysis of the network and the infrastructure performance is done when various network parameters change. Furthermore, the purpose of this work is to examine the impact of dispersing the CN and RAN modules among multiple devices, as opposed to having all the logic focused on a single node.

The road to fulfilling these goals begins with achieving the most basic internet connectivity in a UE, using a baremetal deployment. All the network functions of the CN and the RAN are directly instantiated on a single computer's operating system. This is done to ensure that the open-source modules behave as expected. The second stage consists of the design of a virtualized network using Kubernetes to automate and manage the deployment. Fig. 1 depicts the two K8s-based configurations that have been implemented. The initial K8s cluster (shown by blue lines) consists of two Ubuntu 20.04 PCs, one functioning as a master node and the other as a worker node. The master node is in charge of hosting the cluster's control plane and, among other things, it is responsible for providing scheduling for the pods. Workloads for applications are scheduled using the worker nodes. The pods holding the logic of the CN and the RAN are deployed together in the worker node in this case. In the second scenario (shown by red lines), the K8s cluster is composed of three computers running Ubuntu 20.04, one of which serves acting as the master node and the other two as worker nodes. The pods holding the RAN and CN functionalities are distributed among the two worker nodes in this scenario. In addition to the nodes, the following hardware components are employed: (i) a laptop running Ubuntu 20.04, that serves as the UE; (ii) a HUAWEI LTE USB Stick; (iii) a programmable sysmoUSIM

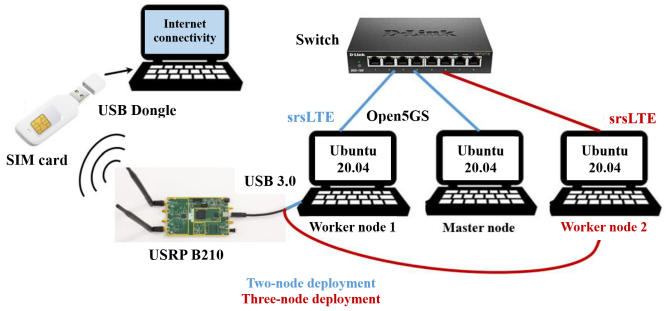


Fig. 1. K8s two-node and three-node deployments.

that will connect the computer to the network in order to gain Internet access; (iv) a USRP B210 antenna with USB 3.0 connected to one of the worker nodes; and (v) a D-Link DGS 108 Switch, to interconnect the nodes of the cluster.

Before setting up the K8s clusters, the following steps must be completed on all nodes: (i) configuring the computer hostnames; (ii) installing Docker; (iii) disabling swap and enabling IP forwarding; and (iv) installing kubectl, kubelet and kubeadm. Docker was chosen as the Container Runtime Engine (CRE) due to the majority of the relevant related-work projects researched were based on it and because it was supported when the project was established, even if the support is no longer maintained. Other CRE, such as containerd [14], should be examined for future works. Calico was chosen as a plug-in for the Kubernetes Container Network Interface (CNI), which offers Kubernetes agents to enable networking for containers and pods via the K8s API.

The srsLTE and Open5GS solutions were chosen to deploy the RAN and CN, respectively. Those open-source efforts were chosen for their: (i) active community; (ii) extensive documentation; and (iii) implementation of both the 4G and 5G stacks. Because the Docker Hub images of both software were too specific, designed exclusively for their own UE, it was required to create new Docker images to enable a generic and automated deployment. In the process, a new configurable feature was added. It enables future users to use their own UE by entering SIM card specifications such as: the Mobile Country Code (MCC), Mobile Network Code (MNC), Tracking Area Code (TAC), and the number of physical resource blocks (PRBs) that need to be allocated. This is made feasible by the inclusion of a script that adjusts the internal settings of the CN and RAN software while launching a descriptor file, as will be discussed further below. These Docker images are freely available to the community in [15].

Following the successful construction of the Docker images, it was essential to create a descriptor file in charge of deploying the various pods and services required for the distributed network to function properly. A manageable piece of this file, which defines the second proposed scenario (one master-node and two worker-nodes), is described below to help the better understanding of its structure and functioning. The descriptor in Code 1 defines the following pods:

```
kind: Pod
    name: epc                #Name of the Pod
spec:
   containers:
   - name: open5gs           #Container name
       image: javipalomares/open5gs:latest
       env:          #Environmental variables
       - name: mcc
           value: "001"
       - name: mnc
           value: "03"
       - name: tac
           value: "7"
   nodeSelector:
       IDname: kworker1      #Node label
---
kind: Pod
   name: srsenb              #Name of the Pod
spec:
   containers:
   - name: srsenb            #Container name
      image: javipalomares/srslte:latest
      env:           #Environmental variables
      - name: mcc
         value: "001"
      - name: mnc
         value: "03"
      - name: n_prb
         value: "75"
   nodeSelector:
      IDname: kworker2       #Node label
```

Code 1. Fragment of the descriptor file.

- The first pod is called epc and consists of a container based on the Open5GS Docker image. The modified version of that image, which includes all of the CN logic, may be found in (javipalomares/open5gs:latest). Following that, the environmental variables (mcc, mnc, and tac) that enable the configurable functionality must be initialized. The MCC and MNC are part of the SIM card's International Mobile Subscriber Identity (IMSI). They are utilized to register the UE. The TAC (Tracking Area Code) is a Tracking Area's unique identifier. Due to one of this work's objectives is to study the impact of separating the CN and RAN logic on various hosts, the whole core is deployed in a single container.
- The second pod is named srsenb and consists of a container based on the srsLTE Docker image. The modified version of that image, which includes all of the RAN logic, may be found in (javipalomares/srslte:latest). The environmental variables (mcc, mnc, and n_prb) that enable the configurable functionality must be initialized. As in the preceding pod, the MCC and MNC are utilized to register the UE, and the n_prb represents the number of the amount of physical resource blocks to be allocated.

IV. Performance Evaluation and Achieved Results

A. Methodology

This section compares the baremetal deployment's behavior, using its results as a baseline, to the all-Docker-based project

TABLE I
LIST OF THE EXPERIMENTS PERFORMED.

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

and the Kubernetes deployments. This study compares the impact of having the CN and RAN software modules divided into various devices to having all the logic concentrated on a single one. The analysis focuses mainly on resource performance, with a particular emphasis on the findings obtained in open-source-based Kubernetes scenarios.

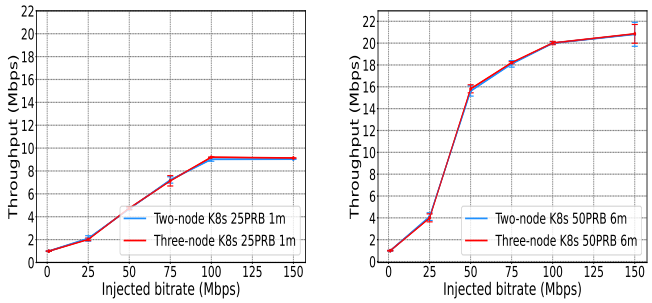
The tests were carried out with the iPerf3 software, which is a tool for creating data streams for both Transport Control Protocol (TCP) and User Datagram Protocol (UDP) to analyze network performance. Only UDP streams will be utilized for deployment evaluation in order to adjust the transmission rate parameter. The iPerf3 flows' endpoints are the host executing the RAN logic and the registered UE. For each of the following scenarios, a 95% confidence interval is provided to estimate the measurements:

- Different deployments: Baseline, Docker-based project and two-node and three-node Kubernetes clusters.
- Maximum number of Physical Resource Blocks (PRB) on the eNB/gNB: 25, 50 and 75.
- Distances between RAN-UE (meters):[1, 3, 6, 10, 15, 20].
- Transmission speed of the packet streams: 1Mbps, 25Mbps, 50Mbps, 75Mbps, 100Mbps and 150Mbps.

The parameter values are designed to accurately represent and study the testbed's behavior and resource usage, ranging from tiny to limiting quantities. Table I outlines the set of experiments done for each deployment and defines the parameters involved and their analyzed values in each case.

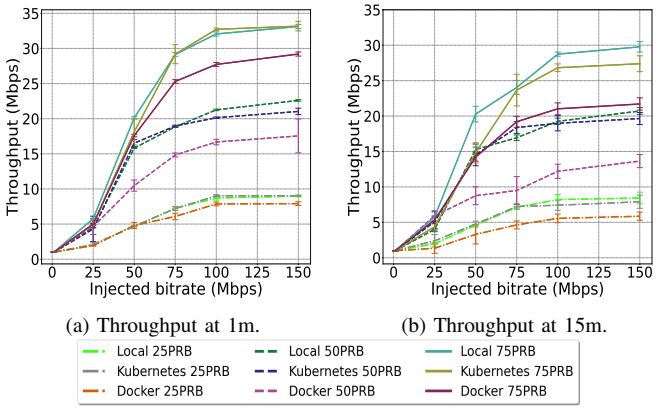
B. Results Discussion

Before delving into the results, it is important to note that, due to the similarities in the data collected from the two Kubernetes scenarios, only the results of the most distributed deployment will be shown. As previously stated, this work assesses the impact of separating the CN and RAN logic modules into various devices. The closeness of the results of having the CN and RAN modules separated in multiple Kubernetes worker-nodes vs having all the logic focused on one worker-node can be seen in Fig. 2. It compares the throughput of two-node (blue) and three-node (red) K8s clusters under two different sets of conditions. Fig. 2a exhibits the 25 PRBs and 1-meter distance between the UE and the antenna, whereas Fig. 2b depicts the 50 PRBs and 6-meter distance. Due to output similarities, for brevity, and because one of the goals of this article is to make deployment as dispersed as feasible,



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

Fig. 2. Results comparison between the two-node and the three-node K8s deployments for two different sets of conditions.

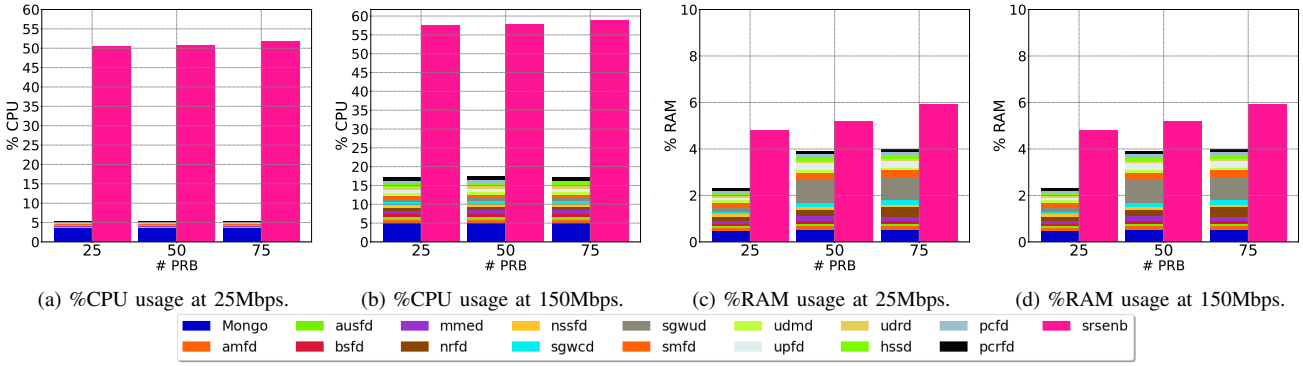


(a) Throughput at 1m. (b) Throughput at 15m.

Fig. 3. Comparison between deployments and PRBs at different distances (experiments 1 and 5).

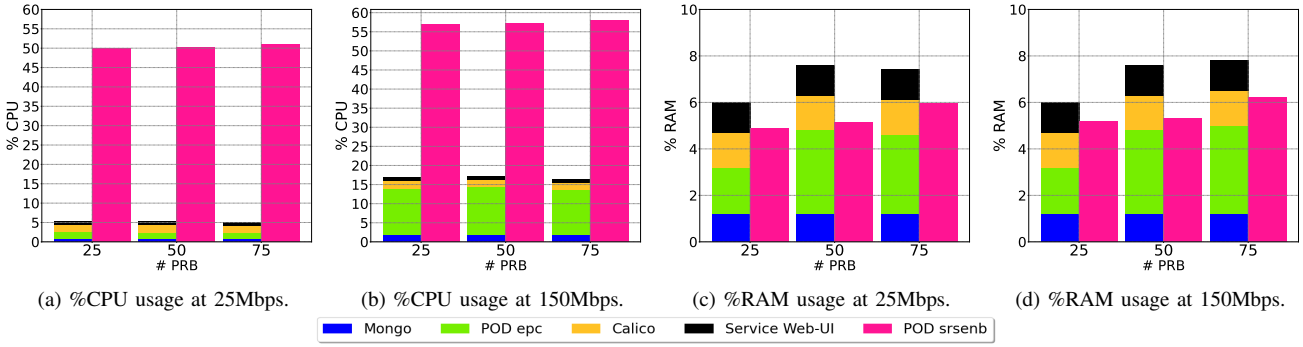
the following figures and results will only display findings from the three-node Kubernetes scenario.

1) Measuring throughput: In experiments 1–6, the average throughput was measured using the average values provided by iPerf3 at the end of each test. Fig. 3 displays the average throughput of trials 1 and 5, comparing the three distinct deployments while varying the amount of PRB. By analyzing the results, it is readily observed in Fig. 3 that distance and number of PRBs influence the throughput of the Kubernetes deployment. As can be deduced, the more resources allocated (the greater the quantity of PRB), the higher the throughput. However, it has been shown that the greater the distance between the UE and the RAN, the lower the throughput. Furthermore, as the distance grows, so does the confidence interval at 95%. This signifies that the value of the throughput fluctuates more with regard to the average. Interferences and free space losses are the cause of these behaviors. When comparing the K8s deployment to the baseline setup, the network distribution has almost no negative impact on the throughput values. When comparing the K8s deployment to the baseline configuration, there is almost no negative impact associated with the distribution of the network on the throughput. Their tendencies and values are very similar. In contrast, the attained



(a) %CPU usage at 25Mbps. (b) %CPU usage at 150Mbps. (c) %RAM usage at 25Mbps. (d) %RAM usage at 150Mbps.

Fig. 4. Comparison of the resource consumption at different injected bitrates and number of PRB, in the baseline deployment (experiments 7 and 11).



(a) %CPU usage at 25Mbps. (b) %CPU usage at 150Mbps. (c) %RAM usage at 25Mbps. (d) %RAM usage at 150Mbps.

Fig. 5. Comparison of the resource consumption at different injected bitrates and number of PRB, in the K8s deployment (experiments 7 and 11).

throughput in the all-Docker-based setup is significantly lower than the other two, making our deployment more efficient than the one found in the literature.

Fig. 3 depicts that the RAN software dynamically assigns the number of PRBs based on the injected bitrate until it reaches the maximum available, causing the throughput to be substantially lower than the injected bitrate. It can also be appreciated that at 75 PRBs and 1-meter distance, saturation is reached at around 33 Mbps.

2) Measuring resource consumption: The computing resources utilized by each module of Open5GS and srsLTE were assessed in experiments 7-11. The main goal of these tests is to examine the effect that injected bandwidth and the number of PRBs have on the resources used.

Fig. 4a, Fig. 5a, Fig. 4b and Fig. 5b depict the CPU resource utilization of the baseline and Kubernetes deployments respectively, at two different injected bitrates. The resource consumption of both scenarios is approximately the same. After evaluating the results, the following conclusions were reached in the K8s deployment:

- The ”srsenb” module, which represents the RAN software, consumes far more resources than the other modules combined, accounting for about 60% of CPU capacity at high injected bitrates. This behavior is replicated in the rest of the setups. The amount of PRB has a little influence on the ”srsenb” module’s CPU consumption, but has essentially no effect on the rest of the components. The baseline deployment exhibits the same behavior.
- The ”epc” pod, which contains all of the core functions, consumes the same amount of CPU resources as all of the baseline CN functions’ combined. This is expected because, under the same network conditions, both deployments launch the same core functions.

Fig. 4c, Fig. 5c, Fig. 4d and Fig. 5d depict the use of RAM resources in the baseline and Kubernetes setups. In contrast to the results obtained in the CPU resources experiments, RAM utilization is more connected to the number of PRBs than to the traffic transiting the network. Furthermore, even if the ”srsenb” module still has the biggest contribution, its magnitude is no longer dominant in terms of RAM utilization. Furthermore, the Kubernetes deployment is the most RAM-consuming scenario, by 4% on average. When compared to the flexibility and ease of deployment provided by the K8s arrangement, this additional 4% can be regarded as trivial. This difference is due to the fact that there is no memory allocation on demand and all the modules are always working.

3) Measuring forced reconnecting time: The goal of experiment 12 is to observe how the system responds to a breakdown in one of its modules and if it is capable of resolving the problem and continuing to function. This is especially important in dispersed contexts where a virtual network function may

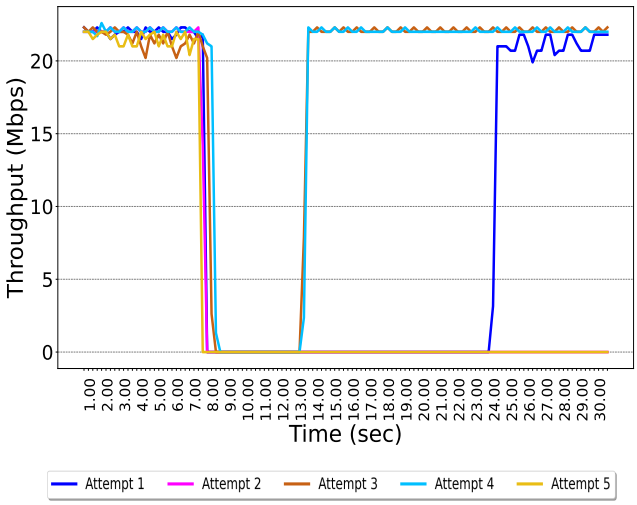


Fig. 6. K8s forced reconnection time (experiment 12).

be unavailable for a short period of time. In this regard, this experiment aims to determine how long it takes to reestablish the connection when the modules reconnect after a failure. To do so, some of the CN modules have been forced to fail randomly. Fig. 6 depicts the Kubernetes deployment's reconnection attempts. If any of the modules involved in the registration and storing of UE information fails, the system would lose connection. This behavior may be seen in tests 2 and 5. This is because the K8s descriptor file does not contain volumes to store the UE's sensitive data, and if one of these storing modules fails, the data is lost. When a deployment pod fails, Kubernetes will recreate it with the same functionality but in an empty state. A K8s volume is a directory or hard disk where sensible data may be stored and accessed by pods. This project's future work will consist of including these volumes in the deployment file to avoid data loss from re-occurring.

Fig. 6 shows two distinct sets of reconnection time: t and t+10 sec. This is related to the way the "srsenb" module operates. If the connection is lost, this module attempts to reconnect every 10 seconds and repeats the process if the previous attempts fail.

In terms of recovery time, the deployments are ordered as follows: baseline (7s), all-docker-based project (5.5s), and Kubernetes (5s). This outcome can be explained by the fact that the latter two are containerized installations. Instead of relaunching the whole deployment, each time a container fails, only it gets restarted immediately. This shortens the reconnecting time.

V. Conclusions

In this study, we have presented the architecture of a containerized, distributed, and customizable K8s open-source solution for cellular network deployments. It provides flexibility, low set up time and ease of deployment. Based on this prototype, we conducted an in-depth analysis that included a comparison with a baseline of the open-source RAN and CN frameworks employed, as well as a state-of-the-art project that was entirely Docker-based. From this analysis, we can conclude that the CPU consumption of the prototype is the resource that is most influenced when varying injected traffic and distance. RAM utilization, on the other hand, is more susceptible to changes related to the number of physical resource blocks allocated. Furthermore, the given K8s-based prototype has been shown to be more robust to failures and has a faster reconnection time than the baseline deployment. When compared to the benefits obtained, the difference in throughput and CPU use is negligible.

We planned the following future lines of work: (i) increasing the number of registered UEs; (ii) resolving issues with non-persistent UE registration data; (iii) extending the analysis performed to other criteria, such as energy efficiency; (iv) testing the deployment for 5G NSA and 5G SA equipment to ensure the same performance; and (v) separating each core function into a container or implementing them as microservices.

Acknowledgment

This work has been supported by the EU's H2020 projects 5GaaS (958832) and AI@EDGE (101015922). The authors would also like to acknowledge CERCA Programme / Generalitat de Catalunya for sponsoring this work. This work has been also supported by the EU "NextGenerationEU/PRTR", MCIN and AEI (Spain) under project IJC2020-043058-I.

References

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proc. of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.
[2] ETSI, "Network Functions Virtualisation (NFV); Architectural Framework," December 2014, ETSI GS NFV 002 V1.2.1.
[3] N. Kratzke and P.-C. Quint, "Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.
[4] AWS, "AWS," https://aws.amazon.com/, Accessed on 08.02.2023.
[5] S. Imadali and A. Bousselmi, "Cloud Native 5G Virtual Network Functions: Design Principles and Use Cases," in *Proc. of IEEE SC2*, Paris, France, 2018.
[6] W.-P. Lai, Y.-H. Wang, and K.-C. Chiu, "Containerized Design and Realization of Network Functions Virtualization for a Light-Weight Evolved Packet Core Using OpenAirInterface," in *Proc. of APSIPA ASC*, Honolulu, HI, USA, 2018.
[7] D.-H. Luong, H.-T. Thieu, A. Outtagarts, and Y. Ghamri-Doudane, "Cloudification and Autoscaling Orchestration for Container-Based Mobile Networks toward 5G: Experimentation, Challenges and Perspectives," in *Proc. of IEEE VTC Spring*, Porto, Portugal, 2018.
[8] T. AHMED, E. Dubois, J.-B. Dupé, R. Ferrus, P. Gélard, and N. Kuhn, "Software Defined Satellite Cloud RAN," *International Journal of Satellite Communications and Networking*, vol. 36, p. 108–133, 2018.
[9] herlesupreeth, "docker-open5gs," https://github.com/herlesupreeth/docker_open5gs, Accessed on 08.02.2023.
[10] srsRAN, "srsRAN," https://docs.srsran.com/, Accessed on 08.02.2023.
[11] open5Gs, "open5Gs," https://open5gs.org/, Accessed on 08.02.2023.
[12] A. Esmaeily, K. Kralevska, and D. Gligoroski, "A Cloud-based SD-N/NFV Testbed for End-to-End Network Slicing in 4G/5G," in *Proc. of IEEE NetSoft*, Ghent, Belgium, 2020.
[13] C. V. Nahum, L. De Nóvoa Martins Pinto, V. B. Tavares, P. Batista, S. Lins, N. Linder, and A. Klautau, "Testbed for 5G Connected Artificial Intelligence on Virtualized Networks," *IEEE Access*, vol. 8, pp. 223 202–223 213, 2020.
[14] containerd, "containerd," https://containerd.io/, Accessed on 08.02.2023.
[15] javipalomares, "docker-images," https://hub.docker.com/search?q=javipalomares&type=image, Accessed on 08.02.2023.