ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

MASTER'S DEGREE
IN
TELECOMMUNICATIONS ENGINEERING

O-RAN SOFTWARE DEPLOYMENT AND ORCHESTRATION ON VIRTUALIZED INFRASTRUCTURES

Master Thesis
in
Laboratory of Networking M

Supervisor
Prof. WALTER CERRONI

Candidate
PAOLO MALPEZZI

Co-supervisor
Dr. GIANLUCA DAVOLI

SESSION II
ACADEMIC YEAR 2020/2021

Contents

Abstract

Future Radio Access Networks will need to cope with an increasing complexity and new challenges due to the introduction of heterogeneous application scenarios, from enhanced Mobile Broadband to Network Slicing service diversification. In this context, the new trend of Open RAN is gaining importance, envisioning a transformation of traditional Radio Access Networks toward softwarization, virtualization and disaggregation of network functionalities, leveraging open and programmable protocols and interfaces. Embracing this new trend, this thesis aims to evaluate the software-based solutions promoted by the O-RAN Alliance, one of the major contributors toward an open and intelligent RAN architectural framework, aligned with 5G standards. The work will focus on the practical challenges which have been encountered while deploying the available O-RAN software over a virtualized infrastructure, investigating the compatibility of its integration within a containerized environment orchestrated by *Kubernetes*.

Chapter 1

Introduction

In the last decade Software Defined Networking (SDN) and Software Defined Radio (SDR) paradigms took much more importance in the scenario of modern 5G cellular networks.

In fact the need to cope with heterogeneous 5G applicative scenarios and use cases, including performance enhancement for mobile broadband users, ultra-reliable low latency services and massively dense connectivity scenarios, increased the complexity of mobile radio networks and the need of flexibility in the struggle of keeping under control satisfying Quality of Experience (QoE) levels while possibly serving in a different way differentiated kind of services leveraging Network Slicing techniques [1].

For these reasons the research and a number of standardization bodies and industrial associations are moving from the monolithic black box approach of 4G cellular networks towards softwarization, virtualization and disaggregation of network functionalities [2].

Starting from these premises, new techniques and trends are currently under study and several 5G software-based projects and alliances are embracing the open source approach while addressing the introduction of SDN approaches at the edge of networks and in particular in the Radio Access Network (RAN) segment. Even if from one hand this trend produced new frameworks and libraries available to the wireless community, on the other hand the adoption of open source softwarization has led to a multitude of solutions whose interoperability and interactions are often unclear [3]. Figure 1.1 shows an high level view over the main frameworks involved in the context of current programmable 5G networks.

Regarding the RAN evolution, the trend is to move gradually towards the concept of *"Open RAN"*: the term Open RAN, which is pushed ahead by a number of major telecom operators and standardization bodies [4], extends both the concept of Centralized/Cloudified RAN (C-RAN) - a RAN in which specific functionalities can be centralized at the Base Band Unit (BBU) such that to have a wider view and a better control over the network while exploiting simpler and cheaper Remote Radio Headers (RRH) - and the concept of vRAN - a RAN in which specific functionalities can be abstracted from specialized black box hardware and designed through a Network Function Virtualization (NFV) approach in which softwarized network functions can be deployed over general purpose servers and hardware. In addition to including these two concepts, the Open RAN has two more important aspects: the disaggregation of the RAN architecture, for which functionalities are intended to be split among different logical blocks, and the introduction of open standards and interfaces among these building blocks [5].

Following the trend of an Open RAN, at the moment the most important contribution is the work brought ahead by the O-RAN Alliance consortium [6] and in this document the reference design proposed by such organization will be presented.

The work presented in this thesis aims to investigate the feasibility of the deployment of the O-RAN software on a virtualized infrastructure starting from the documentation currently made available by the open source community [7]. Moreover the work will highlight possible critical issues which could be encountered while approaching the integration of different infrastructural components and it will analyze in particular the role of *Kubernetes* in the orchestration of the virtualized resources [8]. Given the objectives of this work, the adopted approach will focus on practical aspects of the deployment from an early stage.

In Chapter 2 the O-RAN architecture will be presented, from the description of its building blocks and functionalities up to the current O-RAN related works and exemplar platforms. In Chapter 3, a possible solution for the deployment of a virtual infrastructure able to host O-RAN software components will be described, and the role of *Kubernetes* as Virtual Infrastructure Management (VIM) framework will be contextualized. Moreover, a detailed description of the procedural installation steps and choices adopted for the integration of such technologies over the virtualized environment will be shown. Chapter 4 then

will present the available path toward the deployment of specific softwarized O-RAN functional blocks referring to the currently available software release. Finally in Chapter 5 a performance assessment over the last deployed component will be done, highlighting in particular the performance achieved by the system for the installation of this component over the underlying containerized environment managed by *Kubernetes*.

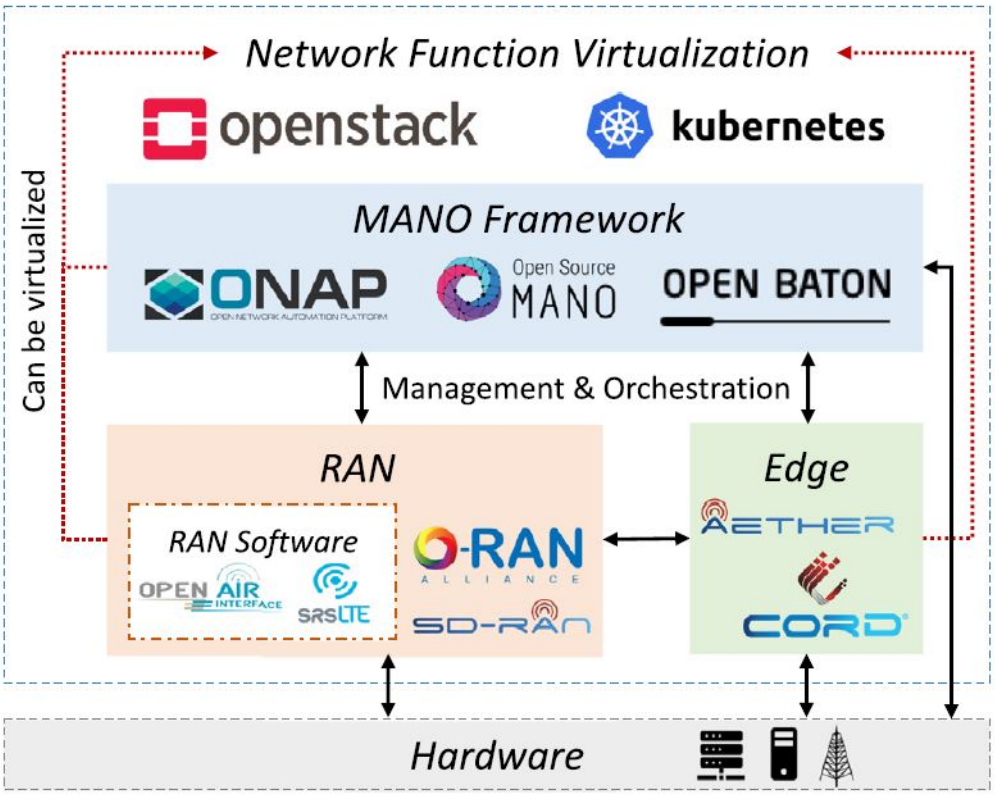


Figure 1.1: High-level relationship among MANO, RAN and edge frameworks, and virtualization components [3]

Chapter 2

O-RAN Alliance

2.1 Overview

O-RAN Alliance is an industrial consortium which aims to bring future Radio Access Networks (RAN) toward openness and intelligence, defining standardized open network interfaces and APIs and leveraging emerging deep learning techniques to embed intelligence in every layer of the RAN architecture [9].

Being founded in February 2018 by AT&T, China Mobile, Deutsche Telekom, NTT DOCOMO and Orange [6], O-RAN Alliance can be considered precursor and promoter of the new concept of **Open RAN** defined as *"a disaggregated RAN, which is subdivided into several independent systems by using open and interoperable protocols and interfaces"* [5].

Leveraging these premises, O-RAN *"aims to drive the mobile industry towards an ecosystem of innovative, multi-vendor, interoperable, and autonomous RAN, with reduced cost, improved performance and greater agility"* [10].

The key principles of the O-RAN Alliance are:

1. pushing the industry towards open and interoperable interfaces, leveraging RAN virtualization and AI enabled data-driven RAN Intelligence and automation;
2. maximizing the utilization of common-off-the-shelf (COTS) hardware;
3. specifying APIs and interfaces, driving standardization processes or evaluate open-source implementation where possible.

In the scenario of *"open and programmable"* future 5G software-defined cellular networks O-RAN represents one of the major solutions towards softwarization, virtualization and disaggregation of RAN functionalities.

In fact, well-defined interfaces between the different elements of the RAN enable the possibility to define an open and agile architecture in which any O-RAN component exposes the same APIs such that different implementations of the same functionality can be interchanged easily, letting operators and third-party entities to deploy differentiated services [3].

Moreover, following the trend of cloud-native infrastructures, O-RAN will enhance RAN disaggregation and virtualization by means of the definition of a specific type of Edge Cloud system, called O-Cloud, leveraging lightweight virtualization technologies and edge-cloud orchestration frameworks offered by open source communities like Kubernetes and Linux Foundation's Open Network Automation Paltform (ONAP) [9].

Finally it aims to bring intelligence to the RAN by means of an AI powered hierarchical controller structure which could enable different kinds of closed-control loops enhancing RAN automation at different time-scales [3].

2.2 Overall architecture of O-RAN

Following the principles introduced in Section 2.1, O-RAN relies on the disaggregated RAN architecture proposed by 3GPP for the Next Generation RAN (NG-RAN) [11] which is based on a functional split among three main entities: Centralized Unit (CU), Distributed Unit (DU) and Remote Radio Unit (RU) - as shown in Figure 2.1 [12].

In addition to this O-RAN defines and aims to standardize a set of open interfaces between these components and introduces the so called Radio Intelligent Controller (RIC) which embeds data-driven intelligence and can control some specific Radio Resource Management (RRM) functionalities (like Mobility Management or Interference Management) exposed by the rest of the processing units in the RAN while keeping available legacy RRM [13].

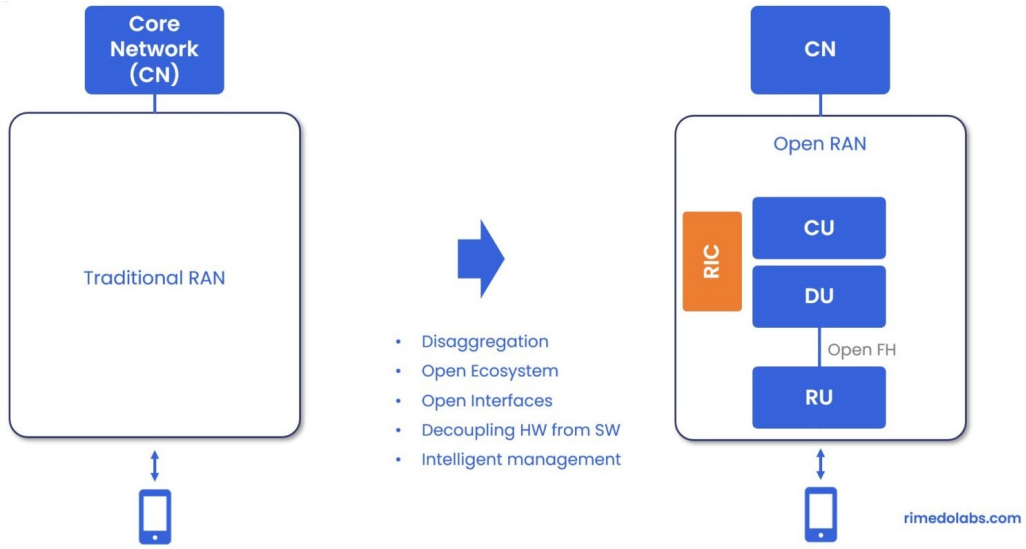


Figure 2.1: Open RAN - Network Transformation [12]

The functional split adopted by O-RAN for these three logical blocks is shown in Figure 2.2 and concerns:

- O-CU-CP which embeds the control protocols of layer 3 and terminates the 3GPP N2 interface towards Access and Mobility Function (AMF) of the 5G Core network (5GC);
- O-CU-UP which embeds data plane protocols of layer 3 and terminates the 3GPP N3 interface towards User Plane Function (UPF);
- O-DU which embeds layer 2 and High-PHY functions and terminates F1 3GPP interfaces towards respectively O-CU-CP and O-CU-UP;
- O-RU which is intended to enable intra-PHY Lower-Layer Split (LLS) and for which is under study an Open-Fronthaul interface in order to enable real-time control from O-DU and IQ samples data-transfer [14].

Moreover O-RAN introduces a hierarchical controller structure made up of two different RICs: the **Non-Real Time RIC (Non-RT RIC)** and the **Near-Real Time RIC (Near-RT RIC)**. The former is intended to bring to the RAN an higher level point of view to determine RAN optimization actions while the latter is intended to monitor and control via policies specific functionalities of the other processing nodes.

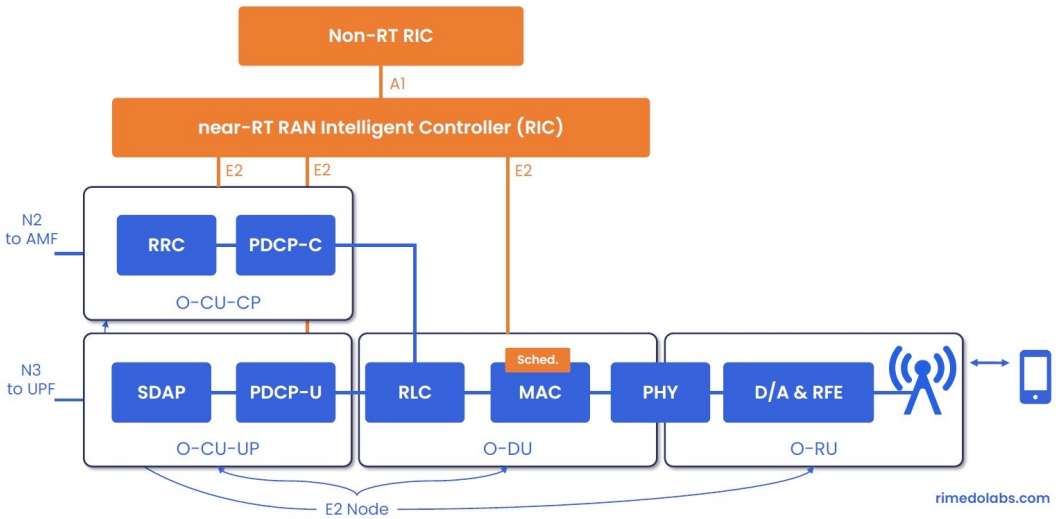


Figure 2.2: O-RAN Functional Split [12]

Finally the O-RAN architecture is intended to be integrated with the latest Long Term Evolution (LTE) E-UTRAN (Evolved UMTS Terrestrial Radio Access Network) and the current 5G NG-RAN (Next Generation RAN) in which New Radio (NR) gNBs and ng-eNBs - respectively 5G base stations and evolved LTE base stations to be deployed within 5G Core (5GC) - cooperate in non-stand-alone RAN deployments [11].

For this reason, to interconnect all these logical blocks and to make them interoperable, O-RAN alliance includes and specifies two well-defined and open control interfaces called A1 and E2: these two interfaces, similarly to 3GPP defined interfaces, permit to expose specific RAN functionalities and data towards and within the hierarchical controller structure and in particular:

- The **E2 interface** permits to connect the Near-RT RIC towards any node in the RAN which supports the E2 termination and uses O-RAN open APIs (like O-DUs, O-CUs, O-eNBs...), enabling the RIC to collect data and monitor/control specific RRM functionalities (like Radio Scheduling policies or handover thresholds for Mobility Management);
- The **A1 interface** supports different kinds of workflow messages between the two RICs *(for example enrichment data, policy-based guidance and ML management from Non-RT RIC to Near-RT RIC)* and represent the main component of the Non-RT RIC framework.

As shown in Figure 2.3 the E2 interface terminates in the previous defined logical blocks which can be either part of the O-Cloud infrastructure (see Subsection 2.2.1) - hosted in VMs and containers - or be proprietary sites like for example E-UTRAN eNBs exposing open interfaces [13]; for this reason in O-RAN specifications they are called **E2 Nodes**.

In Section 2.3 the hierarchical RIC structure and other two relevant management interfaces O1 and O2 will be described in details and three different time-scale control loops will be characterized.

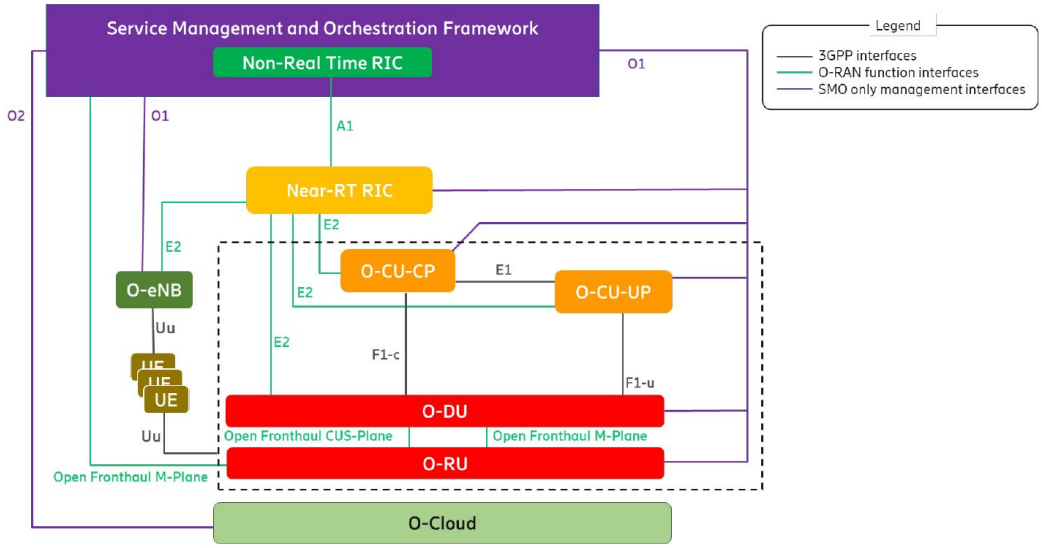


Figure 2.3: Overall Architecture and Open Interfaces [13]

2.2.1 O-Cloud Platform and Deployment Options

The O-RAN effort towards hardware and software disaggregation of the RAN is led by the definition of a specific kind of Edge-Cloud Platform called O-Cloud: this platform comprises all the physical infrastructure nodes capable to host O-RAN functions (like Near-RT RIC, O-DU, etc.) and all the supporting software components (e.g. OS, VM monitoring, container runtime) [15].

This kind of abstraction permits to characterize differentiated deployment options for the O-RAN architecture, even if most commonly is assumed the solution providing Near-RT RIC and O-CU located in Regional or Edge Cloud servers and the O-RU located at the Cell Site due to the tight relation with specialized radio hardware and latency stringent functionalities. Different solutions can then be assumed for the O-DU and depending by the use case it can be deployed either in physical resources near the Radio Unit or virtualized in an Edge Cloud server as shown in Figure 2.4 [3].

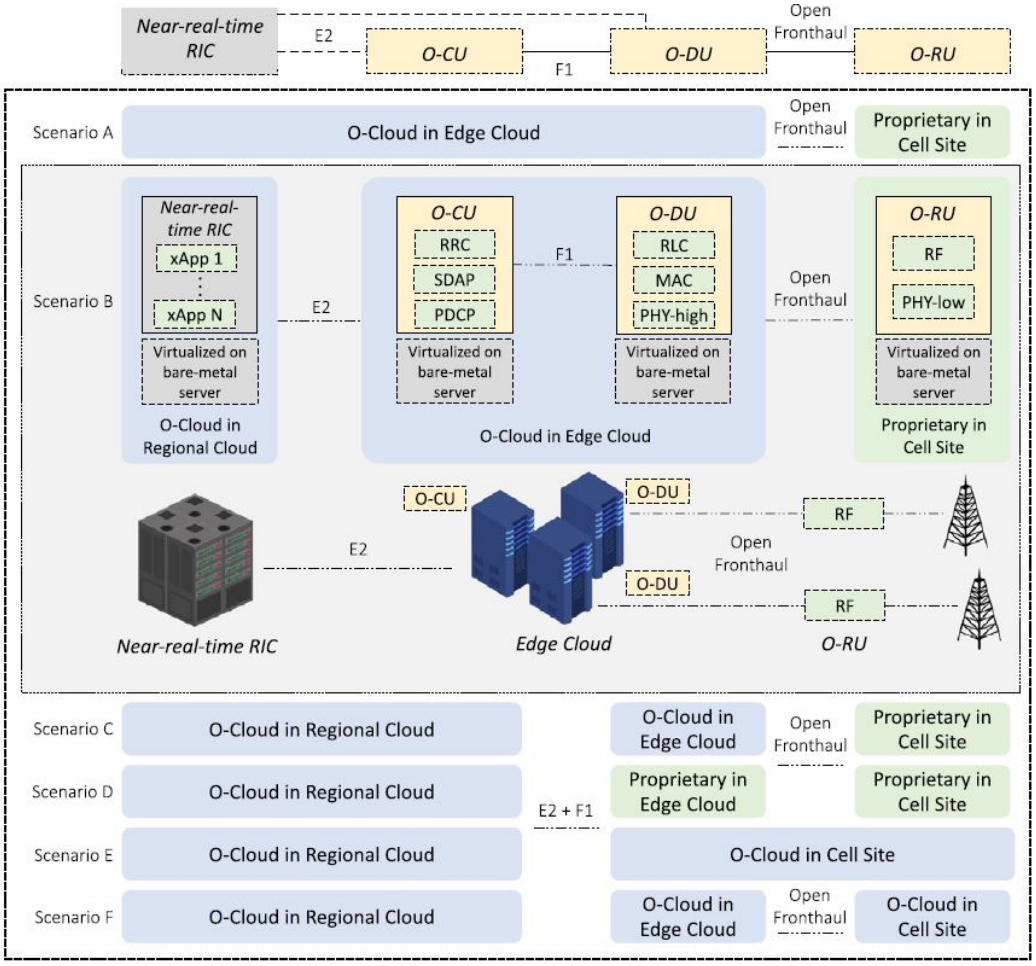


Figure 2.4: Logical and Physical deployment options for O-RAN. [3]

2.3 O-RAN RIC Structure and Control Loops

The O-RAN RIC hierarchical structure enables two of the three different control loops supported by O-RAN, the Non-Real Time control loop and the Near-Real Time control loop, but in practice all elements of the architecture can interact differently in all the three types of loops [13].

Basically, the effort of O-RAN Alliance moves toward separating radio resource management aspects depending on the time scale, introducing where possible value-added services like intelligent management and optimization.

In Figure 2.5 the control loops are defined based on the controlling entity and since typically these entities are meant to be part of different segments of the RAN they are expected to react in specific latency intervals, even if the timing of these control loops is mostly use case dependent.

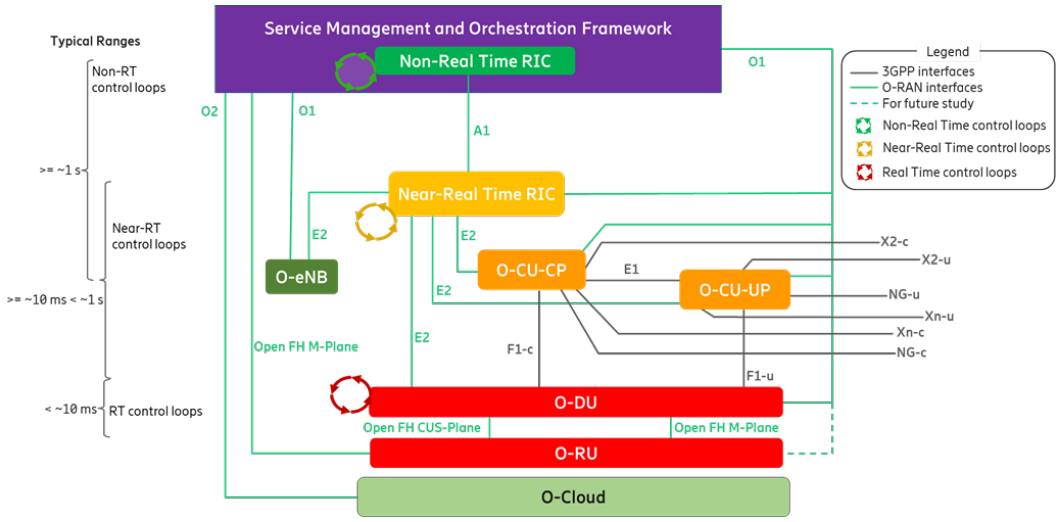


Figure 2.5: O-RAN Control Loops [13]

2.3.1 Real Time control loop

Being controlled by O-DU it concerns all the RRM actions which take place at MAC layer, like for example Dynamic Radio Resource assignment (Radio Scheduling), or at High-PHY layer, like Link Adaptation techniques such as Adaptive Modulation and Coding (AMC).

As told introducing Section 2.3 O-DU will not be the sole entity involved in the control loop; actually it continuously receives Channel Quality Indicators (CQIs) -estimations of the channel conditions, user throughput and spectrum usage- from User End-devices (UEs) and takes decision in order to choose for example the best modulation and coding schemes to be taken in order to satisfy specific performance requirements.

Moreover O-DU could receive specific control messages from O-CU through F1-c interface, for example on the desired resource allocation for data traffic [16], making O-DU de facto controlled by O-CU.

What's introduced by O-RAN is the possibility to expose part of the O-DU functions to the Near-RT RIC by means of the E2 interface Service Model [17].

Then depending on the specific use case and based on the functions exposed in the E2 Service Model, the Near-RT RIC may monitor, suspend/stop, override or control via policies the behavior of the controlled entity.

Finally the Real Time control loop, involving entities belonging to localities of the Edge Cloud near to radio headers, is typically characterized by latency intervals below 10 milliseconds.

2.3.2 Near-Real Time control loop

The Near-RT RIC is logically placed between the Service Management and Orchestration (SMO) layer and the underlying RAN disaggregated architecture. Typically is intended to be deployed at the top of the Edge Cloud or in Regional Cloud - refer to Subsection 2.2.1 for deployment options - and can be connected to multiple O-CUs, O-DUs and O-eNBs enhancing operational challenging functions such as per-UE controlled load-balancing, Resource Block (RB) management, interference detection and mitigation [9].

As mentioned introducing Section 2.2 this controller structure is compatible with legacy RRM and aims to introduce value-added services thanks to embedded intelligence and by means of the possibility to on-board modular control applications called **"xApps"** which can be also provided by third parties. Such kinds of applications are made of one or more microservices and are intended to monitor or control specific RAN functions in specific controlled E2 nodes [17].

The Near-Real Time control loop can then be enabled thanks to a fine grained RAN data collection from E2 interface and it is steered via policies and enrichment data provided via A1 interface from Non-RT RIC [13].

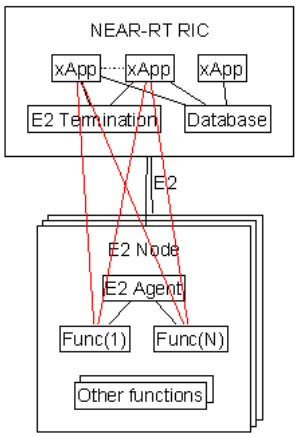


Figure 2.6: Near-RT and controlled Nodes [17]

Finally the data collected by the xApps are processed and can be fed to other xApps which could implement different services like, for example, the inference of a Machine Learning model in order to make predictions. Once a prediction is made then specific policies are defined and instantiated by other xApps such that configuration commands could be sent via E2 interface directly to CU/DU. This kind of workflow is just one of the possibilities enabled by the RIC hierarchical structure: different possible use cases are introduced in Section 2.5 and explained in details in [18]. In particular the *"QoE prediction use case"* shows a possible implementation involving Machine Learning (ML) models: such kind of solution requires that ML models should be trained offline in Non-RT RIC, while model inference is meant to be executed real-time in Near-RT RIC [10].

Due to its characteristics the Near-RT control loop is expected to react between 10 milliseconds up to 1 second.

2.3.3 Non-Real Time control loop

The Non-RT RIC is intended to enable an high level intelligent RAN optimization and it is co-located within the Service Management and Orchestration (SMO) framework - see Section 2.4. Even if it is logically separated by SMO framework, it represents a subset of SMO functionalities which are needed to enable the communication via A1 interface towards the Near-RT RIC.

In addition to this the Non-RT RIC can host a set of modular applications called "rApps" which, similarly to "xApps" for Near-RT RIC, provides value-added services like:

- Providing policy-based guidance and enrichment information across A1 interface;
- Performing data analytics, AI/ML training and inference for RAN optimization or for the use of other rApps;
- Recommending configuration management actions over O1 interface.

This modular structure is designed in order to make portable the Non-RT RIC and to make it independent from the SMO framework deployment.

Furthermore the Non-RT RIC, to enhance RAN optimization actions, may need to access specific SMO services, but how this can be done is not in the scope of O-RAN Alliance work. However there are particular SMO functionalities that the Non-RT RIC can leverage, like for example O1 and O2 interfaces defined by O-RAN:

- The **O1 interface** take care of Fault, Configuration, Accounting, Performance and Security (FCAPS) network management aspects between SMO and O-RAN network elements - refer to Section 2.4 for further details;
- The **O2 interface** is responsible for instantiation and life cycle management of Virtual Network Functions (VNFs) hosted by O-Cloud.

Even if such kind of network management functionalities are not meant to be part of the Non-RT RIC, they may be influenced by Non-RT RIC for the sole purpose of RAN resources optimization.

For this reasons the Non-RT RIC can enable the Non-Real Time control loop based on an high amount of data collected either from Near-RT RIC policies feedbacks, RAN data, network management interfaces and external cloud resources leveraging big-data analytics and ML/AI training/inference to determine RAN optimization actions.

Finally thanks to its high level point of view the Non-RT RIC can either provide policy-based guidance, ML management and enrichment information towards Near-RT RIC and activate RAN optimization actions with latency in the order of 1 second [19].

2.4 Service Management Aspects

As anticipated in Section 2.3 the Service Management and Orchestration (SMO) framework must ensure the Non-RT RIC to access specific functionalities related to RAN optimization actions like in particular collecting Performance Measurements (PM) through O1 and O2 interfaces.

In addition to this the SMO must take care of the orchestration of the Network Functions Virtualization Infrastructure (NFVI), managing the life cycle of O-RAN network elements (Near-RT RIC, O-CU, O-DU, O-RU) which can be either Virtual Network Functions (VNFs) hosted in specific location of the O-Cloud infrastructure or Physical Network Functions (PNFs) exposed by cell sites.

For non-virtualized parts, typically O-RU functionalities which are related to area coverage and need to be placed at cell sites, the SMO supports the deployment of physical network elements on dedicated physical resources with management through the O1 interface.

For virtualized network elements, the SMO has the capability to interact with the O-Cloud to perform network element life cycle management, for example it can instantiate the virtualized network element on the target infrastructure through the O2 interface or indicate the selected geo-location for each VNF to be instantiated.

Finally the Service Management and Orchestration framework must be able to support the communication between the deployed network elements and so it is in charge of IP addressing, network reconfiguration and system updates.

Then to guarantee various deployment solutions the Operation and Maintenance architecture defined by O-RAN describes in details the requirements needed such that the SMO framework can be provided by third-party Network Management Systems (NMS) or orchestration platforms like for example the Linux Foundation's Open Network Automation Platform (ONAP) [20].

2.5 O-RAN Use Cases

As anticipated in Subsection 2.3.2 with the example of the *"QoE prediction use case"* workflow, O-RAN Alliance provides within the reference design specifications also the description of an initial set of use cases each one with the purpose to demonstrate different capabilities of the architecture.

Some of these use cases - like *"QoE prediction use case"* - describe how the utilization of ML technologies can be exploited to deploy models using long term data along with policies to control real time behavior of RAN, some other refer to solutions for RAN optimization in which only policies and configurations mechanisms are involved, like for example the *"Traffic Steering use case"*.

As shown in Figure 2.7 O-RAN propose a variety of use cases, some of them very general like for example the *"RAN Sharing use case"* that aims to show the CAPEX reduction within the adoption of an O-RAN approach to this kind of scenario, some other concern the enhancement of particular functionalities like the previous cited *"Traffic Steering use case"* or the *"QoS Based Resource Optimization"* one, some other specifically address particular applications like for example *"Context Based Dynamic Handover Management for V2X"* or *"Flight Path Based Dynamic UAV Resource Allocation"* [10].

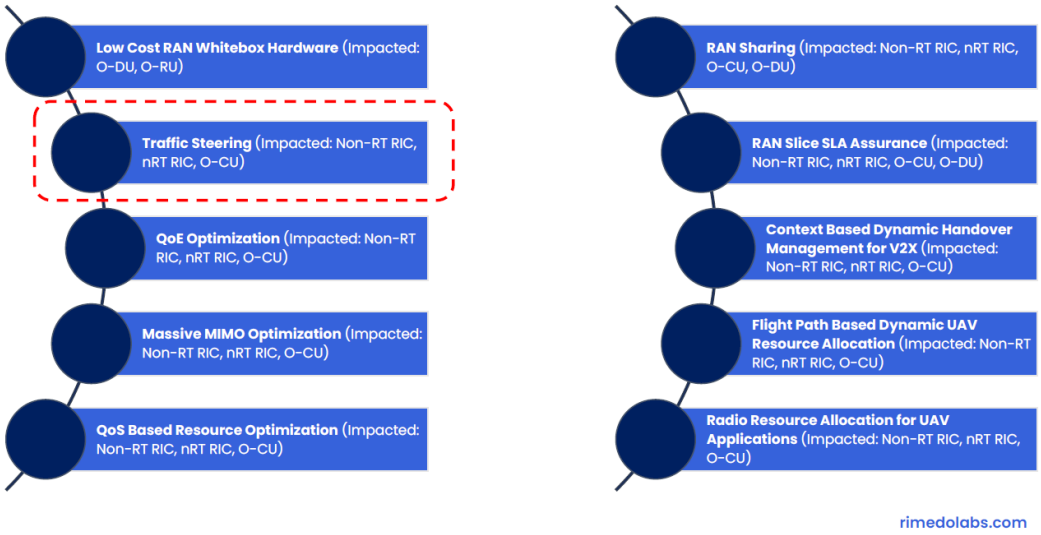


Figure 2.7: O-RAN Use Cases [10]

2.5.1 Traffic Steering Use Case

This particular use case proposes a solution to activate *per-UE load balancing* leveraging O-RAN capabilities to move from the typical cell-centric approach.

In fact traditional network traffic control optimization typically requires a lot of manual intervention and being a passive mechanism it's also characterized by slow feedback responses.

The workflow proposed by O-RAN for an efficient UE-centric traffic steering mechanism starts from a fine grained data collection via O1 interface from O-CU and O-DU towards the Non-RT RIC.

The collection of data and enrichment information then is expected to enable a long term data analytics at the Non-RT RIC in order to determine specific actions to be taken for different UEs, based on QoS requirements or S-NSSAI (Single-Network Slice Selection Assistance Information).

After that new policies are defined reporting which specific cell is expected to serve the specific UE, based on the preferred Radio Access Technology (RAT), frequency band and other access channel characteristics.

Finally these policies are fed via A1 interface towards the *Traffic Steering xApp* in the Near-RT RIC where they can be converted into control messages and sent towards E2 nodes [18].

It is possible to notice that two control loops are involved:

- The Non-RT loop which updates policies with a latency greater than 1s;
- The Near-RT loop which controls handovers with a latency in the order of 10ms.

This specific use case is reported as an example of the possible control procedures which O-RAN envisions to support in future open RAN deployments and shows how two different time scale control loops can be involved to jointly reach the desired optimization goal.

2.6 Exemplar Platforms

In order to contextualize the vision and the reference design promoted and developed by the O-RAN Alliance, in this Section the most relevant entities involved in the definitions and developments of O-RAN related works will be presented.

Figure 2.8 collocates O-RAN specifications as a derivation of specific 3GPP 5G Standards, in particular those one related to the RAN, like the functional

split among CU and DU, the definition of interfaces among them and the relation with the Service Management and Orchestration Framework (MANO) of Network Function Virtualization (NFV). As introduced in Section 2.2 in fact part of the work of the O-RAN group aims to align and integrate the proposed architecture within the 5G RAN reference design.

Starting from this O-RAN aims to provide additional features to the RAN, introducing the Radio Intelligent Controller functional blocks definitions within the SMO/Non-RT RIC framework and the Near-RT RIC framework, defining a set of new open and programmable interfaces (A1, E2, O1 and O2) and specifying the proposed Lower-Layer Split among DU and RU within the related Open-Fronthaul interface.

On the other hand the O-RAN Alliance reference design is being adopted by other organizations in order to develop O-RAN based exemplar platforms. For example the SD-RAN project of the Open Networking Foundation (ONF) aims to build an exemplar platform based on the realization of a Near-RT RIC (called µONOS-RIC) and a set of exemplar xApps to provide developers and operators with open source emulators on which testing O-RAN functionalities. Finally, O-RAN Alliance together with the Linux's Foundation created the O-RAN Software Community (OSC) that focus on the development of an open-source software reference design for the whole O-RAN as will be introduced in Chapter 3 [12].

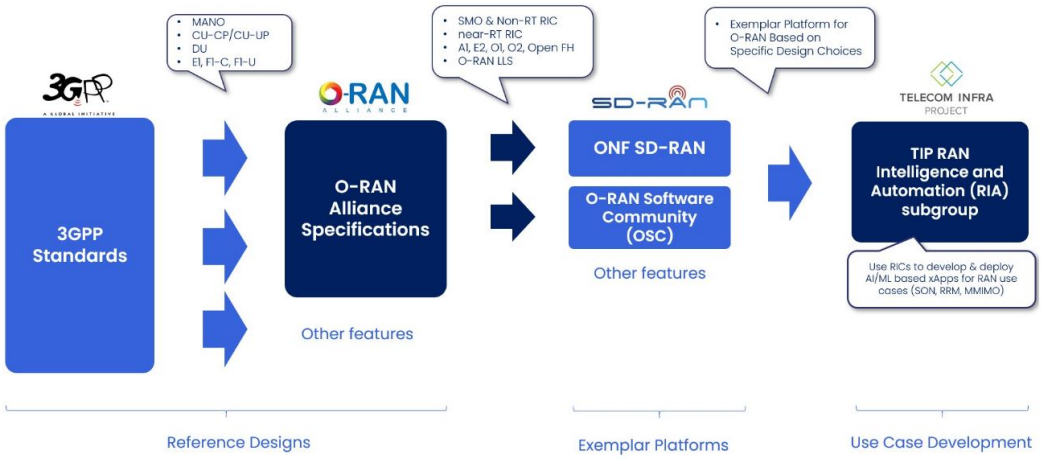


Figure 2.8: O-RAN Related Entities [12]

Chapter 3

Virtual Infrastructure and Management Frameworks

As introduced in Chapter 2, O-RAN Alliance provides a reference design for the Open RAN architecture they envisions for which they also specifies all the requirements needed to guarantee satisfying openness, security, intelligence and automation standards. In this Chapter an experimental approach for the deployment of the required software components will be presented and described in details as a possible testbed for further experimental demonstrations.

3.1 Open Platforms

3.1.1 O-RAN SC

Even if for the moment there's not a unique and standardized approach for the integration of O-RAN functionalities within the RAN, O-RAN Alliance, in collaboration with the Linux's Foundation, created the O-RAN Software Community (OSC) to support the creation of software for the Radio Access Network and to leverage other Linux's Foundation projects while addressing challenges like software scaling or 3GPP standards alignment [7].

Aiming to provide an open-source software reference design for the whole O-RAN, from November 2019 until now, the OSC released four versions of the software including in each new one support for additional components of the architecture. Even if first ones supported few functionalities, from the *"Bronze release"* - June 2020 - a lot of documentation and some procedures to

reach the deployment of basic components were released. Then starting from the *"Cherry release"* - out date December 2020 -, apart from the updated software reference for the simulation of the A1, E2 and O1 interfaces, there was a more complete support for different functionalities, starting from the demonstration of some use cases (like for example the policy-based Traffic Steering workflow using A1 and E2 interfaces) up to data gathering solutions for different interfaces or compatibility solutions for OAM issues [21]. Finally with the *"Dawn release"* - out date July 2021 - a better integration with the other software components was reached and, as will be shown in Chapter 4, at the moment results to be the more stable solution for the software deployment.

3.1.2 MANO/VIM Frameworks

Considering the *"open approach"* which O-RAN aims to provide to the telco industry, it's not specifically addressed which kind of solution to adopt in term of MANO frameworks, Virtual Infrastructure Managers (VIM) or Cloud Platforms, but for the moment among the open source projects providing such kind of frameworks is clear which one are more likely to be integrated within future O-RAN deployments. The first is the Linux's Foundation **Open Network Automation Platform (ONAP)** - as anticipated in Section 2.4 - which from the *Frankfurt release* - out date June 2020 - specifically addresses to be compatible with O-RAN specifications and, aiming to be a comprehensive platform for real-time, policy-driven orchestration and automation of physical and virtual network functions, it is recommended by the OSC as the first choice for SMO/Non-RT RIC framework deployment [22]. The second one is **Kubernetes (K8s)** to be exploited as a VIM, thanks to it's effort in providing an higly scalable platform to orchestrate containerized applications [8].

3.2 Deployment Choices

3.2.1 Reference Documentation

For the software development the OSC refers to the web-based collaborative platform *Confluence*, on which shares the source code and all the documentation needed for the installation of the last software releases [23]. In particular

it provides a *Getting Started* guide to easily reach a set up of O-RAN including SMO/Non-RT RIC framework, Near-RT RIC with Traffic Steering related xApps, A1 and O1 interfaces use case flow demonstration [24].

The guide is based on the OSC's projects documentation in [25], and in particular it refers to the *"Integration and Testing"* project's **it/dep** repository which hosts deployment and integration artifacts such as scripts, Helm charts, and other files used for deploying O-RAN basic components over a Kubernetes cluster.

However, even if the guide aims to demonstrate the deployment of the O-RAN software release, it does not permits to manage in a modular fashion the installation of the underlying virtual infrastructural modules (like Kubernetes and its package manager known as *Helm* [26]).

For this reason the choice was to follow the **it/dep** documentations in order to refer directly to the related *"Installation Guide"* [27], in particular to derive the requirements and to find an alternative way to deploy a kubernetes cluster.

This choice, apart from giving the possibility to obtain a customized installation, was made in order to keep track of possible installation problems while integrating up to date new releases of different software modules.

Regardless of the underlying infrastructure, the deployment of any O-RAN component present in the **it/dep** repository requires to be installed on top of a Kubernetes cluster with the following characteristics [27]:

- Kubernetes v.1.16.0 or above;
- helm v2.12.3 or above;
- Read-write access to directory /mnt.

So, to avoid the bare execution of the installation scripts provided by the OSC and with the intent of setting up each component in a modular way, once having derived the requirements from the *"Installation Guide"*, it was decided to proceed following a different guide provided by the ONAP Operation Manager (OOM) project for the deployment of a Kubernetes cluster. Such a guide refers to the deployment of the module SDN-C (Software Defined Network - Controller) on some VMs managed by a Kubernetes cluster composed of a master node and three worker nodes [28].

This choice was steered by the fact that starting from the OSC meeting of 12th August 2019 [29] was clarified the expected integration of the O-RAN

OAM Architecture into ONAP and so both the OSC project and some ONAP projects are working on such integration.

Finally the ONAP guide in [28] provides a step by step approach to the deployment of a simple Kubernetes cluster, making easier to keep track of the installation of each single infrastructural module.

3.2.2 The Host

Given the fact that - referring to the *Installation Guide* in [27] - the deployment can be done on a wide range of hosts, including bare metal servers, OpenStack VMs, and VirtualBox VMs, having the possibility to utilize an HP server with 72 logical cores and 256 GB of RAM, the choice was to instantiate some VMs directly on the server in order to recreate a local deployment and to have the possibility to derive reliable performance evaluations over the deployed components.

First of hall the OS requirements were derived from the prerequisites of the virtual infrastructural components needed to be installed on top of the host installation. Since the last versions of Kubernetes, Docker and Helm requires at least *Ubuntu 18.04 Bionic Beaver*, the adoption of such distribution was taken to create a simple OS image to be replicated in the deployment - refer to Appendix A for further details.

Then some VMs were instantiated thanks to Linux Kernel-based Virtual Machine module (KVM) - *a full virtualization solution for Linux on x86 hardware* - through which is possible to run multiple virtual machines running unmodified Linux images thanks to the fact that each VM is provided with private virtualized hardware [30]. In order to get remote access to the HP server then, a two hop port forwarding was needed such that it was possible to configure remotely the VMs through the *Virtual Machine Manager* interface.

In Appendix A some screenshot of the configuration on *Virtual Machine Manager* and the port forwarding commands needed for remote connection are reported, while in Figure 3.1 the final virtual infrastructure is represented graphically:

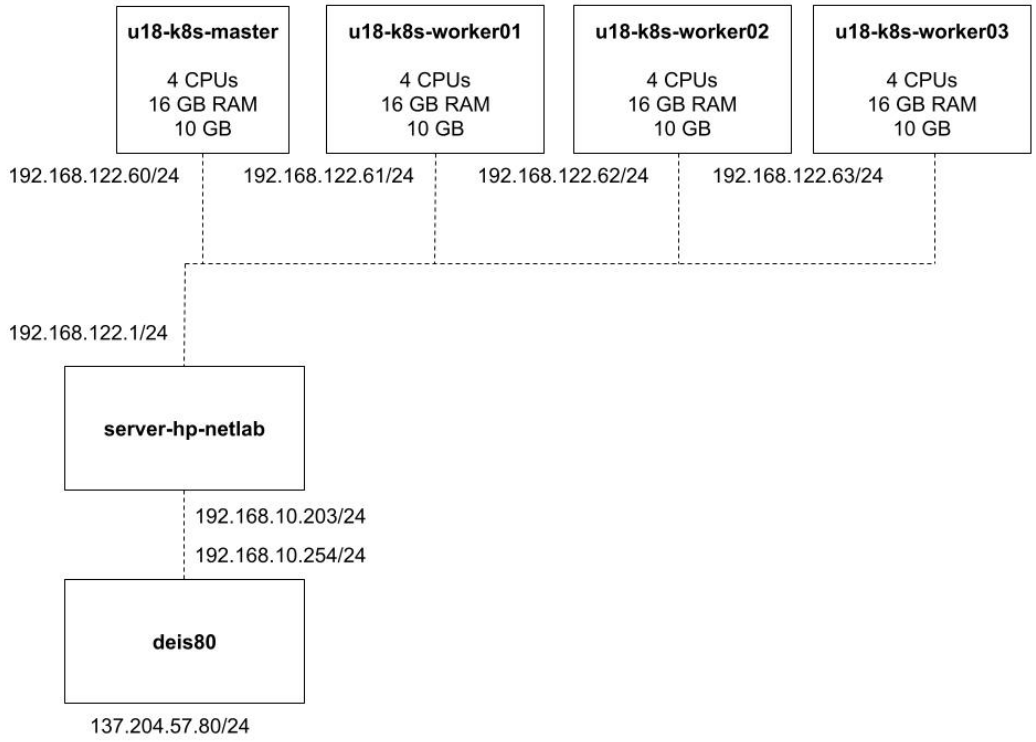


Figure 3.1: Virtual Infrastructure

As shown in Figure 3.1 the choice was to install four identical VMs on which allocate the minimum resource requirements in term of number of virtual CPUs, RAM memory and HDD storage, with the intent to understand the limit of the overall software installation.

3.2.3 Container Runtime and Kubernetes

The container runtime environment within Kubernetes management framework represents the virtualized platform on which each O-RAN logical component, considered as a set of containerized applications, needs to be installed on. As shown in Figure 3.2 for example each O-RAN building block (like the Near-RT RIC) is orchestrated by a Kubernetes cluster which is in charge to schedule resources like storage and containers for each different softwarized

functionality (colored blocks like the E2 Termination) managing also their lifecycle. Practically Kubernetes organizes resources to be scheduled in logical structures called *Pods* which are the execution environment for each different application [27].

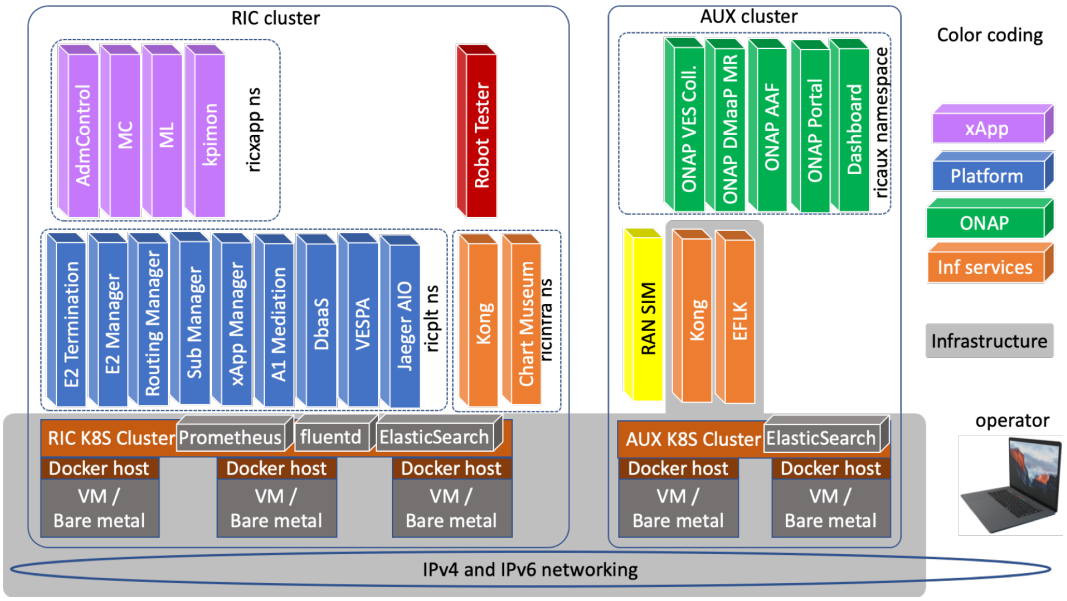


Figure 3.2: Installation Architecture [27]

Given these premises, Kubernetes has a crucial role in the orchestration and resource management aspects for the deployment of an O-RAN based solution over a virtualized infrastructure.

3.3 Installation steps

Starting from the VMs deployment presented in Subsection 3.2.2, the subsequent step was to setup each VM in order to host Kubernetes facilities following the previous cited ONAP guide in Subsection 3.2.1 [28] for reference.

3.3.1 VMs setup

In this Subsection are reported the procedural steps executed to **setup each VM**; most of them have to be executed on each VM, few ones only on the master:

(1) Add host names of kubernetes nodes (master and workers) to `/etc/hosts`:

```
ubuntu@u18-k8s-master:~$ sudo vim /etc/hosts

ubuntu@u18-k8s-master:~$ cat /etc/hosts
127.0.0.1 localhost
127.0.1.1 u18-k8s-master

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
127.0.0.1       host.minikube.internal
192.168.122.60  u18-k8s-master control-plane.minikube.internal
192.168.122.61 u18-k8s-worker01
192.168.122.62 u18-k8s-worker02
192.168.122.63 u18-k8s-worker03
```

(2) Turn off firewall and allow all incoming HTTP connections through `iptables` command:

```
ubuntu@u18-k8s-master:~$ sudo iptables -nL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy DROP)
target     prot opt source               destination
DOCKER-USER  all  --  0.0.0.0/0            0.0.0.0/0
DOCKER-ISOLATION-STAGE-1  all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0          0.0.0.0/0    ctstate RELATED,ESTABLISHED
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Notice: as shown input and output chain rules already accept connections. This step is crucial since integrated networking services among nodes and containers will leverage `iptables` proxy.

(3) Fix server timezone and select local timezone with command `sudo timedatectl set-timezone Europe/Rome` and check:

```
ubuntu@u18-k8s-master:~$ date
Fri Jul 23 17:19:03 CEST 2021
```

(4) Setup Network Time Protocol (NTP) server on your image if needed. It is important that all the VM's clocks are synchronized or it will cause problems joining kubernetes nodes to the kubernetes cluster:

```
sudo apt install ntp
sudo apt install ntpdate

ubuntu@u18-k8s-master:/etc$ cat ntp.conf
...
# Specify one or more NTP servers.

# Use servers from the NTP Pool Project. Approved by Ubuntu Technical Board
# on 2011-02-08 (LP: #104525). See http://www.pool.ntp.org/join.html for
# more information.
pool 0.ubuntu.pool.ntp.org iburst
pool 1.ubuntu.pool.ntp.org iburst
pool 2.ubuntu.pool.ntp.org iburst
pool 3.ubuntu.pool.ntp.org iburst

# Use Ubuntu's ntp server as a fallback.
pool ntp.ubuntu.com
...
```

Notice: It was left the already present configuration with the NTP Pool Project's servers.

(5) Update the VMs with the latest core packages and reboot:

```
sudo apt clean
sudo apt update
sudo apt -y full-upgrade

sudo reboot
```

(6) Some of the clustering scripts require JSON parsing, so install `jq` on the **master only**:

```
sudo apt install -y jq
```

(7) Finally it was possible to check time synchronicity by launching sequentially `date` command in all the three worker nodes through a cycle:

```
ubuntu@u18-k8s-master:~$ for i in 1 2 3; do echo $i; time ssh ubuntu@u18-k8s-
    worker0$i 'date'; done
1
```

```
Fri Jul 23 17:25:26 CEST 2021

real    0m0.612s
user    0m0.028s
sys     0m0.000s
2
Fri Jul 23 17:25:27 CEST 2021

real    0m0.591s
user    0m0.021s
sys     0m0.004s
3
Fri Jul 23 17:25:27 CEST 2021

real    0m0.596s
user    0m0.017s
sys     0m0.011s
```

3.3.2 Docker container runtime

Once the VMs were properly configured, the following step was to install the container runtime and the choice was to proceed **installing Docker Engine** by following Docker documentations in [31]:

(8) Installing Docker Runtime Environment:

Uninstall the Docker Engine, CLI, and Containerd packages:

```
sudo apt-get purge docker-ce docker-ce-cli containerd.io
```

Notice: Images, containers, volumes, or customized configuration files on your host are not automatically removed, so delete all images, containers, and volumes through:

```
sudo rm -rf /var/lib/docker
sudo rm -rf /var/lib/containerd
```

Update the `apt` package index and install packages to reach the repository over HTTPS:

```
sudo apt-get update
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

Add Docker's official GPG key:

```
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --
    dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

Use the following command to set up the stable repository:

```
sudo echo \
  "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
      https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list >
      /dev/null
```

Install docker engine and reboot:

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

sudo reboot
```

Verify Docker installation:

```
sudo docker run hello-world

ubuntu@u18-k8s-master:~$ sudo docker ps -a
CONTAINER ID IMAGE       COMMAND  CREATED     STATUS    PORTS NAMES
31367262b441 hello-world "/hello" 6 hours ago Exited (0)      adoring_hoover
```

3.3.3 Kubernetes packages

Finally, having installed and verified the container runtime environment on each VM, the last step before configuration was to **install Kubernetes packages** following Kubernetes documentations in [32]:

(9) To run K8s is necessary to disable swap in order to make sure that the `kubelet` agent will work properly.

So swap was disabled with the following command and by launching `free` command on each worker node it was possible to see no swap space allocated:

```
sudo swapoff /swapfile

ubuntu@u18-k8s-master:~$ for i in 1 2 3; do echo $i; ssh ubuntu@u18-k8s-
    worker0$i 'free -h'; done
1
```

```
              total        used        free      shared  buff/cache   available
Mem:           15G        739M        9.6G        796K        5.4G          14G
Swap:          0B          0B          0B
2
              total        used        free      shared  buff/cache   available
Mem:           15G        741M        9.6G        816K        5.4G          14G
Swap:          0B          0B          0B
3
              total        used        free      shared  buff/cache   available
Mem:           15G        742M        9.6G        816K        5.4G          14G
Swap:          0B          0B          0B
```

(10) Before the installation of kubernetes packages is important to verify that the MAC address and *product_uuid* are unique for every node.

(11) Another step before installation is to make sure that the network plugins will be able to see bridged traffic through the `iptables` proxy in order to manage the IP network among containers and pods. For this reason the `br_netfilter` module was loaded and consequently the `sysctl` configuration was changed as follow:

Check if the `br_netfilter` is loaded:

```
ubuntu@u18-k8s-master:~$ lsmod | grep br_netfilter
br_netfilter           24576  0
bridge                155648  1 br_netfilter

‘‘cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
br_netfilter
EOF’’
```

Configure `sysctl`:

```
‘‘cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF’’

ubuntu@u18-k8s-master:~$ sudo sysctl --system
* Applying /etc/sysctl.d/10-console-messages.conf ...
kernel.printk = 4 4 1 7
* Applying /etc/sysctl.d/10-ipv6-privacy.conf ...
net.ipv6.conf.all.use_tempaddr = 2
net.ipv6.conf.default.use_tempaddr = 2
* Applying /etc/sysctl.d/10-kernel-hardening.conf ...
kernel.kptr_restrict = 1
* Applying /etc/sysctl.d/10-link-restrictions.conf ...
fs.protected_hardlinks = 1
fs.protected_symlinks = 1
* Applying /etc/sysctl.d/10-lxd-inotify.conf ...
fs.inotify.max_user_instances = 1024
```

```
* Applying /etc/sysctl.d/10-magic-sysrq.conf ...
kernel.sysrq = 176
* Applying /etc/sysctl.d/10-network-security.conf ...
net.ipv4.conf.default.rp_filter = 1
net.ipv4.conf.all.rp_filter = 1
net.ipv4.tcp_syncookies = 1
* Applying /etc/sysctl.d/10-ptrace.conf ...
kernel.yama.ptrace_scope = 1
* Applying /etc/sysctl.d/10-zeropage.conf ...
vm.mmap_min_addr = 65536
* Applying /usr/lib/sysctl.d/50-default.conf ...
net.ipv4.conf.all.promote_secondaries = 1
net.core.default_qdisc = fq_codel
* Applying /etc/sysctl.d/99-sysctl.conf ...
* Applying /etc/sysctl.d/k8s.conf ...
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
* Applying /etc/sysctl.conf ...
```

(12) Installing `kubeadm`, `kubelet` and `kubectl`.

- `kubeadm`: the command in charge to initialize the k8s cluster in the master node, used also to attach other nodes to the cluster;
- `kubelet`: the component that runs on all of the machines in your cluster in charge to manage resources, instantiate Pods and Containers and manage their lifecycles;
- `kubectl`: the command line util to talk to your cluster, used to configure manually Pods, Deployments, Services and other k8s resources in the cluster.

Update the `apt` package index and install packages needed to use the Kubernetes `apt` repository:

```
sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates curl
```

Download the Google Cloud public signing key:

```
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://
    packages.cloud.google.com/apt/doc/apt-key.gpg
```

Add the Kubernetes `apt` repository:

```
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] https
    ://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.
    list.d/kubernetes.list
```

Update apt package index, install kubelet, kubeadm and kubectl, and pin their version:

```
sudo apt-get update

sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

(13) Configuring a cgroup driver:

- cgroups (a.k.a. Control groups) are Linux kernel tools used to constrain resources that are allocated to processes. Typically Linux distributions use systemd as init system, which also act as a cgroup manager.

Notice: Matching the container runtime and kubelet cgroup drivers is required or otherwise the kubelet process will fail.

Given these premises, since a single cgroup manager simplifies the view of what resources are being allocated, the choice was to configure systemd as cgroup driver for both the container runtime and the kubelet agent:

Create (or edit) the /etc/docker/daemon.json configuration file and include the following:

```
ubuntu@u18-k8s-master:~$ sudo cat /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"]
}
```

Restart Docker and check:

```
sudo systemctl restart docker

ubuntu@u18-k8s-master:~$ sudo docker info | grep "Cgroup Driver"
 Cgroup Driver: systemd
```

Then since the kubelet agent is configured by default to use systemd as cgroup driver there was no need of further configurations.

(14) If there was a previous K8s installation:

Make sure /etc/kubernetes/ directory to be empty:

```
sudo rm -rf /etc/kubernetes/*
```

Check if k8s ports are not already in use:

For example it was found an instance of kubelet already listening over port 10250 so it is was decided to reset the service:

```
ubuntu@u18-k8s-master:~$ sudo lsof -nPi | grep kubelet
kubelet   8580   root   13u  IPv4 147813    0t0  TCP 127.0.0.1:42111 (LISTEN)
kubelet   8580   root   16u  IPv4 145982    0t0  TCP
    192.168.122.60:52670->192.168.122.60:6443 (ESTABLISHED)
kubelet   8580   root   29u  IPv4 147824    0t0  TCP 127.0.0.1:10248 (LISTEN)
kubelet   8580   root   34u  IPv6 146046    0t0  TCP *:10250 (LISTEN)

sudo kubeadm reset

sudo systemctl restart kubelet
```

Conclusion: As told at the beginning of this Section, these steps need to be executed on each VM, while in Subsection 3.3.4 the specific configuration of the master node will be shown.

3.3.4 Master Node Configuration

This Subsection, starting from the ONAP guide cited in Subsection 3.2.1 [28], introduces the steps needed to configure the master node of the proposed K8s cluster. The master node in a cluster, is a node which hosts the control plane functionalities needed to orchestrate and control the operation of the cluster, by exposing interfaces and APIs in order to deploy, schedule and manage the lifecycle of containers [33].

(1) To setup the K8s master node it was sufficient to launch the command below:

```
ubuntu@u18-k8s-master:~$ sudo kubeadm init
...
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

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

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each
    as root:
```

```
kubeadm join 192.168.122.60:6443 --token rkgi2l.ayu7qs8gxci5r9c7 \
        --discovery-token-ca-cert-hash sha256:
            e77ee7ca568f89a0b2b26f46cfa36f46e74c3627664824ca38b5cf3640355572
```

Notice: the reported lines of the output log show the token needed to attach worker nodes to the cluster, for this reason is important to save this log in a file. However, for security reasons, the token will expire in 24 hours. In Subsection 3.3.5 will be shown how to regenerate the token.

In order to start using the cluster as a regular user, the suggested commands were launched:

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

Then, after the initialization of all the control plane *Pods*, it was possible to check what the `kubeadm` agent actually deployed:

```
ubuntu@u18-k8s-master:~$ sudo kubectl get pods -A
NAMESPACE     NAME                          READY   STATUS    RESTARTS   AGE
kube-system   coredns-558bd4d5db-7bvf5      0/1     Pending   0          3h39m
kube-system   coredns-558bd4d5db-fxt2h      0/1     Pending   0          3h39m
kube-system   etcd-u18-k8s-master           1/1     Running   0          3h40m
kube-system   kube-apiserver-u18-k8s-master 1/1     Running   0          3h39m
kube-system   kube-controller-manager-u18-k8s-master 1/1 Running   0      3h39m
kube-system   kube-proxy-txfdk              1/1     Running   0          3h40m
kube-system   kube-scheduler-u18-k8s-master 1/1     Running   0          3h39m
```

Notice: The K8s version installed, by default install the add-on *CoreDNS* which is a flexible, extensible DNS server that can serve as the Kubernetes cluster DNS.

Other K8s control plane services are respectively [8]:

- *etcd*: a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data;
- *kube-apiserver*: represent the front end of the K8s control plane, exposing APIs such that to enable interaction toward and within the cluster;
- *kube-controller-manager*: a daemon that embeds the core control loop which watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state. Practically it is the entity which enable automatic control over the cluster;

- *kube-scheduler*: the process in charge to schedule resources to different applications. In practice each application is intended to run within a set of *Pods* - i.e. the abstraction of the set of resources (containers, volumes) required by the application itself - and so the scheduler determines which Nodes are valid placements for each Pod in the scheduling queue according to constraints and available resources in term of CPU and memory.

Notice: *kube-proxy* is a network proxy that runs on each node in the cluster. It maintain network rules in the node enabling pods to communicate each other in the cluster and letting services to be exposed inside and/or outside the cluster.

Once having the control plane *Pods* up and running, by following the guide in [28], the subsequent steps were to intall a *Pod Network add-on* - needed to let Pods communicate with eachother - and to install the latest version of *Helm package manager*.

(2) For the Pod Network the choice was arbitrarily and the *Weave Net plugin* was installed following these steps:

Download the configuration file:

```
ubuntu@u18-k8s-master:~$ wget -O weaver.yaml "https://cloud.weave.works/k8s/
    net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
```

Apply the configuration:

```
ubuntu@u18-k8s-master:~$ sudo kubectl apply -f weaver.yaml
serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.apps/weave-net created
```

In order to verify that the Pod Network is up, it's possible to check the new Pod deployed and notice that now the two CoreDNS Pods are running:

```
ubuntu@u18-k8s-master:~$ sudo kubectl get pods -A
NAMESPACE     NAME                          READY   STATUS RESTARTS   AGE
kube-system   coredns-558bd4d5db-7bvf5  1/1     Running   0        34d
kube-system   coredns-558bd4d5db-fxt2h  1/1     Running   0        34d
kube-system   etcd-u18-k8s-master       1/1     Running   0        34d
kube-system   kube-apiserver-u18-k8s-master 1/1 Running   0        34d
kube-system   kube-controller-manager-u18-k8s-master 1/1 Running   0  34d
```

```
kube-system   kube-proxy-txfdk           1/1     Running   0       34d
kube-system   kube-scheduler-u18-k8s-master 1/1 Running   0       34d
kube-system   weave-net-t8z5l            2/2     Running   1       34s
```

(3) Intalling Helm from script:

The official *Installation Guide* for Helm provide an installation script that automatically dowload and install the latest version [34]:

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/master/
    scripts/get-helm-3

ubuntu@u18-k8s-master:~$ chmod 700 get_helm.sh

ubuntu@u18-k8s-master:~$ ./get_helm.sh
Downloading https://get.helm.sh/helm-v3.6.3-linux-amd64.tar.gz
Verifying checksum... Done.
Preparing to install helm into /usr/local/bin
helm installed into /usr/local/bin/helm

ubuntu@u18-k8s-master:~$ helm version
version.BuildInfo{Version:"v3.6.3", GitCommit:"
    d506314abfb5d21419df8c7e7e68012379db2354", GitTreeState:"clean", GoVersion
    :"go1.16.5"}
```

Notice: Differently from older versions, starting from Helm 3.0.0, there was no more need of installing Tiller - *the server side of Helm*. The removal of Tiller was chosen for security reasons: while in a first phase Tiller was helpful to allow first-time users to start experimenting with Helm and Kubernetes in a multi-tenant shared cluster scenario in which the same set of releases was shared with Tiller, now it's possible to simply fetch Helm release information from the Kubernetes API server [35].

3.3.5 Join Worker nodes to cluster

Once having initialized the master node the next step was to join to the cluster the other three worker nodes.

Please notice that this step was not a requirement for the deployment of any component present in the **it/dep** repository of the *Integration and Testing* OSC's project, but the choice of maintaining three additional worker nodes was taken in order to see how Kubernetes facilities would behave orchestrating O-RAN components over a multi-node cluster with lack of resources in order to discover possible critical situations or limits of the virtual infrastructure manager framework.

In order to join nodes to an existing cluster there is the need of a bidirectional trust procedure made up of a discovery phase, in which the worker node trust the Kubernetes Control Plane, and a TLS bootstrap phase, in which the Kubernetes Control Plane trust the worker node. `kubeadm` tool provides the possibility to realize this procedure thanks to a shared token:

First check if the token can be retrieved by command on master:

```
kubeadm token list
```

If there is no output of the above command then there is the need to generate the token again from master:

```
ubuntu@u18-k8s-master:~$ kubeadm token create --print-join-command
kubeadm join 192.168.122.60:6443 --token mdiw0c.c1u9x190x7kuwd3w
--discovery-token-ca-cert-hash
sha256:e77ee7ca568f89a0b2b26f46cfa36f46e74c3627664824ca38b5cf3640355572
```

Notice: The token generated will last 23 hours.

Then it is sufficient to launch the generated command on any of the worker nodes.

Finally it is possible to check that new nodes are connected directly on the master:

```
ubuntu@u18-k8s-master:~$ kubectl get nodes -A
NAME               STATUS   ROLES                  AGE     VERSION
u18-k8s-master     Ready    control-plane,master   39d     v1.21.3
u18-k8s-worker01   Ready    <none>                 5m37s   v1.21.3
u18-k8s-worker02   Ready    <none>                 112s    v1.21.3
u18-k8s-worker03   Ready    <none>                 56s     v1.21.3
```

Notice: For any worker node which joins the cluster, two new Pods: one `weaver-net` and one `kube-proxy`) are deployed on the master node and assigned to the worker:

```
ubuntu@u18-k8s-master:~$ kubectl get pods -A
NAMESPACE     NAME                          READY   STATUS    RESTARTS   AGE
kube-system   coredns-558bd4d5db-7bvf5      1/1     Running   0          39d
kube-system   coredns-558bd4d5db-fxt2h      1/1     Running   0          39d
kube-system   etcd-u18-k8s-master           1/1     Running   0          39d
kube-system   kube-apiserver-u18-k8s-master 1/1 Running   0          39d
kube-system   kube-controller-manager-u18-k8s-master 1/1 Running  0  39d
kube-system   kube-proxy-2kgxw              1/1     Running   0          10m
kube-system   kube-proxy-llrbh              1/1     Running   0          5m40s
kube-system   kube-proxy-sm9qx              1/1     Running   0          6m36s
kube-system   kube-proxy-txfdk              1/1     Running   0          39d
kube-system   kube-scheduler-u18-k8s-master 1/1 Running   0          39d
kube-system   weave-net-dsjn4               2/2     Running   1          6m36s
```

```
kube-system   weave-net-nzpr7          2/2     Running   1          10m
kube-system   weave-net-sp278          2/2     Running   0          5m40s
kube-system   weave-net-t8z5l          2/2     Running   1          4d5h
```

3.4 Considerations

This Chapter aimed to describe a particular approach to setup the prerequisites needed to start using O-RAN software components in a specific virtualized environment.

The choice was to deploy the system on a virtual infrastructure made up of four nodes with limited resources in order to possibly analyze the behavior of management software components like Kubernetes in not ideal conditions. Moreover the lack of available standardized configuration procedures for the overall system led to the choice to discover a path toward the deployment of such components which was as modular as possible in order to be possibly adopted as a starting point for further works or to be tailored based on different requirements.

Finally, the choice to adopt the most recent releases of all the software components highlighted the fact that the integration among them is supported and permits to start deploying O-RAN software modules on the base of an up to date underlying virtualized environment.

As it will be shown in Chapter 4 a different consideration must be done over the O-RAN software release. At the moment in fact the development of the O-RAN software reference is still under work in order to be fully integrated with the latest releases of all the other infrastructural modules.

Chapter 4

O-RAN Software deployment

As anticipated in Chapter 3 the choice made for the deployment of the O-RAN software components developed by the OSC was to start from the **it/dep** repository of the *Integration and Testing* project.

4.1 Integration and Testing project

The *Integration and Testing* project is focused on testing the requirements documented in each release, either for end to end demonstrations and for use case testing. In particular its **it/dep** repository provides deployment and integration artifacts such as scripts, Helm charts, and other files for the installation of the *Cherry release* and of the *Dawn release* of the O-RAN software and time by time is being updated in order to adjust the software reference with bug corrections and in order to align the current release with the new versions of the other infrastructural components, like for example introducing the compatibility with *Helm 3.0.0* [36].

At the moment the **it/dep** repository contains:

- Deployment scripts for a 1-node Kubernetes cluster;
- Deployment scripts and Helm charts for Near-RT RIC Platform;
- Deployment scripts and Helm charts for infrastructure services supporting the Near-RT RIC Platform (like services to expose APIs outside the cluster, for example to exhange data with the SMO/Non-RT RIC framework on another cluster);

- Deployment scripts and Helm charts for auxiliary services (like ONAP management functionalities as Virtual Event Streaming (VES) collectors for performance metrics);
- Deployment scripts for O-DU high project;
- Deployment scripts for SMO/Non-RT RIC Platform.

Please notice that, at the moment, the available support documentation for the O-RAN related software is still incomplete and there's not a unique approach toward the deployment of all the different components. The software community in fact is still updating the online documentation with the troubleshoot of different problematic situations, like for example the fact that some specific components or Helm charts result to be deprecated, but at the moment for some of the O-RAN components related projects it is still missing a reference (e.g. descriptions of E2 Termination or E2 Manager components).

Given these premises, the choice was to start deploying the Near-RT RIC software components referring to the **it/dep** *Installation Guide* in [27] with the intent to demonstrate their deployment feasibility and to keep track of all the possible installation issues.

Then in Section 4.4, the deployment procedures to on-board and install *xApps* over the RIC Platform Cluster will be shown.

4.2 RIC Platform Cluster

As shown in Chapter 3 each O-RAN building block is represented by a set of Kubernetes resources collected and deployed over a cluster. Referring to Figure 3.2 the components of the Near-RT RIC are collected under the RIC Platform cluster and they can be deployed exploiting RIC Platform deployment artifacts present in the **it/dep** repository.

To organize the deployments artifacts of the different components, the various RIC Platform components are placed into two groups: infrastructure and platform, respectively denoted as *ric-infra* and *ric-platform* groups [37].

The ***ric-infra*** group is expected to be deployed in each cluster. It consists of components such as *Kong Ingress Controller* or additional various functionalities like *Prometheus Alert Manager* or *Prometheus Server* - a service to

collect performance metrics in containerized environments [38]. In particular the *Kong Ingress Controller* proxies incoming API calls into the cluster and makes service APIs provided by Kubernetes resources accessible at the cluster node IP and port via a URL path [27]. For its characteristic the *Kong Ingress Controller* is exploited for cross-cluster communications and to expose services, but also to let the `kubelet` agent to exchange data within the cluster through HTTP/REST methods or to enable the developer to access Kubernetes resources with `curl` command.

The ***ric-platform*** group is deployed in the RIC cluster and consists of all Near-RT RIC Platform components, including:

- DBaaS: realized with a single container running Redis database provides a backend service for Shared Data Layer (SDL). Typically exploited for sharing the state data of an application within a namespace;
- E2 Termination;
- E2 Manager;
- A1 Mediator: listens for policy type and policy instance requests sent via HTTP (the "northbound" interface), and publishes those requests to running xApps via RMR messages (the "southbound" interface);
- Alarm Manager: is responsible for managing alarm situations in RIC cluster applications and interfacing with northbound applications such as *Prometheus Alert Manager* towards the SMO;
- Routing Manager: is responsible for distributing routing policies among the other platform components and xApps;
- Subscription Manager: is responsible for managing E2 subscriptions from xApps to the E2 Node (eNodeB or gNodeB);
- App manager: tool for deploying and managing various RIC xApp applications through HTTP/REST methods;
- xApp On-Boarder: provides the xApp On-Boarding service to operators. It consumes the xApp descriptor and optionally additional schema file, and produces xApp Helm charts;

- VESPA Manager: adapts near-RT RIC internal statistics' collection using *Prometheus Server* to scrape metrics from platform and xApp microservices and forward them to ONAP collectors, for example in the SMO framework;
- Jaeger Adapter: manager for kubernetes Custom Resources - extension of kubernetes API;
- O1 Mediator: support for the O1 interface;
- InfluxDB(optional): provided a persistent data storage in previous releases. At the moment NFS Server Provisioner needed component is deprecated.

4.3 RIC Platform Installation

Approaching the installation of the RIC Platform the choice was to follow the **it/dep** *Installation Guide* in [27] having already installed the kubernetes cluster described in Chapter 3 which is different from the cluster proposed by the OSC.

First of all the cluster installed in this work consists of a master node and three worker nodes while the solution proposed by the *Integration and Testing* project consists of a 1-node kubernetes cluster. Secondly the choice taken was to install the most recent versions of Docker, Kubernetes and Helm, while, complying with the **it/dep** requirements, specific versions of such software components are recommended for the installation.

Given these premises, the first step was to download the **it/dep** repository:

```
ubuntu@u18-k8s-master:~$ git clone https://gerrit.o-ran-sc.org/r/it/dep
Cloning into 'dep'...
remote: Counting objects: 3, done
remote: Total 5116 (delta 0), reused 5116 (delta 0)
Receiving objects: 100\% (5116/5116), 2.41 MiB | 1.59 MiB/s, done.
Resolving deltas: 100\% (2039/2039), done.
```

Notice: The previous command creates the `dep` directory in the file system, cloning inside it the set of installation scripts and support files needed for the deployment.

Then to download the specific configuration recipes and artifacts related to the RIC Platform, the command below was required in order to recursively

update the repository:

```
ubuntu@u18-k8s-master:~$ cd dep/

ubuntu@u18-k8s-master:~/dep$ git submodule update --init --recursive --remote
Submodule 'ric-dep' (https://gerrit.o-ran-sc.org/r/ric-plt/ric-dep) registered
     for path 'ric-dep'
Cloning into '/home/ubuntu/dep/ric-dep'...
Submodule path 'ric-dep': checked out '
    cffb2d9d3bf96bc49d0b48ffc2483151ca73115e'
```

After that it was possible to see which configuration files have been downloaded in the repository:

```
ubuntu@u18-k8s-master:~$ ls dep/RECIPE_EXAMPLE/PLATFORM
example_recipe.yaml
example_recipe_oran_cherry_release.yaml
example_recipe_oran_dawn_release.yaml
```

Notice: The latest version of the repository consists of a Master branch for the installation of the last stable version of the software, the *Cherry release*, and another branch with the most recent updates for the deployment of the *Dawn release*.

At this point it was required to modify the recipe files with the host IP in order to make possible the access to RIC Platform K8s resources using this address:

```
ubuntu@u18-k8s-master:~$ host u18-k8s-master
u18-k8s-master has address 127.0.1.1
u18-k8s-master has address 192.168.122.60

ubuntu@u18-k8s-master:~$ vim dep/RECIPE_EXAMPLE/PLATFORM/
    example_recipe_oran_cherry_release.yaml
...
extsvcplt:
  ricip: "192.168.122.60"
  auxip: "10.0.0.1"
...
...
dbaas:
  ...
  ...
  # Enable pod anti affinity only if you have more than 3 k8s nodes
  enablePodAntiAffinity: true
  ...
```

Notice: Since in this work it will not be deployed the AUX Cluster - a cluster containing auxiliary services - there was not the need to specify its IP address. Moreover, as suggested by the comment in the `dbaas` global settings, Pods Anti-affinity was enabled - this function will permit the *kube-scheduler* to assign

Pods to specific Nodes based on which Pods are currently running on it.
The recipe modifications were applied in both the two configuration files: `example_recipe_oran_cherry_release.yaml` and `example_recipe_oran_dawn_release.yaml`.

Once configured properly the recipe files, they have to be passed as inputs to the deployment scripts provided by the software community in the `dep/bin` directory. In particular the `deploy-ric-platform` script prepares common files and sets up an Helm local repository, then launches the installation script `dep/ric-dep/bin/install`. This second installation script prepares some global variables and configures some kubernetes resources needed before the installation - like the namespaces needed to subdivide resources into virtually isolated subgroups inside the cluster - and finally it launches recursively the `helm install` command for the deployment of each RIC Platform component.

As will be shown in the following Subsections, the installation script will deploy in the `ricplt` namespace all the components cited in Section 4.2.

4.3.1 Cherry Release (Master branch)

For the installation of the *Cherry release* there was the need to pass as input the path to the related configuration recipe while launching the deployment script:

```
ubuntu@u18-k8s-master:~$ cd dep/bin/

ubuntu@u18-k8s-master:~/dep/bin$ nohup sudo ./deploy-ric-platform ../
    RECIPE_EXAMPLE/PLATFORM/example_recipe_oran_cherry_release.yaml > deploy-
    ric-platform-cherry-log.out 2>&1 &
```

Notice: The choice was to launch the command using the `nohup` tool to avoid interruptions during its execution. With this tool it is also possible to redirect the standard output in a file in order to check possible error messages.

After the execution of the installation script it was possible to check the current Pods deployment:

```
ubuntu@u18-k8s-master:~/dep/bin$ kubectl get pods -n ricplt
NAME                                            READY STATUS   RESTARTS  AGE
deployment-ricplt-a1mediator-866d459c54-xw58m    1/1   Running  0         77s
deployment-ricplt-alarmmanager-757c64c75c-zfd52  1/1   Running  0         35s
deployment-ricplt-appmgr-79db86497b-kw5xq        1/1   Running  0         111s
deployment-ricplt-e2mgr-56f7c9887d-5zwfd         1/1   Running  0         94s
deployment-ricplt-e2term-alpha-5ff4df546-q8rj4   0/1   Running  1         86s
deployment-ricplt-jaegeradapter-5bf9b64956-pdcm7 1/1   Running  2         52s
```

```
deployment-ricplt-o1mediator-7bb959ccd6-2msdh     1/1   Running  0          43s
deployment-ricplt-rtmgr-5b7bc745f9-9fs2d          1/1   Running  0          103s
deployment-ricplt-submgr-8f56cb6cb-d7vvl          1/1   Running  0          69s
deployment-ricplt-vespamgr-bc9696b54-9zz6p        1/1   Running  0          60s
deployment-ricplt-xapp-onboarder-7d5657b97c-xbspc 2/2   Running  0          2m
r4-infrastructure-kong-599499fbd8-gjvgp           1/2 ImagePullBackOff 0   2m17s
r4-infrastructure-prometheus-alertmanager-7cc48c5988-gk97 2/2 Running 0   2m17s
r4-infrastructure-prometheus-server-7f74bdfc6d-ns5l7 1/1 Running       0   2m17s
ricplt-influxdb-meta-0                            0/1   Pending  0          26s
statefulset-ricplt-dbaas-server-0                 1/1   Running  0          2m8s
```

Notice: from the list of Pods it is possible to derive some considerations:

- InfluxDB Pod is `Pending`: since, for deployment choice, the optional Persistent Volume storage was not initialized, the scheduling mechanism is blocked since it cannot found any node compatible with the deployment. By calling `kubectl describe`, looking at the `Event` section of the output, it is possible to see the warning message from the scheduler:

```
ubuntu@u18-k8s-master:~/dep$ kubectl describe pod ricplt-influxdb-
    meta-0 -n ricplt
...
Events:
Type     Reason            Age                  From
----     ------            ----                 ----
Warning  FailedScheduling  48s (x18 over 18m)   default-scheduler
Message
----
0/4 nodes are available: 4 pod has unbound immediate
    PersistentVolumeClaims.
```

- E2 Termination Pod is not `READY`: the Pod is `Running` which means that the container is running. The fact that it is not ready means that it is failing some of the initializing procedures. In Subsection 4.3.2 this issue will be analyzed in details.

- Kong Pod is in `ImagePullBackOff`: this Pod's state means that one of its containers could not start because Kubernetes could not pull a suitable container image. With `kubectl describe` it is possible to see the messages from the `kubelet` agent:

```
ubuntu@u18-k8s-master:~/dep$ kubectl describe pod r4-infrastructure-
    kong-599499fbd8-gjvgp -n ricplt
...
Events:
Type     Reason  Age                   From    Message
----     ------  ----                  ----    ----
Warning  Failed  6m47s (x3 over 7m39s) kubelet Failed to pull image
```

```
        "kong-docker-kubernetes-ingress-controller.bintray.io/kong-ingress-
            controller:0.7.0"
        Warning  Failed  6m47s (x3 over 7m39s) kubelet Error: ErrImagePull
        Warning  Failed  6m33s (x5 over 7m38s) kubelet Err: ImagePullBackOff
        Normal   BackOff 2m40s (x21 over 7m38s)kubelet BackOff pulling image
```

Before starting troubleshooting any of the other issues, the first thing was to **resolve the Kong Ingress Controller** one. In fact, as told introducing the ***ric-infra*** components in Section 4.2, the *Kong Ingress Controller* is necessary to communicate via HTTP/REST methods within the cluster.

By searching inside the Kong's Helm chart, it was found that the chart was deprecated:

```
ubuntu@u18-k8s-master:~$ cat dep/ric-dep/helm/infrastructure/subcharts/kong/
    README.md
# DEPRECATED

This chart has been deprecated in favor of
Kong's official chart [repository](https://github.com/kong/charts).

All users are advised to immediately migrate over to the new repository.
...
```

So the subsequent step was to find a way to update the current image within the running Kong deployment.

First of all the new location of the *Kong Ingress Controller* image was found by searching the repository over the web-based collaborative platform *GitHub* - accessible here [39]. Then the choice was to directly modify the running deployment with the command `kubectl edit deploy` and change the entry related to the container image with the name of the new repository.

Here it is possible to see the configuration file before the change:

```
ubuntu@u18-k8s-master:~$ kubectl get deployment r4-infrastructure-kong -n
    ricplt -o yaml
apiVersion: apps/v1
kind: Deployment
...
...
spec:
...
    image: kong-docker-kubernetes-ingress-controller.bintray.io/kong-ingress-
        controller:0.7.0
    imagePullPolicy: IfNotPresent

...
```

Here it is reported the command utilized and the edited file with the new entry:

```
ubuntu@u18-k8s-master:~$ kubectl get deployments -A | grep kong
ricplt        r4-infrastructure-kong

ubuntu@u18-k8s-master:~$ kubectl edit deploy r4-infrastructure-kong -n ricplt
apiVersion: apps/v1
kind: Deployment
...
...
spec:
...
    image: kong/kubernetes-ingress-controller:0.7.0
    imagePullPolicy: IfNotPresent
```

Notice: After this command, the *kube-controller-manager* entity of the Kubernetes control plane is noticed that the configuration file has been edited and automatically restarts the deployment.

After this procedure it was possible to check that the new deployed Pod was running correctly:

```
ubuntu@u18-k8s-master:~/dep/bin$ kubectl get pods -n ricplt | grep kong
NAME                                        READY STATUS   RESTARTS AGE
r4-infrastructure-kong-5b7cdc9dbc-cs5pl  2/2   Running  1        32s
```

Finally, as suggested by the *Installation Guide* in [27], the `curl` command required to check whether the Application Manager container is running correctly was launched:

```
ubuntu@u18-k8s-master:~/dep/bin$ curl -v http://localhost:32080/appmgr/ric/v1/
    health/ready
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 32080 (#0)
> GET /appmgr/ric/v1/health/ready HTTP/1.1
> Host: localhost:32080
> User-Agent: curl/7.58.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 0
< Connection: keep-alive
< Date: Fri, 17 Sep 2021 13:17:46 GMT
< X-Kong-Upstream-Latency: 4
< X-Kong-Proxy-Latency: 1
< Via: kong/1.4.3
<
* Connection #0 to host localhost left intact
```

Notice: as shown in the output the Kong service proxies correctly the request toward the Application Manager container.

At this point the choice was not to get further with the deployment of xApps on the *Cherry release* because of another deployment **issue with the Routing Manager component**. The problem was noticed observing that, time-by-time, the Routing Manager pod was restarting until reaching the `CrashLoopBackOff` state. This particular error state is difficult to troubleshoot since could happen for various reasons and there is not a unique strategy to cope with that.

A common method to start facing this kind of problems is start collecting output logs from containers, in order to catch more specific information.

So the choice was to catch some logs from the Routing Manager's container thanks to `kubectl logs` command:

```
ubuntu@u18-k8s-master:~/dep/bin$ kubectl logs deployment-ricplt-rtmgr-5
    f7545bfc5-vpdr9 -n ricplt | grep -B 6 -A 6 appmgr
{"ts":1631882816860,"crit":"INFO","id":"rtmgr","mdc":{"time":"2021-09-17T12
    :46:56"},"msg":"Invoked httprestful.httpGetXApps: http://service-ricplt-
    appmgr-http:8080/ric/v1/xapps"}
{"ts":1631882821771,"crit":"INFO","id":"rtmgr","mdc":{"time":"2021-09-17T12
    :47:01"},"msg":"restapi: method=GET url=/ric/v1/health/ready"}
{"ts":1631882821771,"crit":"INFO","id":"rtmgr","mdc":{"time":"2021-09-17T12
    :47:01"},"msg":"restapi: method=GET url=/ric/v1/health/alive"}
{"ts":1631882831861,"crit":"WARNING","id":"rtmgr","mdc":{"time":"2021-09-17T12
    :47:11"},"msg":"cannot get xapp data due to: Get http://service-ricplt-
    appmgr-http:8080/ric/v1/xapps: net/http: request canceled (Client.Timeout
    exceeded while awaiting headers)"}
```

Notice: the last `WARNING` message suggests that there could be a problem receiving an HTTP response from the App Manager component.

Given the output obtained by the Routing Manager logs, it was decided to check also the logs of the Application Manager:

```
ubuntu@u18-k8s-master:~/dep/bin$ kubectl logs deployment-ricplt-appmgr-7475457
    bd7-zsxq6 -n ricplt
{"ts":1631882840152,"crit":"INFO","id":"appmgr","mdc":{"time":"2021-09-17T12
    :47:20Z","xm":"0.4.3"},"msg":"Listing deployed xapps failed: Error:
    context deadline exceeded\n"}
{"ts":1631882840152,"crit":"INFO","id":"appmgr","mdc":{"time":"2021-09-17T12
    :47:20Z","xm":"0.4.3"},"msg":"Helm list failed: Error: context deadline
    exceeded\n"}
{"ts":1631882841599,"crit":"ERROR","id":"appmgr","mdc":{"time":"2021-09-17T12
    :47:21Z","xm":"0.4.3"},"msg":"Command failed: exit status 1 - Error:
    context deadline exceeded\n, retrying"}
```

Notice: From the output it is possible to see that there is an execution error.

As told before, the choice was not to get further into troubleshooting this issue since it deviates from the scope of this work. However this kind of issues are frequent approaching the deployment and the integration of different open-source software components and this example was cited to highlight the fact that at the moment there is not a unique path to get through the installation of the O-RAN software without encounter similar obstacles.

4.3.2 Dawn Release

Before starting with the deployment of the *Dawn release* there was the need to uninstall the previous deployment shown in Subsection 4.3.1.

The **it/dep** repository provide also scripts for the disassembling of the deployment and they can be found in the `dep/bin` directory. Similarly to the installation scripts, the uninstalling command requires the same recipe files passed for the installation:

```
ubuntu@u18-k8s-master:~/dep/bin$ nohup sudo ./undeploy-ric-platform ../
    RECIPE_EXAMPLE/PLATFORM/example_recipe_oran_cherry_release.yaml > undeploy
    -ric-platform-cherry-log.out 2>&1 &
```

After the termination of the command above it was possible to check that only `kube-system` pods were left:

```
ubuntu@u18-k8s-master:~$ kubectl get pods -A
NAMESPACE     NAME                                READY   STATUS  RESTARTS   AGE
kube-system   coredns-558bd4d5db-7bvf5            1/1     Running 0          54d
kube-system   coredns-558bd4d5db-fxt2h            1/1     Running 0          54d
kube-system   etcd-u18-k8s-master                 1/1     Running 0          54d
kube-system   kube-apiserver-u18-k8s-master       1/1     Running 0          54d
kube-system   kube-controller-manager-u18-k8s-master 1/1 Running 0          54d
kube-system   kube-proxy-2kgxw                    1/1     Running 0          15d
kube-system   kube-proxy-llrbh                    1/1     Running 0          15d
kube-system   kube-proxy-sm9qx                    1/1     Running 0          15d
kube-system   kube-proxy-txfdk                    1/1     Running 0          54d
kube-system   kube-scheduler-u18-k8s-master       1/1     Running 0          54d
kube-system   weave-net-dsjn4                     2/2     Running 1          15d
kube-system   weave-net-nzpr7                     2/2     Running 1          15d
kube-system   weave-net-sp278                     2/2     Running 0          15d
kube-system   weave-net-t8z5l                     2/2     Running 1          19d
```

Then it was possible to proceed with the deployment of the *Dawn release*:

```
nohup sudo ./deploy-ric-platform ../RECIPE_EXAMPLE/PLATFORM/
    example_recipe_oran_dawn_release.yaml > deploy-ric-platform-dawn-log.out
    2>&1
```

Once the command was terminated, in order to check if the *Kubernetes Ingress Controller* Pod was running the correct image, - see Subsection 4.3.1 for reference - the troubleshooting procedure adopted previously was repeated:

```
ubuntu@u18-k8s-master:~/dep/bin$ kubectl get deployments -A | grep kong
ricplt      r4-infrastructure-kong 0/1     1            0           2m31s

ubuntu@u18-k8s-master:~/dep/bin$ kubectl edit deployment r4-infrastructure-
    kong -n ricplt
deployment.apps/r4-infrastructure-kong edited
```

Notice: this time it was left less time to the containers to initialize before editing the kong deployment, so to ensure that start-up phases were not interrupted, the choice was to exploit recursively the `kubectl rollout restart` command to trigger the re-initialization of the deployed components:

```
ubuntu@u18-k8s-master:~$ kubectl get deployments -n ricplt | awk '{print $1}'
    | xargs -I {name} kubectl rollout restart deployment {name} -n ricplt
Error from server (NotFound): deployments.apps "NAME" not found
deployment.apps/deployment-ricplt-a1mediator restarted
deployment.apps/deployment-ricplt-alarmmanager restarted
deployment.apps/deployment-ricplt-appmgr restarted
deployment.apps/deployment-ricplt-e2mgr restarted
deployment.apps/deployment-ricplt-e2term-alpha restarted
deployment.apps/deployment-ricplt-jaegeradapter restarted
deployment.apps/deployment-ricplt-o1mediator restarted
deployment.apps/deployment-ricplt-rtmgr restarted
deployment.apps/deployment-ricplt-submgr restarted
deployment.apps/deployment-ricplt-vespamgr restarted
deployment.apps/deployment-ricplt-xapp-onboarder restarted
deployment.apps/r4-infrastructure-kong restarted
deployment.apps/r4-infrastructure-prometheus-alertmanager restarted
deployment.apps/r4-infrastructure-prometheus-server restarted
```

After that it was possible to check the pods state:

```
ubuntu@u18-k8s-master:~$ kubectl get pods -n ricplt
NAME                                            READY   STATUS    RESTARTS   AGE
deployment-ricplt-a1mediator-68b5dd6d68-wncvb   1/1     Running   0          2m9s
deployment-ricplt-alarmmanager-bcd465c98-qvq44  1/1     Running   0          2m9s
deployment-ricplt-appmgr-686b64f4fb-92spm       1/1     Running   0          2m9s
deployment-ricplt-e2mgr-577f59dfb7-xrbgs        1/1     Running   0          2m9s
deployment-ricplt-e2term-alpha-85d8964d98-skd9f 0/1     Running   1          2m9s
deployment-ricplt-e2term-alpha-9d6698f67-4skvs  0/1     Running   12         35m
deployment-ricplt-jaegeradapter-699f775f6c-jt5vj 1/1    Running   0          2m9s
deployment-ricplt-o1mediator-85b5b576c-86jvx    1/1     Running   0          2m8s
deployment-ricplt-rtmgr-667ff65bdc-b6vhw        1/1     Running   0          2m8s
deployment-ricplt-submgr-86b4c7bbf7-w75zs       1/1     Running   0          2m8s
deployment-ricplt-vespamgr-78db4c995c-5wl5j     1/1     Running   0          2m8s
deployment-ricplt-xapp-onboarder-cf59d477f-mcg8c 2/2    Running   0          2m8s
r4-infrastructure-kong-7d65fb8c65-w57l5         2/2     Running   1          2m8s
r4-infrastructure-prometheus-alertmanager-bf47c57c6-kbvg9 2/2 Running  0   2m7s
```

```
r4-infrastructure-prometheus-server-fd6f5c4cb-r6mc9 1/1  Running  0       2m7s
ricplt-influxdb-meta-0                              0/1    Pending  0       34m
statefulset-ricplt-dbaas-server-0                   1/1    Running  0       35m
```

Notice: this time, a part from the InfluxDB and the E2 Termination issues, there were not problems with the Routing Manager component.

E2 Termination: As anticipated in Subsection 4.3.1, it was possible to characterize in details the E2 Termination issue:
By checking the `Event` log with the command `kubectl describe pod`, it was possible to understand the reason of the failure:

```
ubuntu@u18-k8s-master:~$ kubectl describe pod deployment-ricplt-e2term-alpha-5
    d58698bc6-7m858 -n ricplt
...
Events:
  Type     Reason     Age    From               Message
  ----     ------     ----   ----               -------
[FAIL] too few messages recevied during timeout window: wanted 1 got 0
  Warning  Unhealthy  2m29s  kubelet  Liveness probe failed: 1631814604269 80/
      RMR [INFO] ric message routing library on SI95 p=43523 mv=3 flg=01 id=a
      (11838bc 4.7.4 built: Apr 27 2021)
[FAIL] too few messages recevied during timeout window: wanted 1 got 0
  Normal   Killing    2m29s  kubelet  Container container-ricplt-e2term failed
       liveness probe, will be restarted
```

Notice: the Liveliness probe of the E2 Termination deployment is failing. Liveliness probe and Readiness probe are automatic healthcheck procedures needed to guarantee the correct behavior of the `Running` container (e.g. they could be used to check if the component communicates correctly with other ones). In particular this check is done by executing a command on the `Running` container and evaluating the result.
In order to mitigate and eventually resolve this issues it was decided to modify specific configurations of such Liveliness and Readiness probes:

```
ubuntu@u18-k8s-master:~$ kubectl get deploy deployment-ricplt-e2term-alpha -n
    ricplt -o yaml
...
livenessProbe:
          exec:
            command:
            - /bin/sh
            - -c
            - /opt/e2/rmr_probe -h 0.0.0.0:38000
          failureThreshold: 3
          initialDelaySeconds: 120
          periodSeconds: 10
          successThreshold: 1
          timeoutSeconds: 60
```

...

Notice: Since increasing timeouts was not sufficient to mitigate the problem, the last choice was to try a second common procedure toward troubleshooting. A common way to derive further information over a container failure is to execute `kubectl exec` command in order to launch a command or a bash terminal over the problematic container. The choice was to try executing the `/opt/e2/rmr_probe` directly on the container in order to check its output:

```
ubuntu@u18-k8s-master:~$ kubectl exec -ti deployment-ricplt-e2term-alpha-
    b6fd4489b-vtjxg -n ricplt -- bash

root@e2term-alpha:/opt/e2# ./rmr_probe --help
usage: ./rmr_probe [-h host:port] [-n msg-count] [-p port | -r] [-t seconds]
    [-v]

root@e2term-alpha:/opt/e2# ./rmr_probe -h 0.0.0.0:38000 -t 10 -v
[INFO] listen port: 43949; sending 1 messages
1631893202000 116/RMR [INFO] ric message routing library on SI95 p=43949 mv=3
    flg=01 id=a (11838bc 4.7.4 built: Apr 27 2021)
[INFO] RMR initialised
[INFO] starting session with 0.0.0.0:38000, starting to send
[INFO] connected to 0.0.0.0:38000, sending 1 pings
[INFO] sending message: health check request prev=0 <eom>
[FAIL] too few messages recevied during timeout window: wanted 1 got 0
```

Notice: given the output derived from the command above, the last choice in order to mitigate the error was to increase up to 30 seconds the timeout and consequently to configure such change in the deployment artifact:

```
root@e2term-alpha:/opt/e2# ./rmr_probe -h 0.0.0.0:38000 -t 30 -v
[INFO] listen port: 43808; sending 1 messages
1631894293734 56/RMR [INFO] ric message routing library on SI95 p=43808 mv=3
    flg=01 id=a (11838bc 4.7.4 built: Apr 27 2021)
[INFO] RMR initialised
[INFO] starting session with 0.0.0.0:38000, starting to send
[INFO] connected to 0.0.0.0:38000, sending 1 pings
[INFO] sending message: health check request prev=0 <eom>
[INFO] got: (OK) state=0
[INFO] good response received; elapsed time = 15727787 mu-sec

ubuntu@u18-k8s-master:~$ kubectl edit deploy deployment-ricplt-e2term-alpha -n
     ricplt
...
livenessProbe:
          exec:
            command:
            - /bin/sh
            - -c
            - /opt/e2/rmr_probe -h 0.0.0.0:38000 -t 30
          failureThreshold: 3
          initialDelaySeconds: 120
```

```
          periodSeconds: 10
          successThreshold: 1
          timeoutSeconds: 60
...
readinessProbe:
          exec:
            command:
            - /bin/sh
            - -c
            - /opt/e2/rmr_probe -h 0.0.0.0:38000 -t 30
          failureThreshold: 3
          initialDelaySeconds: 120
          periodSeconds: 60
          successThreshold: 1
          timeoutSeconds: 60

...
```

Notice: this workaround does not resolve the problem - in fact the Liveliness and Readiness probes still fails sometimes - but this procedural steps aimed to show a possible procedure toward troubleshooting.

Finally, before continuing with xApp deployment, there was the need to check the health of the App Manager container:

```
ubuntu@u18-k8s-master:~$ curl -v http://localhost:32080/appmgr/ric/v1/health/
    ready
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 32080 (#0)
> GET /appmgr/ric/v1/health/ready HTTP/1.1
> Host: localhost:32080
> User-Agent: curl/7.58.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 0
< Connection: keep-alive
< Date: Thu, 16 Sep 2021 16:58:55 GMT
< X-Kong-Upstream-Latency: 6
< X-Kong-Proxy-Latency: 2
< Via: kong/1.4.3
<
* Connection #0 to host localhost left intact
```

4.4 RIC xApp Deployment

Starting from the *Dawn release* deployment, the first choice for the installation of xApps was to follow the *Installation Guide* made available by the OSC in [27] in order to deploy the exemplar *HelloWorld* xApp documented in [40].

The deployment of an xApp consists of two steps: the On-Boarding phase and the actual deployment phase. The On-Boarding phase consists to provide to the xApp On-Boarder component of the RIC Platform cluster an *xApp descriptor* - a `.json` file which defines the behavior of the xApp - and an *optional schema* - a `.json` schema file that validates the self-defined parameters. These files, needed for deploying exemplar xApps, are provided by the OSC. Each testing xApp is released in a different project's repository, for example the Hello World xApp files can be retrieved over the GitHub web based platform under the `ric-app/hw` repository in [41].

Given these premises, the first step was to install the `dms_cli` tool which facilitates the On-Boarding by consuming the xApp descriptor and an additional schema file, producing related xApp Helm charts:

(1) Create a local Helm repository:

```
ubuntu@u18-k8s-master:~$ sudo docker run --rm -u 0 -it -d -p 8090:8080 -e
    DEBUG=1 -e STORAGE=local -e STORAGE_LOCAL_ROOTDIR=/charts -
v $(pwd)/charts:/charts chartmuseum/chartmuseum:latest
Unable to find image 'chartmuseum/chartmuseum:latest' locally
latest: Pulling from chartmuseum/chartmuseum
596ba82af5aa: Pull complete
97cda76ac4f8: Pull complete
7cd1b4b8c77a: Pull complete
Digest: sha256:7
    fb4cd65d68978b1280f39cedc8c4db8c96efe6f622160a109b425a95098615f
Status: Downloaded newer image for chartmuseum/chartmuseum:latest
9c98f979c08b93bbe8b879600f0664b00967301530a68a825809be999fcda945
```

Notice: in the On-Boarding phase, the xApp Helm chart will be loaded into this private Helm repository for which is important to specify a port different from 8080.

(2) Set up the environment variable needed to `dms_cli` tool to connect with the local repository: `export CHART_REPO_URL=http://0.0.0.0:8090`.

(3) Install `dms_cli` tool:

Download `dms_cli` artifacts:

```
ubuntu@u18-k8s-master:~$ git clone "https://gerrit.o-ran-sc.org/r/ric-plt/
    appmgr"
Cloning into 'appmgr'...
remote: Counting objects: 9, done
remote: Total 698 (delta 0), reused 698 (delta 0)
Receiving objects: 100\% (698/698), 1.05 MiB | 1.05 MiB/s, done.
Resolving deltas: 100\% (219/219), done.
```

Change directory to `./appmgr/xapp_orchestrater/dev/xapp_onboarder` and install Python `pip3` package installer:

```
sudo apt install python3-pip
sudo chmod -R 755 /usr/local/lib/python3.6

ubuntu@u18-k8s-master:~/appmgr/xapp_orchestrater/dev/xapp_onboarder$ pip3 --
    version
pip 9.0.1 from /usr/lib/python3/dist-packages (python 3.6)
```

Notice: the Python package installer is needed to launch the `dms_cli` installation script.

To avoid possible problems with conflicting situations launch the following command to uninstall previous instances:

```
pip3 uninstall xapp_onboarder
```

Finally install `dms_cli` with:

```
pip3 install ./
```

At this point it was possible to start with the **On-Boarding procedures** for the actual xApp deployment:

(1) Prepare configuration descriptor and optional schema:
For this step two files have been created in the `/home/ubuntu` directory:

`hwxapp_descriptor_test.json`

`hwxapp_schema_test.json.`

In the former was copied the content of `ric-app/hw/init/config-file.json` file while in the latter was copied the content of `ric-app/hw/init/schema.json` file. The reference files were found available under the `ric-app/hw` repository in [41].

(2) Launch On-Boarding command:

```
ubuntu@u18-k8s-master:~$ dms_cli onboard /home/ubuntu/hwxapp_descriptor_test.
    json /home/ubuntu/hwxapp_schema_test.json
{
    "status": "Created"
}
```

Notice: this command consumes the descriptor in order to generate a Helm chart and other deployment artifacts for the xApp.

(3) Once the xApp is On-Boarded, launch the following command to install it in the `ricxapp` namespace:

```
ubuntu@u18-k8s-master:~$ dms_cli install hwxapp 1.0.0 ricxapp
status: OK

ubuntu@u18-k8s-master:~$ kubectl get pods -n ricxapp
NAME                              READY   STATUS              RESTARTS   AGE
ricxapp-hwxapp-76d4684bb5-r4bc2   0/1     ContainerCreating   0          11s
```

Notice: the command correctly reaches the local repository and exploiting the Hello World xApp's Helm chart instantiate the required Pod instance in the `ricxapp` namespace.

Observation: the procedure shown in this Section exploit a local repository and the `dms_cli` tool to interface with the deployed kubernetes cluster. An xApp deployment method involving the xApp On-Boarder component unfortunately is still not implemented in the *Dawn release*.

4.5 Considerations

At this point it was reached a quite stable installation of the RIC Platform cluster, in which it was also determined a method to deploy xApp microservices. This represents the conclusion of the deployment work presented in this thesis. The next Chapter otherwise will focus on the derivation of some measures related to the deployment of multiple xApps over the obtained cluster.

Chapter 5

Performance Assessment

In this Chapter will be shown the performance assessment performed over the last software component installed. The *HelloWorld* xApp's project provides an xApp able to trigger specific communications toward and within other components of the RIC Platform cluster (like triggering E2 and A1 workflow messages), and for this reason it does not provides a service which could be directly accessed or queried through HTTP/REST methods [40]. Given this premise the choice was to evaluate the performance in the on-boarding and deploying phases achieved by both the `dms_cli` service and the *Kubernetes* control plane. As anticipated in Section 4.4 while deploying the *HelloWorld* xApp, each different xApp is characterized by a descriptor which permits to define the behavior of the xApp's microservice. When passed to the `dms_cli` tool, this last one is in charge to populate the local repository with the Helm chart related to the requested xApp deployment. This procedure is similar to that one operated by the xApp On-Boarder component, but provides a simpler interface to operators and developers for installing their xApps.

The objective of this Chapter is to assess the expected deployment time of multiple xApps over the obtained cluster. The measures concern in practice two situations:

1. the evaluation of the average deployment time given an empty cluster in which either the namespace and other xApps are still not present;

2. the evaluation of a possible change in the expected deployment time given a load of previous installed xApps while installing subsequently multiple xApps over the same cluster.

In order to perform the measurements, the choice was to relay on bash scripting and to create cyclic procedures to reiterate measures multiple times. In particular the choice was to capture the timestamp before and after each execution of the `dms_cli` in order to derive the CPU time needed to execute the command and before and after the check over the deployed Pod state. In the case of `dms_cli` response, it was expected a relatively higher execution time for the `dms_cli install` w.r.t the `dms_cli uninstall` command, since the former one must establish a connection to the local Helm repository and access the related Helm charts before the deployment. In the case of waiting the deployment status of each xApp instead, it was expected to evaluate the average time needed by *Kubernetes* in order to satisfy the deployment request of a particular xApp given the previous cited situations: the first deployment of the xApp in the empty system and the sequential installation of multiple identical xApps over a system in which is already present a load of previous installed xApps.

In the following paragraphs are reported graphically the measures performed. In particular in Figure 5.1 is reported a table showing the average deployment times given an empty cluster, for which a single xApp was installed and uninstalled sequentially 50 times.

	dms_cli install command	Pod in Running state
Average time(ms)	904	3024

	dms_cli uninstall command	Pod Terminated
Average time(ms)	734	36800

Figure 5.1: Average deployment times in an empty cluster

As expected it is possible to see that the `dms_cli install` command take more time for its execution since it needs to interact with the repository. On the other hand the `dms_cli uninstall` take less time since it simply needs to initialize the withdrawal procedure. For both installation and withdrawal phases then, the commands executions take less than 1 second while the time needed to complete the installation procedure is quite different from the time needed for the Pod termination. In the installation phase in fact the times are comparable with the time needed for the creation and the start up of containers, while the termination procedures of Pods relies on a series of checks and timeouts managed by Kubernetes, which considerably slows the procedure down.

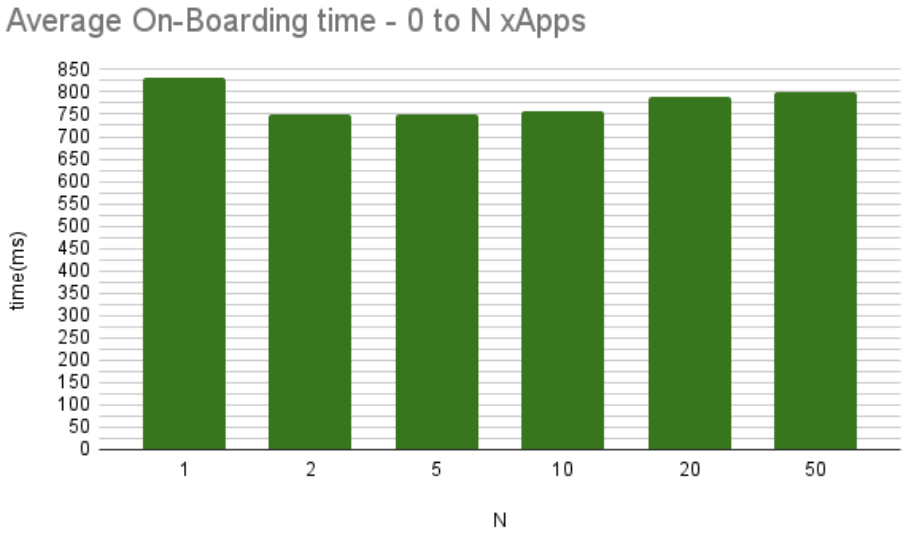


Figure 5.2: Average on-boarding times given an increasing load

Regarding the second case, in which is intended to assess a possible deployment time offset while installing subsequently multiple xApp replica over the same cluster, it was evaluated first the average time obtained while on-boarding recursively the needed Helm charts over the local repository - as shown in Figure 5.2. Then with a cyclic installation of the 50 on-boarded xApps it was possible to show the instantaneous variation of the installation time given the increasing load in terms of previous installed xApps as it is reported in Figure 5.3. Finally in Figure 5.4 is reported the average installation time given the increasing load.

As it is possible to see from the figures the temporal values are not varying in a systematic way and it is not possible to assess a meaningful correlation among the number of xApps and the deployment time. Moreover the time obtained for small N indexes is reported only for reference just to show the fact that the values are comparable with the time values observed with greater N indexes. For example, in Figure 5.2, the fact that the measure related to a single on-boarding results to be greater than the average time of 50 on-boarding reiterations is not indicative of a systematic behavior and it is due to the specific execution of the command.

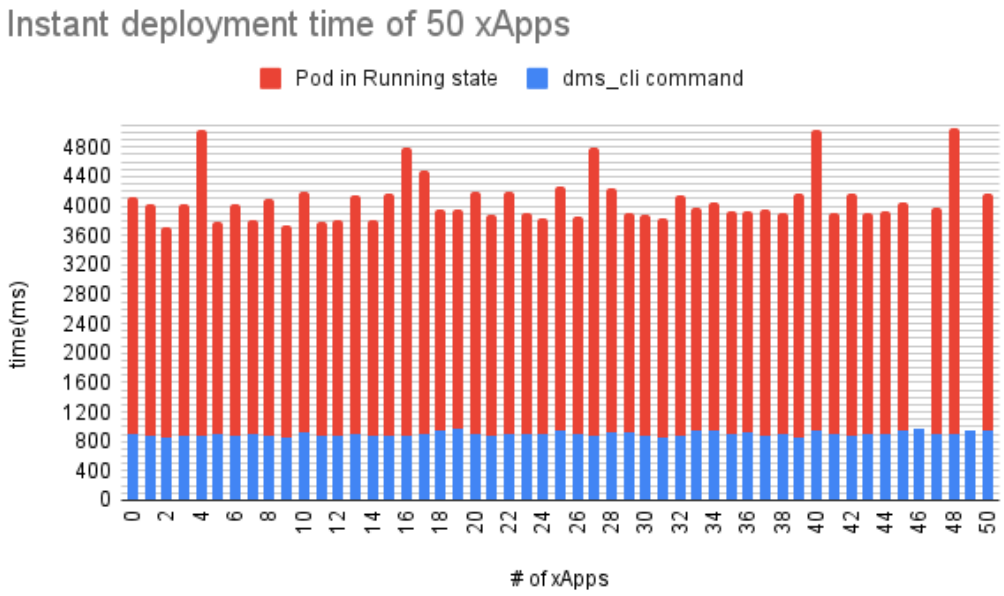


Figure 5.3: Instantaneous installation times given an increasing load

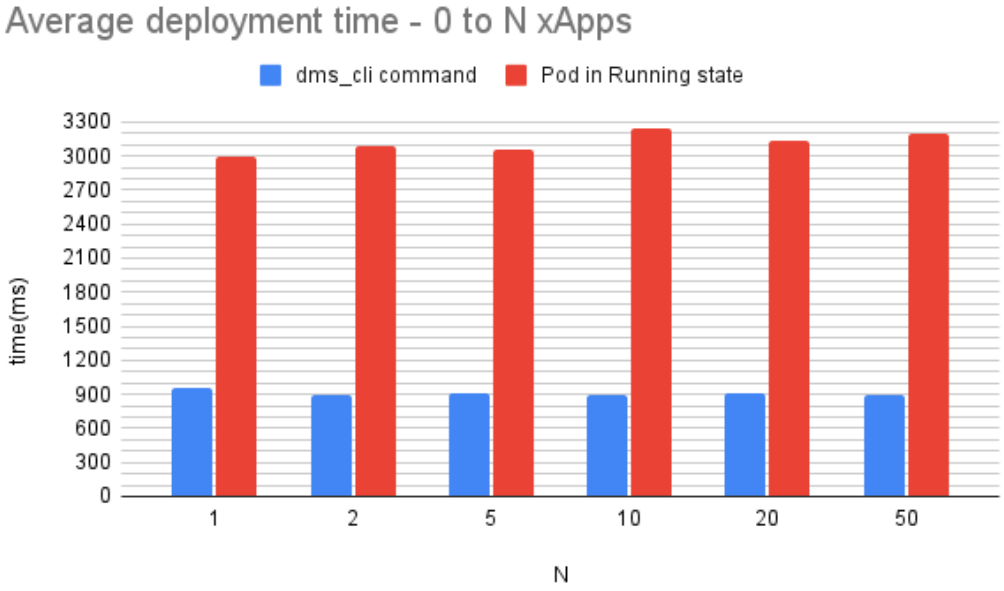


Figure 5.4: Average installation times given an increasing load

Notice: the missing values in Figure 5.3 are irrelevant outliers since they reveal to be excessively out of the range of possible values due to reasons which are independent from the platform examined.

Finally, as shown in Figure 5.3 and Figure 5.4, considering up to 50 reiterations there is not a meaningful correlation among the given load - in terms of installed xApps - and the deployment time measured.

These measurements concluded the work of performance assessment which shows the fact that the deployment time of the HelloWorld xApp's microservice is independent of the number of queries in this specific case. However, it was possible to notice and report how *Kubernetes* react to the increasing load in terms of deployed services, and here is reported the case of a specific xApp re-scheduling on another worker node due to lack of node's resources:

```
ubuntu@u18-k8s-master:~$ kubectl get pods -n ricxapp -o wide
...
ricxapp-hwxapp37-d8f6c74fc-54qnl    1/1     Running   0   ... u18-k8s-worker03
ricxapp-hwxapp37-d8f6c74fc-6tckp    0/1     Evicted   0   ... u18-k8s-worker01
ricxapp-hwxapp37-d8f6c74fc-82qxm    0/1     Evicted   0   ... u18-k8s-worker01
ricxapp-hwxapp37-d8f6c74fc-d27wc    0/1     Evicted   0   ... u18-k8s-worker01
ricxapp-hwxapp37-d8f6c74fc-pf42d    0/1     Evicted   0   ... u18-k8s-worker01
ricxapp-hwxapp37-d8f6c74fc-zr8zg    0/1     Error     1   ... u18-k8s-worker01
...

ubuntu@u18-k8s-master:~$ kubectl describe pod ricxapp-hwxapp37-d8f6c74fc-zr8zg
     -n ricxapp
.......
Events:
  Type     Reason           Age                    From               Message
  ----     ------           ----                   ----               -------
  Normal   Scheduled        6m16s                  default-scheduler
      Successfully assigned to u18-k8s-worker01
  Warning  Evicted          6m15s                  kubelet
      The node was low on resource: ephemeral-storage.
  Normal   SandboxChanged   6m13s                  kubelet
      Pod sandbox changed, it will be killed and re-created.
  Normal   Pulled           6m12s (x2 over 6m15s)  kubelet
      Container image already present on machine
  Normal   Created          6m12s (x2 over 6m15s)  kubelet
      Created container hwxapp
  Normal   Started          6m11s (x2 over 6m15s)  kubelet
      Started container hwxapp
  Normal   Killing          6m11s (x2 over 6m14s)  kubelet
      Stopping container hwxapp
```

Chapter 6

Conclusions

This thesis started by presenting the efforts of the O-RAN Alliance group, which is operating in the context of software based solutions for open and programmable Radio Access Networks. The O-RAN Alliance group is working over a reference design envisioning to provide operators and service providers with a common methodology while approaching the deployment of disaggregated and softwarized 5G RANs. At the same time the combined effort of O-RAN with the Linux's Foundation brought to the production of an early stage of software releases and documentations, delivered as an exemplar platform for the development of further RAN software.

The objective of this thesis was to assess the feasibility of the deployment of the O-RAN software components over a virtualized infrastructure, investigating the capability of the available software at current state. Moreover, this work aimed to underline the critical issues which could be encountered while addressing the integration of different software components and to evaluate the critical role of *Kubernetes* in the orchestration of containerized applications. Given the objectives of this work, the adopted approach converged to practical aspects and in particular it was able to show the deployment of specific O-RAN software components over a customized virtual environment.

The adopted deployment choice was to setup the virtual infrastructure and the virtual infrastructure management framework - represented by *Kubernetes* - in a modular fashion such that it was possible to keep track of each step of the installation. Leveraging this choice then, it was possible to select the most recent release of each infrastructural software component highlighting the compatibility among them. Finally, approaching the O-RAN software

installation it was possible to underline some critical issues encountered in the deployment process. For example it was possible to show the fact that some components present in the provided repository were deprecated and that, referring to specific branches of the software release, the documentation is still incomplete.

Despite the technical obstacles mentioned above, in this thesis the deployment of an exemplar xApp was successfully achieved, and it was possible to perform some basic evaluations on performance. The performance metrics assessed refer to the evaluation of the time required by an xApp to be deployed over the cluster in two different situations: the average deployment time given an empty cluster and the average time given an increasing load of previous installed xApps.

To conclude this work of thesis it is possible to say that the procedural steps adopted can be considered as a baseline for further investigations over the O-RAN software deployment feasibility, also depending on the evolution of the available software components.

Bibliography

[1] Ibrahim Afolabi, Tarik Taleb, Konstantinos Samdanis, Adlen Ksentini, and Hannu Flinck. "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions". In: *IEEE Communications Surveys & Tutorials* 20.3 (2018), pp. 2429–2453.

[2] *5G; System architecture for the 5G System (5GS) (3GPP TS 23.501 version 16.6.0 Release 16)).* URL: https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf.

[3] Leonardo Bonati, Michele Polese, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. "Open, programmable, and virtualized 5G networks: State-of-the-art and the road ahead". In: *Computer Networks* 182 (2020), p. 107516.

[4] *Open RAN.* (Accessed on 06/30/2021). 2021. URL: https://www.3gpp.org/news-events/2150-open_ran.

[5] Deutsche Telekom AG, Orange S.A., Telefónica S.A., and Vodafone Group Plc. "Memorandum of Understanding on the Implementation of Open RAN Based Networks in Europe". 2021. URL: https://www.politico.eu/wp-content/uploads/2021/01/20/POLITICO-Memorandum-of-Understanding-OPEN-RAN-big-four-operators-January-2021.pdf.

[6] *About O-RAN ALLIANCE.* (Accessed on 06/30/2021). URL: https://www.o-ran.org/about.

[7] *O-RAN Software Community.* (Accessed on 26/07/2021). 2021. URL: https://o-ran-sc.org/.

[8] *kubernetes.* (Accessed on 26/07/2021). 2021. URL: https://kubernetes.io/.

[9] *O-RAN: Towards an Open and Smart RAN*. 2018. URL: https://static1.squarespace.com/static/5ad774cce74940d7115044b0/t/5bc79b371905f4197055e8c6/1539808057078/O-RAN+WP+FInal+181017.pdf.

[10] *O-RAN Use Cases and Deployment Scenarios*. 2020. URL: https://static1.squarespace.com/static/5ad774cce74940d7115044b0/t/5e95a0a306c6ab2d1cbca4d3/1586864301196/O-RAN+Use+Cases+and+Deployment+Scenarios+Whitepaper+February+2020.pdf.

[11] *NG RAN: Architecture Description*. TS 38.401. V16.5.0. 3GPP. Apr. 2021. URL: https://www.3gpp.org/ftp/Specs/archive/38_series/38.401/.

[12] RIMEDO Labs. "Introduction to O-RAN: Concept and Entities". (Accessed on 06/30/2021). Mar. 2021. URL: https://www.rimedolabs.com/blog/introduction-to-o-ran/.

[13] *O-RAN Architecture Description*. v04.00. O-RAN Alliance. Mar. 2021. URL: https://www.o-ran.org/specification-access.

[14] *Control, User and Synchronization Plane Specification*. v06.00. O-RAN Alliance. Mar. 2021. URL: https://www.o-ran.org/specification-access.

[15] *Cloud Architecture and Deployment Scenarios for O-RAN Virtualized RAN*. v02.01. O-RAN Alliance. July 2020. URL: https://www.o-ran.org/specification-access.

[16] *NG-RAN: F1 Application Protocol (F1AP)*. TS 38.473. V16.5.0. 3GPP. Apr. 2021. URL: https://www.3gpp.org/ftp/Specs/archive/38_series/38.473/.

[17] *Near-Real-time RAN Intelligent Controller Architecture and E2 General Aspects and Principles*. v01.01. O-RAN Alliance. July 2020. URL: https://www.o-ran.org/specification-access.

[18] *Use Cases Detailed Specification*. v05.00. O-RAN Alliance. Mar. 2021. URL: https://www.o-ran.org/specification-access.

[19] *Non-RT RIC: Functional Architecture*. v01.01. O-RAN Alliance. Mar. 2021. URL: https://www.o-ran.org/specification-access.

[20] *O-RAN Operations and Maintenance Architecture*. v04.00. O-RAN Alliance. 2021. URL: https://www.o-ran.org/specification-access.

[21] *The "Cherry" Release of O-RAN Open Software Moves the O-RAN Ecosystem Closer to Deployment in Mobile Networks Around the Globe*. (Accessed on 26/07/2021). 2021. URL: www.o-ran.org/blog/2021/1/7/the-cherry-release-of-o-ran-open-software-moves-the-o-ran-ecosystem-closer-to-deployment-in-mobile-networks-around-the-globe.

[22] *About ONAP*. (Accessed on 26/07/2021). 2018. URL: https://www.onap.org/about.

[23] *Cherry Release (Dec 2020) - Confluence*. (Accessed on 20/09/2021). 2021. URL: https://wiki.o-ran-sc.org/collector/pages.action?key=REL.

[24] *Getting Started*. (Accessed on 09/20/2021).

[25] *O-RAN SC Projects*. 2021. URL: https://docs.o-ran-sc.org/en/latest/projects.html.

[26] *Helm*. (Accessed on 09/08/2021). URL: https://helm.sh/.

[27] *Installation Guides — it-dep master documentation*. (Accessed on 07/27/2021). 2021. URL: https://docs.o-ran-sc.org/projects/o-ran-sc-it-dep/en/latest/installation-guides.html.

[28] *Deploying Kubernetes Cluster with kubeadm*. (Accessed on 07/28/2021). Aug. 2018. URL: https://wiki.onap.org/display/DW/Deploying+Kubernetes+Cluster+with+kubeadm.

[29] Martin Skorupski. *O-RAN integration into ONAP*. (Accessed on 07/29/2021). Aug. 2019. URL: https://wiki.onap.org/display/DW/O-RAN+integration+into+ONAP.

[30] KVM. *Main Page — KVM*, (Accessed on 27/07/2021). 2016. URL: https://www.linux-kvm.org/index.php?title=Main_Page&oldid=173792.

[31] *Install Docker Engine on Ubuntu*. (Accessed on 08/02/2021). 2021. URL: https://docs.docker.com/engine/install/ubuntu/.

[32] *Installing kubeadm*. (Accessed on 08/13/2021). 2021. URL: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/.

[33] *Glossary — Kubernetes*. (Accessed on 09/03/2021). URL: https://kubernetes.io/docs/reference/glossary/?all=true.

[34] *Helm — Installing Helm.* (Accessed on 09/07/2021). URL: https://helm.sh/docs/intro/install/.

[35] *Changes Since Helm 2.* (Accessed on 09/07/2021). URL: https://helm.sh/docs/faq/changes_since_helm2/.

[36] *Integration and Testing — it-dep master documentation.* (Accessed on 09/11/2021). URL: https://docs.o-ran-sc.org/projects/o-ran-sc-it-dep/en/latest/index.html.

[37] *Developer Guides — it-dep master documentation.* (Accessed on 09/20/2021). URL: https://docs.o-ran-sc.org/projects/o-ran-sc-it-dep/en/latest/developer-guides.html#developer-guides.

[38] *Kubernetes Monitoring with Prometheus.* (Accessed on 09/21/2021). URL: https://sysdig.com/blog/kubernetes-monitoring-prometheus/.

[39] *GitHub - Kong/kubernetes-ingress-controller: Kong for Kubernetes: the official Ingress Controller for Kubernetes.* (Accessed on 09/22/2021). URL: https://github.com/Kong/kubernetes-ingress-controller.

[40] *HelloWorld xAPP Documentation.* (Accessed on 09/28/2021). URL: https://docs.o-ran-sc.org/projects/o-ran-sc-ric-app-hw/en/latest/index.html.

[41] *GitHub - o-ran-sc/ric-app-hw: Mirror of the ric-app/hw repo.* (Accessed on 09/23/2021). URL: https://github.com/o-ran-sc/ric-app-hw.

Appendix A

Appendix: Host VMs setup

In this Appendix is described step by step the procedure adopted to instantiate and obtain remote access to some Ubuntu 18.04 VMs managed by KVM on a remote HP server hosted by the University department as anticipated in Chapter 3.
In Figure A.1 is reported again the final topology for reference.

A.1 Network nodes and NAT port forwarding

To get remote access to the VM deployment there was the need of a two-hop NAT port forwarding for each VM:

- the inner one permits to redirect the connection from the private LAN 192.168.10.0/24 connecting the two servers to the specific IP address of each VM;
- the outer one permits to connect directly from the university public IP address, by redirecting connections for specific TCP ports towards the internal IP address:port pair of the HP server.

Here the forwarding rules applied for each connection:
Inner NAT port forwarding:

```
root@server-hp-netlab:~# iptables -t nat -A PREROUTING -d 192.168.10.203 -p
    tcp --dport <internal port> -j DNAT --to-dest 192.168.122.50:22

root@server-hp-netlab:~# iptables -I FORWARD 1 -d 192.168.122.50 -p tcp --
    dport 22 -j ACCEPT
```

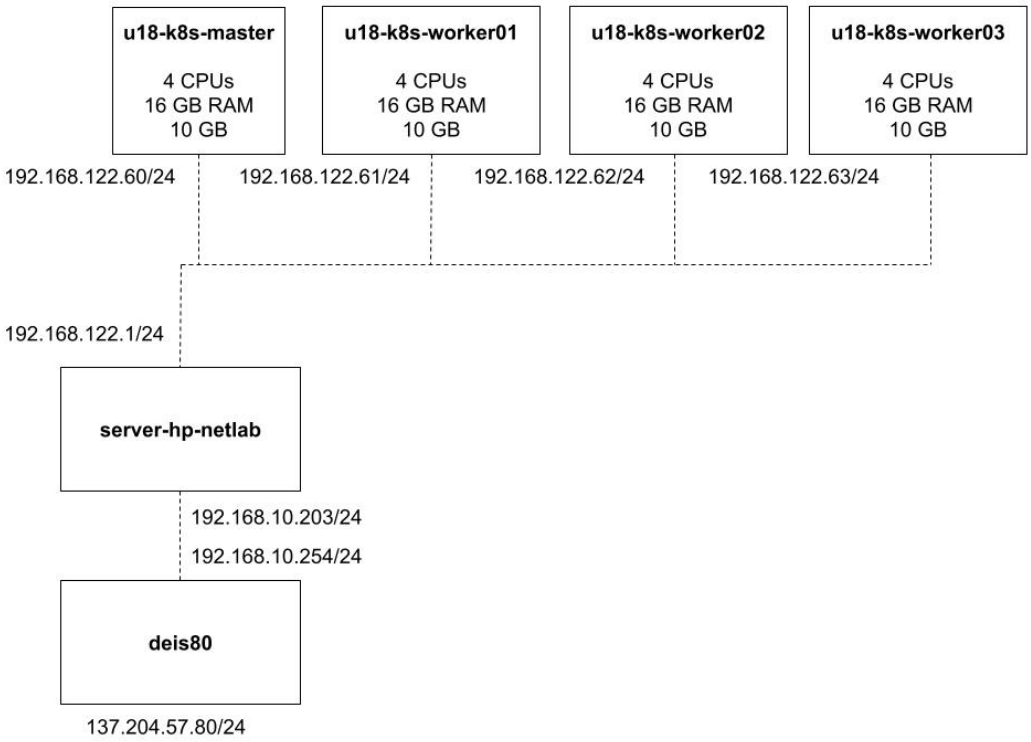


Figure A.1: Virtual Infrastructure Reference

Outer NAT port forwarding:

```
[root@deis80 ~]# iptables -t nat -A PREROUTING -i br-eth5 -d 137.204.57.80 -p
    tcp --dport <external port> -j DNAT --to-dest 192.168.10.203:<internal
    port>

[root@deis80 ~]# iptables -A FORWARD -d 192.168.10.203 -p tcp --dport <
    internal port> -m state --state NEW -j ACCEPT
```

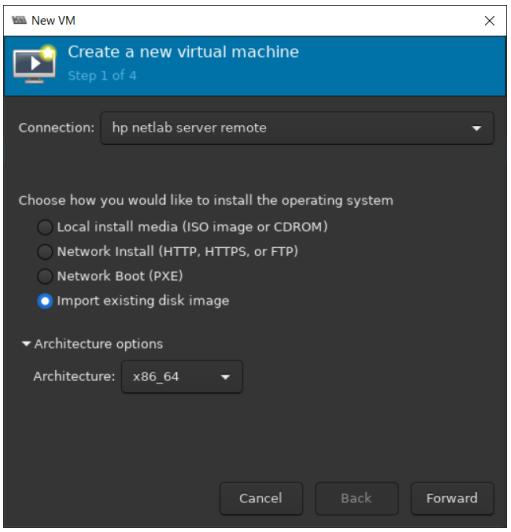
A.2 VMM: VM creation on remote server

In order to deploy four identical VMs based on Ubuntu 18.04 distribution over the remote HP server it was created a basic image of the OS in order to replicate it multiple times.

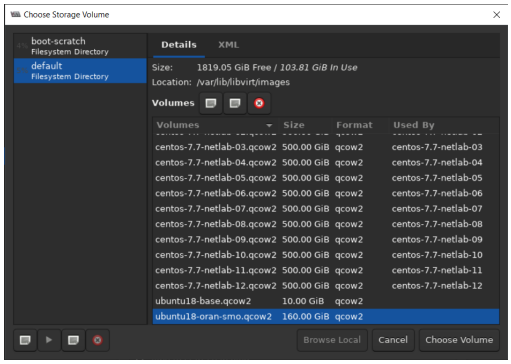
In Figure A.2 are reported the steps to create a VM image with different characteristic with respect to that one exploited in the deployment of this work, but in general the steps are similar and they were done starting from an Ubuntu 18.04 disk image with 4 CPUs and 10 GB of HDD and 16 GB of RAM instead of doing them from the disk image shown in Figure A.2 in which are considered 8 CPUs, 160 GB of HDD and 32 GB of RAM.

Once having created a base VM called *"ubuntu1804-base"*, then was suffiecient to clone such VM and rename each new one with the related name and role as shown in Figure A.3.

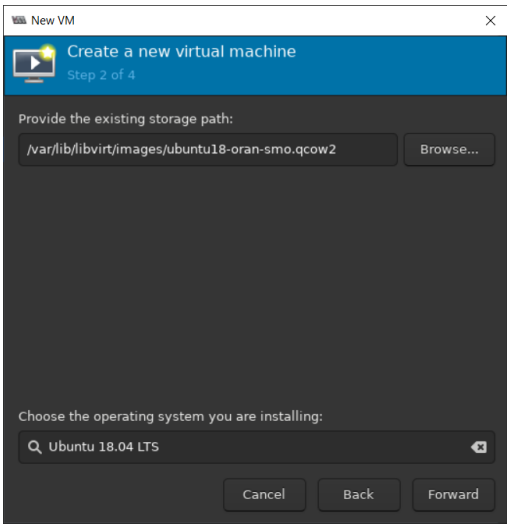
Finally the choice was not to configure an isolated IP network among the new created VMs since was considered sufficient the default private network interconnecting the VMs on the remote HP server.



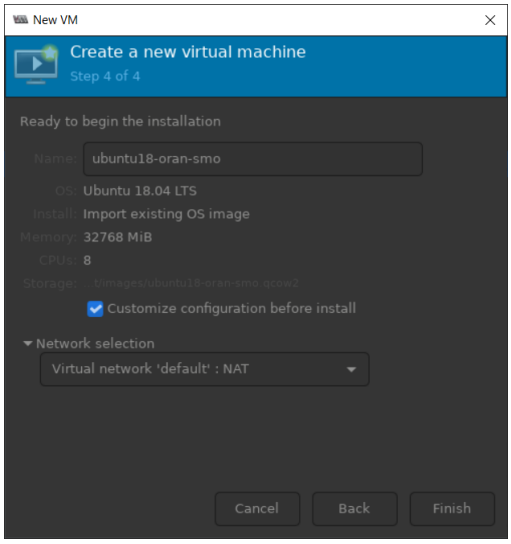
(a) Step 1



(b) Step 2

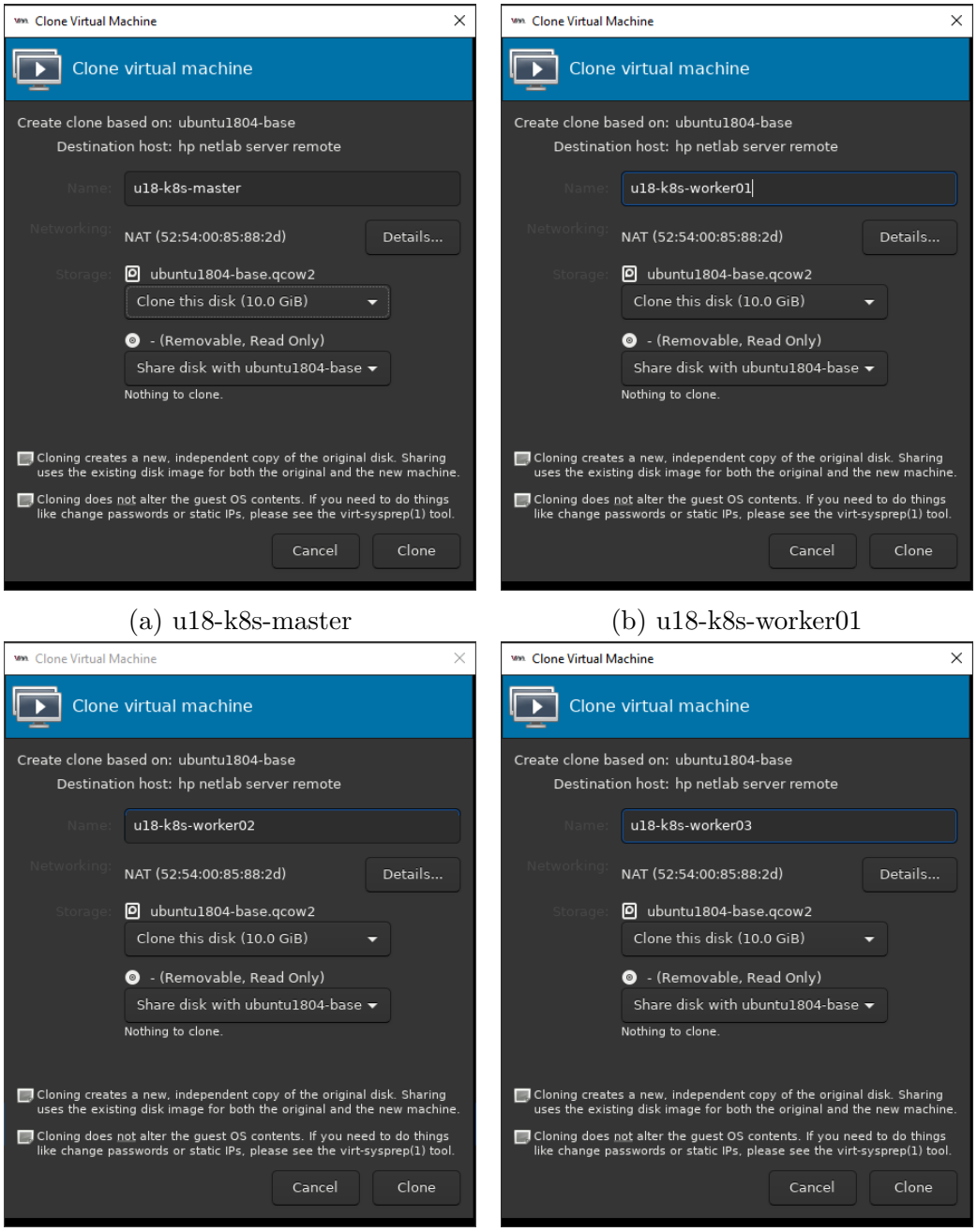


(c) Step 3



(d) Step 4

Figure A.2: VMM procedural steps



(a) u18-k8s-master (b) u18-k8s-worker01

(c) u18-k8s-worker02 (d) u18-k8s-worker03

Figure A.3: VM clones

List of Figures