UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

EDERSON RIBAS MACHADO

A CONTAINER-BASED ARCHITECTURE TO PROVIDE SERVICES FROM SDR DEVICES

Porto Alegre
2021

EDERSON RIBAS MACHADO

A CONTAINER-BASED ARCHITECTURE TO PROVIDE SERVICES FROM SDR DEVICES

Thesis presented to Programa de Pós-Graduação em Engenharia Elétrica of Universidade Federal do Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Electrical Engineering.
Area: Control and Automation

ADVISOR: Prof. Dr. Ivan Muller

Porto Alegre
2021

EDERSON RIBAS MACHADO

A CONTAINER-BASED ARCHITECTURE TO PROVIDE SERVICES FROM SDR DEVICES

This thesis was considered adequate for obtaining the degree of Master in Electrical Engineering and approved in its final form by the Advisor and the Examination Committee.

Advisor: ____________________________________
Prof. Dr. Ivan Muller, UFRGS
Doutor pela Universidade Federal do Rio Grande do Sul – Porto Alegre, Brasil

Porto Alegre, June 2021.

DEDICATÓRIA

Ao meu recém nascido filho Zion e a minha esposa Luana, os pilares e a luz da minha vida.

“A única luta que se perde é aquela que se abandona”
– CARLOS MARIGUELLA

ACKNOWLEDGMENTS

Firstly, I would like to thank my parents Cida and Antônio for always giving me affection and inspiration in my student life. I'm grateful to my sisters Cleonice and Eliane for always supporting me on my daily journey and advising me in my new life cycle as a father. I would like to thank my student's house colleagues Fabrício, Enoque, Felipe and Tiago, to give me support and attention and for listening to me during these days.

I'm honored to have had Prof. Dr. Ivan Müller as adviser, who always supported me through the problems enhancing the motivations for the best. I also thank the colleagues Filipe Guterres and Rodolfo from the LACOM laboratory whom went through the same challenges during this journey.

A special feeling of gratitude to my loving partner Luana, for the wonderful moments with our son, for her affection and love which have inspired me throughout the process. I will always remember fondly her support in all that we went through during these harsh times of COVID-19.

I also thank the examination board for accepting the invitation to evaluate this thesis, the PPGEE committee for the patience, and CAPES to the financial assistance provided to my research.

ABSTRACT

Software Defined Radio is a programmable radio device that, when connected to a computer or as an embedded solution, can transmit and receive data information using radio waves. The programming features of the SDR and its RF bandwidth range extends the application possibility to several areas, including aviation, satellite, radar, and mobile communication. SDR has drawn great attention to network service provision. Acting as a multi-programmable air interface at the edge of wired network environments, SDR can receive, decode and forward radio information, which is used to generate the services. Examples of services including real-time flight tracker web pages, and sensor monitoring data charts. However, to provide network services, SDR must integrate into complex network environments where recent technologies, such as NFV, SDN, containerization and cloud computing, are applied. This thesis addresses the integration of SDRs with containerization. It proposes an easy-to-deploy container-based architecture to provide network services from SDR devices. Using different types of SDR devices (USRP, LimeSDR and RTL-SDR), *GNURadio* platform and *Docker Container*, two use cases of the proposed architecture are presented, demonstrating scenarios where ADS-B and LoRa communication are implemented in order to provide services to end-users. Evaluation of the proposed solution is performed comparing two models of service provision: with the proposed architecture (two levels of network isolation), and without the architecture. The overhead time added to launch the services, the time response and computational resource utilization are compared, showing that there is an overhead added by the architecture which impacts on the system performance. The overhead increases with the applied network isolation level. Conversely, the architecture converts the service functional components into modular components, its application can be extended to different RF projects and SDR types, and offers non-functional benefits such as, real-time capability, network isolation, fine setting of communication parameters, and a set of control and configuration features inherited from container virtualization platform.

Keywords: Software Defined Radio, Containerization, Virtualization, Automatic Dependent Surveillance Broadcast.

RESUMO

Rádio Definido por Software (SDR) é um dispositivo de rádio programável que, conectado a um computador ou como uma solução embarcada, pode transmitir e receber informações usando ondas de rádio. A característica de programabilidade do SDR e sua largura de banda de rádio frequência (RF) estendem sua aplicação a diversas áreas que incluem aviação, satélite, radar e dispositivos móveis. O emprego do SDR tem despertado grande interesse na provisão de serviços de rede. Atuando como uma interface sem-fio multiprogramável na borda de redes cabeadas, o SDR é capaz de transmitir, receber e decodificar informações de rádio. Estas informações são usadas para fornecer serviços, como por exemplo uma página de internet contendo um mapa de rastreamento de aeronaves em tempo real, e gráficos de monitoramento de sensores. No entanto, para ser usado para esta finalidade, o SDR deve integrar-se às correntes tecnologias dos ambientes de rede, como NFV, SDN, containerização, e a computação em nuvem. Esta dissertação está focada na integração do SDR com a technologia de containerização. É proposta uma arquitetura para geração de serviços usando contâineres e o SDR como dispositivo de borda. Usando diferentes modelos de SDRs (USRP, LimeSDR e RTL-SDR), a plataforma *GNURadio* e *Docker containers*, dois cenários de aplicação da arquitetura são apresentados, nos quais a comunicação ADS-B e LoRa são implementadas. A avaliação da solução proposta é realizada comparando-se a geração de serviço com a arquitetura, (com dois níveis de isolação de rede), e sem a arquitetura. O tempo de lançamento e de resposta dos serviços, e a utilização dos recursos computacionais são comparados, mostrando que a arquitetura tem impacto nesses fatores. Este impacto aumenta conforme o nível de isolação de rede utilizado. Por outro lado a arquitetura aplica uma topologia que converte os componentes funcionais do serviço em blocos modulares, tornando possível sua aplicação em diferentes projetos de RF, e oferece benefícios não funcionais, como a capacidade de prover serviços em tempo real, emprego com diferentes modelos de SDR, e isolação de rede. Além disso, a arquitetura adiciona uma série de características de controle herdadas da tecnologia de virtualização.

Palavras-chave: Radio Definido por Software, Containerização, Virtualização, Vigilância Dependente Automática por Radiodifusão.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ADS-B	Automatic Dependent Surveillance-Broadcast
ADC	Analog-Digital Converter
ANOVA	Analysis of Variance
API	Application Programming Interface
CI	Confidence Interval
CLI	Command Line Interface
COTS	Commercial Off-The-Shelf
CPU	Computer Unit
CR	Cognitive Radio
DAC	Digital-Analog Converter
DVT-B	Digital Video Broadcasting
DSP	Digital Signal Processing
FM	Frequency Modulation
FPGA	Field-Programmable Gate Array
GPP	General Purpose Processor
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
LNA	Low-Noise Amplifier
LPWA	Low Power Wide Area
LPWAN	Low Power Wide Area Network
LoRa	Long Range
LoraWAN	Long Range Wide-Area Network
MIMO	Multiple-Input and Multiple-Output
NFV	Network Function Virtualization

OOT	Out-of-tree
RAM	Random-Acess Memory
RF	Radio Frequency
SDN	Software Defined Networks
SDR	Software Defined Radio
SDS	Software Defined Systems
UHD	USRP Hardware Drive
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
VM	Virtual Machine

CONTENTS

1 INTRODUCTION

Internet connectivity and service provision are growing fast. It is estimated that 66% of the global population will have Internet access in 2023 and the number of devices connected to Internet Protocol IP networks will be more than three times the global population, with the result of nearly 3.6 devices per capita (CISCO, 2020). To meet the ever-increase of network utilization and service provisioning in this scenario, where services are reflected in the form of real-time applications, data streaming and storage, a lot of solutions have been implemented using Software Defined Systems (SDS) enabling technologies. These technologies include NFV, SDN, Containerization, Cloud/Edge computing, and Cognitive Radio (ZENGLIN *et al.*, 2020). Following the hardware to software trend, these solutions have brought a variety of options to *DevOps*[1] practices, creating flexibility but also enhancing dynamism and complexity of networks environments.

In these network environments, the applicability of Software Defined Radio (SDR) in edge nodes takes great attention to provide Radio Frequency (RF) services as a multi-programmable air-interface. Acting as an edge device, SDR can transmit and receive radio signals, being responsible to convert the signal from RF to digital domain and vice-versa. Once an RF signal is received and decoded at SDR, digital decoded frames could be forwarded into the network in order to provide services. In this scenario, a service can be defined as a software functionality, or a set of software functionalities, which are used by clients for different purposes, according to (SOA-RAF, 2012). Examples of end-to-end services provided from SDR are internet pages containing real-time flight tracker, and sensor monitoring charts.

In the wireless landscape, SDR devices have gained considerable momentum. Due to reconfiguration capabilities, SDR enables desired flexibility to deploy varied radio communications projects, allowing DSP programming, faster prototype and deployment of RF applications. The success of these devices is leveraged by its capacity to lead achieved popularity, thanks to low-cost COTS devices such as RTL2832U (RTL-SDR), combined

[1]DevOps in this context can be interpreted as a set of practices intended to reduce the time between committing a change to a system and applying this change into normal production, while ensuring high quality (ERICH; AMRIT; DANEVA, 2017).

with its component performance evolution and implementation facilities, allowing ease deployment of Digital Signal Processing (DSP) projects in the communication domain. With the remarkable programming features and their popularity, the SDR devices have obtained great insertion in the industry, academia (prototyping and educational purposes), and open source communities (e.g., GNURadio and GitHub). As a result, SDR emerges from a variety of open-source RF applications in a wide range of areas that includes radar, aviation, satellite, and mobile communication, besides emergent technologies such IoT, 5G and electric car, as well as Low-power Wide-area Networks (LPWAN) segment.

Despite the considerable momentum of SDR, integration to such complex and dynamic wired networks environments to provide services as an edge device, can result in a difficult task; because it requires developer knowledge of radio communication protocols and DSP to develop the data reception script, as well as base knowledge of SDS network technologies involved to integrate them (MARTINS *et al.*, 2020). Furthermore, SDR integration architectures to real-time service provision, for the most part, still remains not virtualized and service-oriented, which do not explore all capabilities of SDR, such as the flexibility to provide a wider variety of services.

To address the aforementioned issues, in this thesis, an easy-to-deploy architecture to provide RF services based on container virtualization is proposed. Using a Universal Software Radio Peripheral (USRP) 2932 as an SDR device, open-source GNURadio platform to DSP programming, and Docker Container as virtualization solution, it has been presented a proof-of-concept of the model demonstrating a scenario where Automatic Dependent Surveillance Broadcast ADS-B, is implemented to provide real-time flight and altitude track services. A second scenario explores Long Range (LoRa) modulation technique to provide a wireless file transfer system as a service.

The proposed architecture evaluation performance is measured in the ADS-B service provision case, in terms of resource utilization (under increasingly user requests condition), startup time overhead and response time of the solution when compared to the service generated directly in the host machine, (i.e., without the containers). The results have shown that the architecture brings non-functional benefits, such as ease of control, scalability, SDR portability and network isolation, but with a cost in system performance. A maximum overhead of 5.1% in CPU utilization was observed, $8.32s$ to start the services, and $50.96ms$ for maximum request waiting time at the $99th$ percentile.

Being limited to the local scope, this work aims, through SDR function containerization, to provide a basis for more complex implementations involving the integration of SDR with emerging technologies in SDS. Therefore, this thesis includes a discussion about the applicability of the solution in network environments along with technologies such as NFV, SDN and distributed systems.

This text is organized as follows. Chapter 2 introduces the main background concepts involved. In Chapter 3, related work is cited and discussed. In Chapter 4 the proposed architecture is presented, and in Chapter 5 its stages of development are specified. Chapter 6 shows some features brought by the architecture implementation, and Chapter 7 shows the comparative results and discussion. Finally, Chapter 8 presents the conclusions and further improvements of this thesis.

2 BACKGROUND

In this chapter, theoretical foundations and technical review of devices are presented. Firstly, Section 2.1 introduces the SDR devices, showing SDR main features, device types, and programming platform. Then, a brief conception of end-to-end service is shown with some illustrated examples (Section 2.2). Next, in Section 2.3, the key virtualization concept has been introduced. Finally, in Section 2.4, ADS-B communication protocol and LoRa modulation technique used in testing scenarios were addressed.

2.1 Software Defined Radio

In a Software Defined Radio the main components, such as filters, mixers, detectors, modulators, demodulators, traditionally implemented in hardware are instead implemented by means of software.

A transceiver could be called a Software Radio (SR) if its communication functions are implemented as programs running on a processor (SMITH, 1995). If an SR can sample its received signal, it can be considered an SDR. However, SDR concept is not new, in the early 1980s a group of researchers at E-Systems Inc. have coined the term "software radio" for the first time, with regard to a radio receiver which applies thousands of adaptive filter taps, leading to interference cancellation and demodulation programming on broadband signals. Afterwards, widely and notable SDR research was conducted by Joseph Mitola III, with significant contributions to SDR implementation, including a deep software defined radio architecture illustration and analysis in mathematical perspective (MITOLA, 1999).

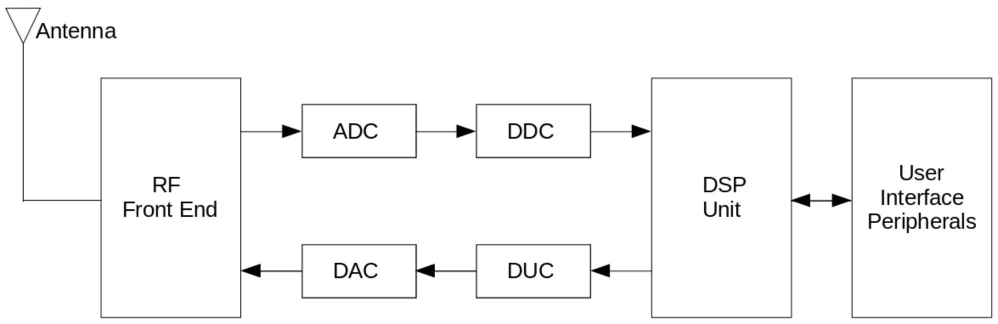
Nowadays, SDR is consolidated as an accessible and valuable alternative to RF projects design, replacing traditional heterodyne radio architecture schemes. The main benefit introduced by SDR is its flexibility achieved by device programming support. Thanks to reduced cost alternatives and diversity of product types, SDR usage is in constant proliferation, covering several domains including industrial and scientific communities, and appearing as a promising contributor towards the development of cognitive radio (CR).

2.1.1 SDR Basic Architecture and Operation

Each SDR design depends on the required performance and capabilities. Commonly, some SDR solutions are designed to signal reception only, such as RTL-SDR. Other devices have transmitting and receive capabilities and extended features, such as Multiple-input and Multiple-output (MIMO) multiplexing, GPS disciplined oscillators (GPSDO) and high processing capability. Examples of this type of device are LimeSDR and USRPs.

Figure 1 summarizes common architecture used in SDR devices, based on (KRISHNAN *et al.*, 2017). The antenna is responsible to receive or transmit encoded information in RF. The antenna is followed by an analog section called RF Front End. This section receives RF signals from the antenna and converts this signal in an Intermediate Frequency (IF), or converts IF signal to a higher frequency signal in order to be transmitted by the antenna in the transmission path. Analog to Digital or Digital to Analog signal conversion are performed by ADC and DAC components. Digital Down Converter (DDC) and Digital Up Converter (DUC) blocks are responsible for channelization and sample rate conversion. Finally, the Digital Signal Processing Unit performs baseband signal processing, (SINHA; VERMA; KUMAR, 2016), and could be programmed by an User Interface Peripheral. Each SDR device has its own Application Programming Interfaces (APIs) and/or drivers required to SDR programming, that must be installed on a host computer.

Figure 1 – SDR Architecture main blocks



Source: (KRISHNAN *et al.*, 2017) modified by the autor

As a core of an SDR device, the Digital Signal Processing Unit performs a set of operations, according to the loaded script. These operations can include encoding, decoding, modulation, demodulation, timing synchronization, digital filtering, Automatic Gain Control (AGC), signal amplification, etc. This unit is composed of one or more processors. Most used processor classes are: Application Specific Integrated Circuit (ASIC), Field-programmable Gate Array (FPGA), Graphics Processing Units (GPUs), General Purpose Processors (GPP) and Digital Signal Processor (DSP). These real-time signal processing classes differ in several attributes such as processing speed, energy consumption, programming language, etc., and there are many factors that influence their choice, for instance performance aspects (FETTE *et al.*, 2009) and market availability (ADAMS, 2002).

2.1.2 SDR Types

In this thesis, three different types of SDR device are employed, the USRP, LimeSDR and the low-cost RTL-SDR. Each one with its own particular hardware characteristics and DSP performing features. Next, main features of these devices are presented.

USRP - Universal Software Radio Peripheral

One of the most commonly used SDR platforms is the Universal Software Radio Peripheral, or USRP. This SDR device is composed of two main components: a motherboard, which contains an FPGA as DSP Unit, for high-speed signal processing, and a daughterboard, which is an interchangeable component that covers different frequency ranges. An illustration of an USRP device (USRP 2932 ©National Instruments) is presented in Figure 2.

Figure 2 – USRP 2932 ©National Instruments

Source: <https://www.yottavolt.com/shop/ni-usrp-29xx-series/>

USRP has a variety of applications, including defense and homeland security, wireless research (CR, MIMO systems, spectrum occupancy), teaching and research (DSP, FPGA design, communication systems). USRP uses an open-source API, the USRP Hardware Drive (UHD). UHD is a user-space library that runs on a GPP and communicates with and controls all the USRP device family (NI, 2020), providing compatibility with SDR programming tools such as GNURadio. If the USRP does not have an embedded GPP processor, it will use the host computer GPP, and then UHD must be installed in this host. Main features of NI-USRP 2932 are presented in Table 1.

Table 1 – Summary of SDR features: USRP 2932, RTL-SDR and LimeSDR

Features	**USRP 2932**	**RTL-SDR (R820T)**	**LimeSDR**
Channels	RX1/TX1, RX2	RX1	RX1, RX2 TX1, TX2
Frequency range (MHz)	400 - 4400	24 - 1766	0.1 - 3800
Bandwidth Max.	20MHz (16 bit sample) 40MHz (8 bit sample)	2.4MHz	61.44MHz
Host Communication	Ethernet Gigabit	USB 2.0	USB 2.0 USB 3.0
Digital Chip	FPGA	RTL2832U R820T	FPRF
Controllable Parameters	frequency, sample rate, bandwidth, gains, IP address	frequency, sample rate, bandwidth, gains	frequency, sample rate, serial number. bandwidth, gains
Additional Features	GPSDO	Tracking Filter	MIMO 2x2

RTL-SDR

RTL-SDR USB dongle is a low cost and largely used SDR device. Generic RTL-SDR device is shown in Figure 3. This kind of SDR can be used for receiving live radio signals using USB connection with a computer. The frequency range of reception dongles varies according to selected device components, ranging from 500 KHz up to 2.2 GHz.

Figure 3 – RTL-SDR USB dongle

Source: <https://hackerwarehouse.com/product/rtlsdr/>

This device was originally designed as a Digital Video Broadcasting (DVB-T), it has been verified the feasibility to turn the exclusive DVB-T tuner device into a wideband software defined radio. The RTL-SDR architecture is based on two main components, the RTL2832U chipset, which consists of an RF digital demodulator that performs ADC conversion, and the tuner. The tuner is a chipset that acts as an RF Front End. Application of RTL-SDR include Frequency Modulation (FM) radio receivers, tracking aircraft and maritime boat positions with ADS-B and Automatic Identification System (AIS) decoding, tracking and receiving meteorological weather balloon data, watching analogue broadcast TV, etc. Highlighted features of R820T RTL-SDR are listed in Table 1.

LimeSDR

LimeSDR, presented in Figure 4, is considered a high-performance and low-power SDR platform. The device is fully open source (board schematic/layout and software) available through Myriad-RF. LimeSDR device application includes: radio astronomy, RADAR, 2G to 4G cellular base station, media streaming, IoT gateway, tire pressure monitoring systems, aviation transponders, among others.

Figure 4 – LimeSDR device

Source: <https://limemicro.com/products/boards/limesdr/>

LimeSDR uses a Field Programmable Radio Frequency (FPRF) chip, the Lime Microsystems LMS7002M transceiver. Highlighted features are listed in Table 1.

2.1.3 SDR Programming Platforms

In order to be able to program SDR devices with a DSP script and operate the device from a personal computer, there is a need for a complementary dedicated software framework. This software framework will be in charge of all low level hardware component interactions, and present an understandable interface whereby DSP script can be drawn up. There are a list of SDR software frameworks options, as seen in (MACHADO-FERANDEZ, 2015), some of the most common solutions used for academic research and industry purposes are: Matlab Simulink/USRP ©*MathWorks Inc.*, Labview ©*National Instruments* and GNURadio Companion ©*GNU Radio Project*.

Once GNURadio is a free and open source radio ecosystem with active development of DSP tools and widespread support in discussion groups, it was chosen as the software platform to build this project. The following subsection presents a brief explanation about this platform.

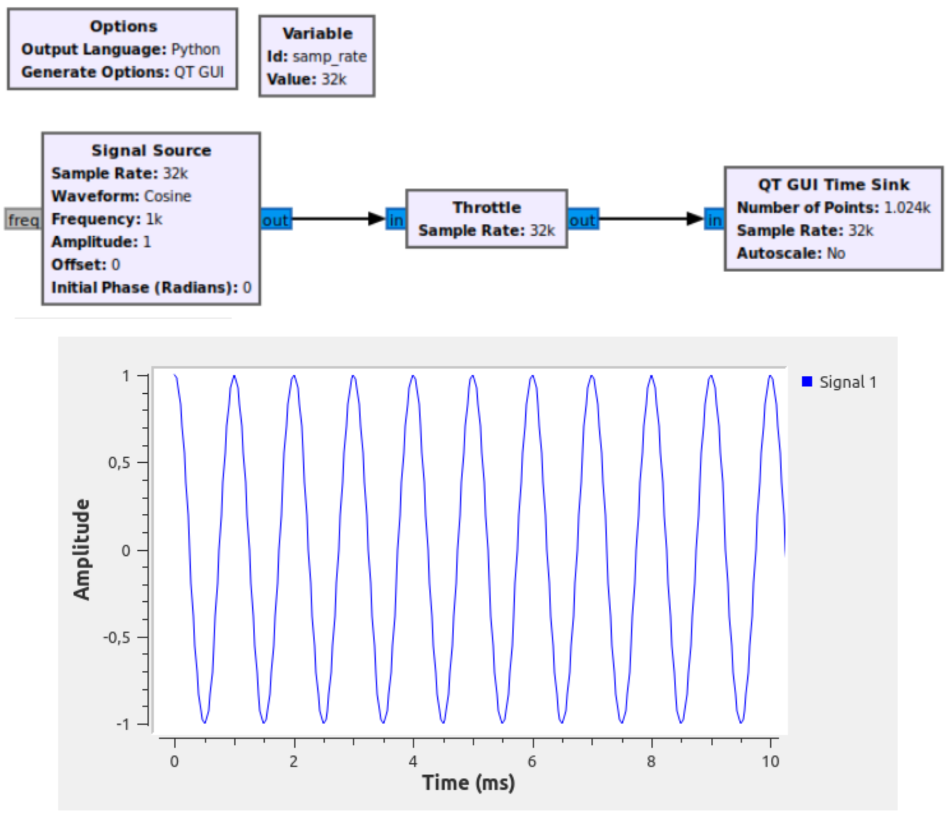
GNURadio

GNURadio is a framework that provides block structures for processing signals for implementation in radio software. It is designed for DSP programming having SDR compatibility. GNURadio is widely used for simulation and implementation of RF communication projects, with great reputation in academia, hobbyists and industry. It offers support for research in wireless communication and a link to build real RF applications with SDR devices. GNURadio is a collective construction framework which has a General Public License (GPL) and can be used freely.

The basic element of GNURadio is the block structure. A block is a unit that performs RF functionality. When installed, GNURadio comes with its own block library. This standard library contains filters, modulators, error corrections, byte operators as well as Fast Fourier Transform (FFT) sinks, stream conversion, USRP blocks, among others. A block can have inputs and outputs. An output of a block can be connected to an input of another block through arrows, with input and output type considerations. A flow of connected blocks compose a flowgraph, which represents the DSP script. GNURadio applies a modular flowgraph based approach to DSP.

Apart from the standard library block that comes with GNURadio installation, there are the so-called out-of-tree modules (OOT). These modules are composed of one or a bunch of blocks created by the GNURadio users. OOT modules allow additional functionality alongside the main GNURadio standard library block. Blocks can be created in python or *C*++ programming languages. A basic flowgraph and its output is shown in Figure 5. This flowgraph (or DSP script) is composed by a *signal source block* that generates a cosine signal, a *throttle block* that imposes a sample rate to the data flow, and a *QT GUI Time sink block* that outputs the cosine signal graph. Once a flowgraph is created and compiled, a corresponding python file is generated. This python file also can be used to run the DSP program.

Figure 5 – GNURadio flowgraph example and its output


Options
Output Language: Python
Generate Options: QT GUI

Variable
Id: samp_rate
Value: 32k

Signal Source
Sample Rate: 32k
Waveform: Cosine
Frequency: 1k
Amplitude: 1
Offset: 0
Initial Phase (Radians): 0

Throttle
Sample Rate: 32k

QT GUI Time Sink
Number of Points: 1.024k
Sample Rate: 32k
Autoscale: No

Source: <https://wiki.gnuradio.org/index.php/Guided_Tutorial_GRC>
modified by the author

GNURadio has standard and OOT blocks that represent the SDR devices. Those blocks output row SDR baseband I/Q samples. Then, the SDR block output can be connected and concatenated to other DSP blocks, which will process and decode the raw I/Q samples. Figure 21 shows an USRP reception script example.

For use of other SDR types, such as LimeSDR and RTL-SDR, there are specific OOT blocks created by the GNURadio community that can be installed and used similarly to standard USRP blocks.

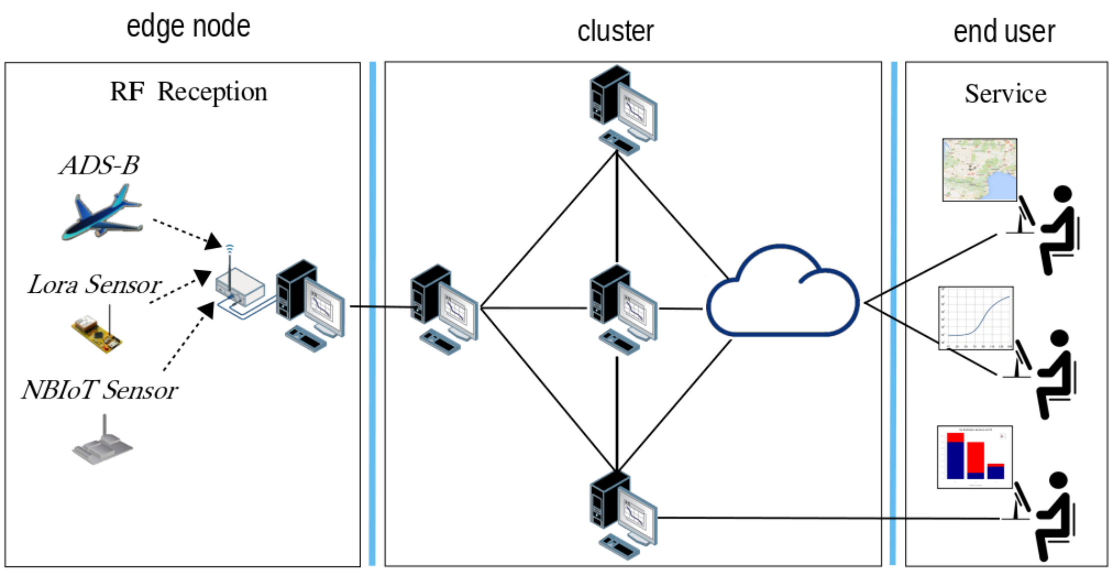
2.2 User application as a service

In this thesis, an end-to-end service can be interpreted as a final user application. It can be a standalone application running only at the host computer, such as an FM or DVB-T image viewer from SDR NOAA weather satellite signal reception, or it's possible to have a client-server base application such as a web page, containing ADS-B airplane real-time tracker service, or maritime boat tracker system, for instance. The client-server service allows client access through a network port using Hypertext Transfer Protocol

(HTTP), therefore in the SDR context, the services are based on SDR data reception as edge network devices. The SDR decoded data is then forwarded to an architecture that will be able to process and create the service. Generally, the architecture provides data manipulation, storage, and communication.

Figure 6 shows a functional illustration scheme to provide real-time services with SDR as edge device. ADS-B, Lora and Narrowband Iot (NBIoT) radio signals are received by an SDR device connected to a computer, (compounding the edge node), and then forwarded to a network cluster. The network cluster may include a cloud computing node and is responsible to create a link to the end users. These users will be able to access final real-time services. The client-server service can be also generated at edge nodes as a standalone application on the host computer.

Figure 6 – SDR end-to-end service generation chain

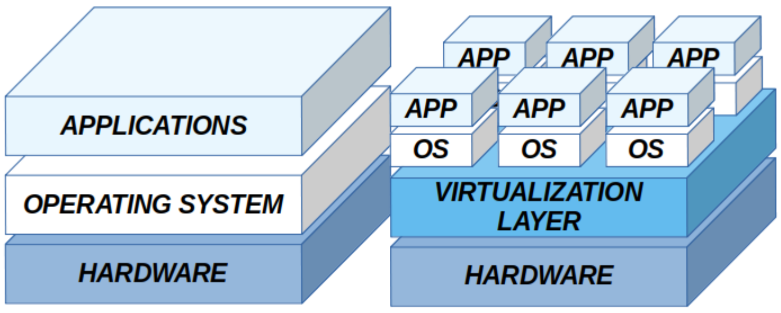


Source: the author

2.3 Virtualization

The process of running a virtual instance of a computer system in a layer abstracted from the actual hardware is called virtualization. The virtualization process allows the hardware elements such as processor, memory, storage of a host computer to be divided into multiple virtual computer instances. Every virtual instance will be allocated above the virtualized layer, as shown in Figure 7, where a comparison with traditional Operation System (OS) layers and virtualization OS layers is presented. The virtualization process provides new instances of virtual computers through hardware resource sharing. The most known form of computing virtualization is the Virtual Machine (VM). VMs behave like independent computers, although they run on a portion of the actual underlying computer hardware, (IBM, 2019).

Figure 7 – Traditional computer OS layers x virtualization OS layers



Source: the author

Today, virtualization is a process largely applied in computing. It is an important element of Information Technology systems (IT) architecture, and it's been also considered a cloud computing foundation. Some highlight benefits brought by virtualization are listed below:

- *Resource sharing*: allows multiple OS creation sharing the same hardware.
- *Reduced capital and operating costs*: replaces traditional hardware functions by VMs.
- *Reduce downtime*: can easily replicate a virtualized service when it crashes, unlike what happens with physical servers.
- *Flexible control*: it's easier to manage a virtualized OS than a physical computer once it's written in software. Virtualized OSs have compatibility with software managements scripts, and it allows dynamically resource allocation, (MALHOTRA; AGARWAL; JAISWAL, 2014).

Virtualization techniques

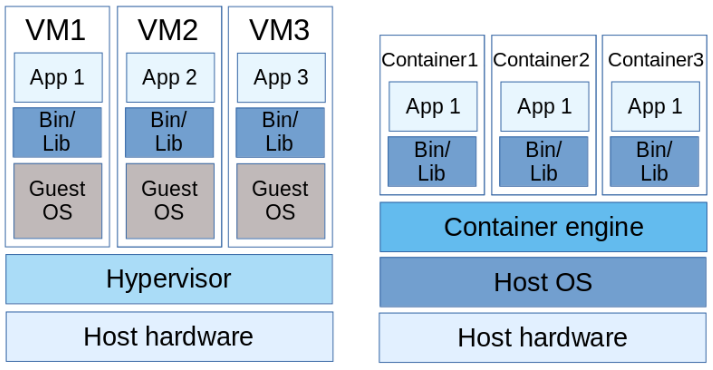
Basically there are three types of virtualization techniques: full virtualization, paravirtualization and OS level virtualization (JAIN; CHOUDHARY, 2016). The first two techniques apply a hypervisor, an abstracted software layer above the computer hardware, that are responsible for creating and running the virtualized OS. In OS level virtualization the host OS has virtualization capabilities itself and carries out the hypervisor function. Container virtualization, also called containerization, is a type of OS level virtualization that provides process isolation.

The software defined architecture presented by this thesis applies the container virtualization method. The next subsection describes container virtualization taking it in contrast with the well known Virtual Machine method.

2.3.1 Containers

Container is a form of virtualization that uses the host OS to create isolated groups of processes, also called virtualized process (COOK, 2017), and dependencies. A container is composed of one or a set of processes that are organized separately from the system (MARTINS *et al.*, 2020). This separation (or isolation) is achieved at OS kernel level by namespace process grouping. Figure 8 illustrates VM and container architectures.

Figure 8 – Virtual Machine and container virtualization architecture



Source: the author

Container architecture has a container engine that handles the container instantiating, bringing virtualization capability. Each container has itself the process, the binaries and library files needed to generate the application. Examples of container engines are OpenVZ, LXC and Docker container.

VM method has a hypervisor, also named Virtual Machine Monitor (VMM), placed above the host hardware, or optionally above the host OS depending on hypervisor type. The hypervisor instantiates the VMs as virtual devices. It generates for each VM an isolated virtual guest OS. This virtual guest OS has its own process, binaries and library files needed for the application, but each virtual guest OS runs as a single, resource-intensive process on the host CPU (COOK, 2017). Examples of VM hypervisors are VirtualBox, VMware and Microsoft Hyper-V.

The isolation principle of the two methods differs. Container virtualization is based on namespace abstraction whereas VM virtualization method is based on virtual device abstraction, (SU, 2020). Each method has its advantages and drawbacks, and the choice depends on many design factors. Hybrid compositions are also possible.

Docker container engine was used in the proposed architecture. The following subsection contains a brief explanation of this tool.

2.3.2 Docker container

Docker is a container engine that creates containerized applications. Basically, the container creation process with Docker, involves two phases. Image creation and running phases. The main components of docker are the image and container entities, listed below, (COOK, 2017; KINNARY, 2018).

- *Docker image*: an image is a template that contains all information for creating the container, i.e. the container foundation. This template consists in a read-only file, called *Dockerfile*, which starts from a base OS image, for instance Ubuntu 18.04, Alpine 3.12.0, Python 3, etc. On the top of this base image, the application stack is composed by adding all needed packages and commands that will run inside the container. Docker image can be seen as the final executable package that contains everything needed for the container embedded application. It comprises the source code, the required libraries, and any dependencies.
- *Docker container*: a container is an instance of a Docker image. When a docker image is running in a host computer, a main process is spawned with its own namespace, this process is the docker container. The main difference between a Docker image and a container consists in the presence of a thin layer known as the container layer. In this layer, read and write commands can be performed. Any changes to the filesystem of a container will be performed in the container layer while the lower layers, which comprise the docker image, remain unchanged.

After the image creation, a *docker run* command can be used to run the container, or *Docker Compose* tool can be applied to run multi-container applications. Using the images created, *Docker Compose* will build, launch and link the containers to provide the final application. Docker compose uses a definition file, written in YAML, called the compose file, that specifies all the images and directives to run the containers. These directives include connection ports, environment variables, restart policies, networking specification, etc.

An important element of container applications is the *Docker registry*, used to store images. These images can then be used as the basis for an application stack. The most known *Docker registry* is *Docker Hub*. Docker Hub stores a lot of images created by the docker community that can be downloaded and used for free. Apart from this, it is also possible to make a local registry to store the images.

A container can communicate with others through a network. The main networking modes explored by this thesis are described below.

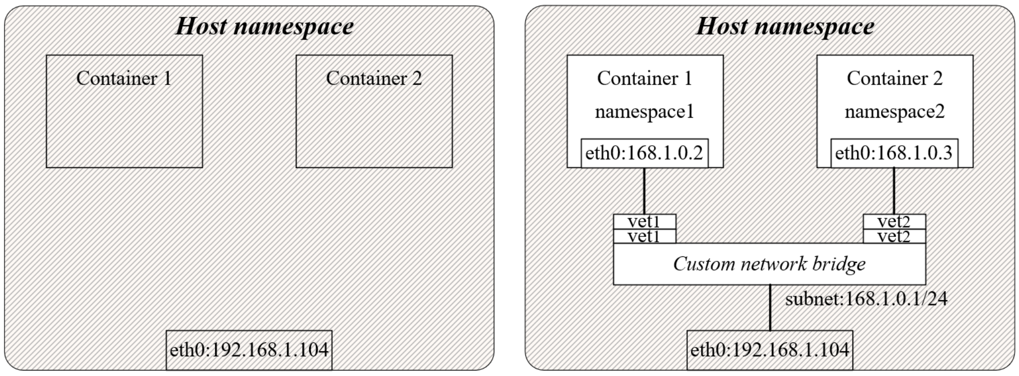
Docker networking modes

Docker engine assigns a network interface for the containers when they are launched. By default, all containers can communicate with each other. Alternatively, there are several network modes that can be configured, in order to determine how the container communication with other containers and the external world will occur. Basically, the network mode is specified by a network drive and network configuration properties. Between the existent modes, three types of networks take great attention: *host networking*, *user bridge networking* and *overlay networking*. Overlay networking is used to provide external access to the containers and also provides multi-host access, for instance in docker Swarm. This option is preferable when a service must be shared and offered in the external network. This thesis is primarily focused on the local scope, then the networks modes of interest are *host networking* and *custom bridge networking*:

- *Host networking*: in this mode, the container will share the host IP address and network namespace. A container IP address is not allocated in this mode. The service that runs inside the container will have the same capabilities as a service running directly on the host (PALURU, 2019).
- *Custom bridge network*: in this mode, the container runs in a private network created internally in the host. Each container has its own network namespace. A Linux network namespace is an isolated copy of the network stack with its own properties, such as routes, firewall rules, etc. The network namespace can be seen as a virtual network barrier that encapsulates processes and isolates their networks connectivity and resources from the other processes (ULUSOY, 2020). Once a custom bridge network is instantiated for a group of containers, the docker daemon connects each container to the custom bridge using a veth device pair. A veth pair is a virtual ethernet device pair which acts as a tunnel between network namespaces, i.e. creating a bridge between them and enabling communication (KERRISK, 2020). The containers that are connected in the same bridge network can communicate with each other. Once they are in the same network the Docker Engine will create the necessary configuration (i.e. internal interfaces, veth pairs, iptables rules etc.), to make this connectivity possible. Finally, the custom bridge network can be connected to the host ethernet interface. Figure 9 shows a network view, highlighting the network namespace isolation of the host networking and custom bridge networking modes, for two containers.

The main difference between the two network modes lies in network isolation. Custom bridges provide better isolation than host networking or default bridge networks, because only the containers attached to the same custom bridge network are able to communicate. Even so, there are some situations where host mode networking can be useful, for instance in performance optimization. Furthermore, because the host network does not require network address translation (NAT), it is suitable when the container must handle a large range of ports. Host networking is available only for Linux hosts.

Figure 9 – Host (in left side) x custom bridge (right side) networking



Source: the author

In turn, custom bridge network (also called user-defined bridge network), can provide automatic DNS resolution between containers, then the containers can communicate with each other by name or alias. The containers of that network will expose all ports to each other, but those ports are only accessible externally through the use of docker publish flag *-p*. Other features include: container attachment/detachment and network configuration on the fly, environment variables sharing with docker compose.

2.4 Communication protocols implemented: ADS-B and LoRaWAN

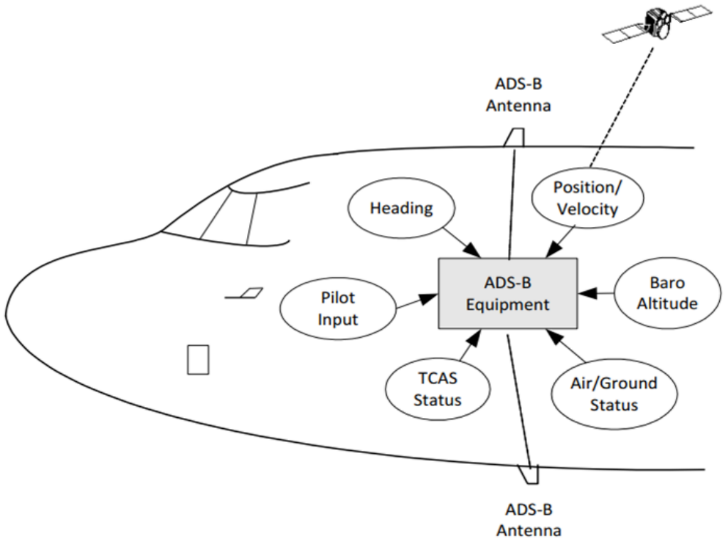
Due to the variety of information contained in ADS-B packets, and thus the possibility of generating different services, the ADS-B service provision was chosen as the focused implementation use case. All the evaluations are made on a scenario in which ADS-B signals are transmitted and received with SDRs. However, with the perspective of growth of applications involving the wireless LPWAN segment, the highest growth: 12% surpassing 10% of 5G (CISCO, 2020), a second use case that implements LoRa modulation technique is briefly presented. The following subsections summarize these communication techniques.

2.4.1 Automatic Dependent Surveillance-Broadcast ADS-B

Automatic Dependent Surveillance-Broadcast (ADS-B) is a surveillance technology that incorporates ground and air equipment to provide Air Traffic Control (ATC), (FAA, 2012). ADS-B works by transmitting, regularly and frequently, data packets containing information from the aircraft. Aircraft information data includes position on the surface (latitude, longitude, altitude), identification code ICAO, speed, vertical rate, heading, among other data. Latitude and longitude are provided by a precision Global Positioning System (GPS) source, and other information is provided by a collection of aircraft sensors, as shown in Figure 10.

ADS-B packets are sent automatically, without any request, within a maximum interval of up to 0.5 seconds. The signal is received by antennas on the ground, and also by any aircraft in their range. ADS-B principle is based on sending as many information packets as possible that would be captured and decoded by the largest number of receivers as possible (EUROCONTROL, 2008).

Figure 10 – ADS-B functional scheme



Source: (FAA, 2012)

The ADS-B system was designed for traffic surveillance and control, with a theoretical range of 250-300 MN (approximately 450 km to 550 km), with the ground receiver, and air-to-air surveillance (between aircraft in flight). For traffic surveillance purpose, ADS-B ground stations receive ADS-B modulated signals, decode and redirect them to the ATC control stations, responsible for presenting this data on the flight controller screen. However, ADS-B data can be free received and decoded with a reception station. To set up a reception station, SDR devices such as USRPs, RTL-SDR dongles and LimeSDR, can be used. ADS-B data is modulated at a frequency of 1.09 GHz, using Pulse Position Modulation (PPM), with 2 MHz bandwidth, and 1 Mbit/s bitrate. The packet consists of a preamble of $8\mu s$, followed by a data field, from $56\mu s$ to $112\mu s$, and a Parity Check - PI error verification, (EUROCAE, 2009).

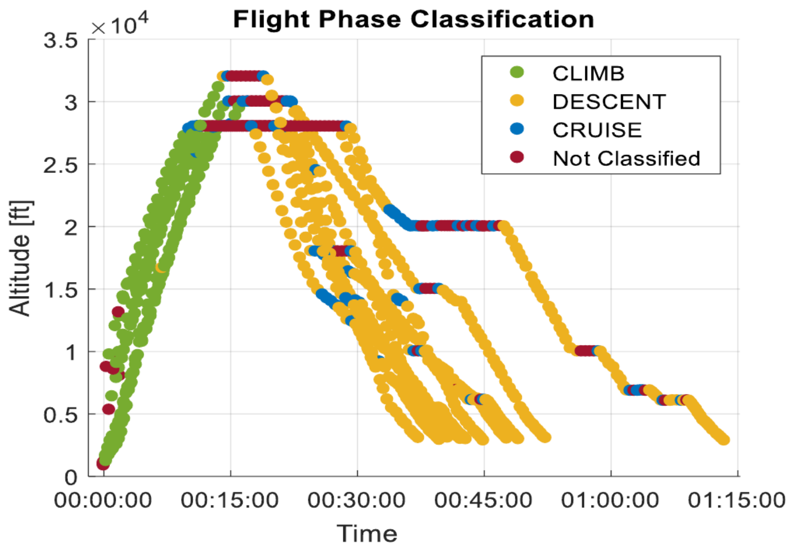
Once the ADS-B data is decoded by an SDR device, this decoded data can be used to serve different services. Examples of services are shown in Figures 11 and 12. Figure 11 presents a real-time aircraft tracker implemented with the Dump1090 program for RTL SDR devices, (SANFILIPPO, 2013), which uses latitude, longitude, aircraft identification and heading information.

Figure 11 – ADS-B flight tracker with dump1090

Source: https://myscope.net/adb-s-flugradar/

Figure 12 presents a flight phase classifier using fuzzy logic and three information fields: altitude, vertical rate and aircraft speed. This classifier determines if an aircraft is in ground, climb, cruise or descent phase, at a given flight time. This figure shows 10 flights in the same time reference, from Santos Dummont (SDU) to Guarulhos (GRU) airports, from 01 to 12 April 2019.

Figure 12 – Flight phase classifier with Fuzzy logic



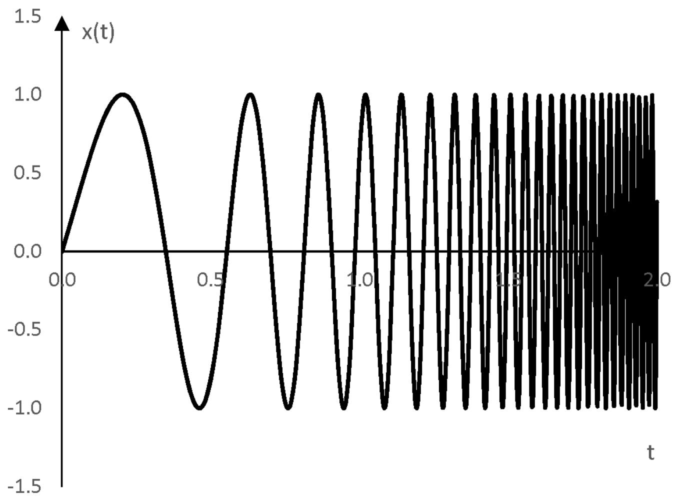
Source: the author

2.4.2 Long Range (LoRa) modulation technique

LoRa is a proprietary modulation technique that applies a spread technique called Chirp Spread Spectrum (CSS), designed to cover long ranges with low power consumption. LoRa is classified as a LPWAN network communication type, that is based on three factors: *long range*: few kilometers (urban) up to 10 km in rural settings, *low power*: power consumption optimized, generally devices using battery; and *low cost*: aiming at reduce complexity of the hardware design, then lower its cost. Applications of LoRa technique include agriculture processing, animal and fleet tracking, air pollution monitoring, etc.

CSS method consisting of sweeping a sinusoidal signal frequency in a defined bandwidth (JOACHIN, 2020). Figure 13 shows a sine sweep signal example. A chirp is defined by a spreading factor (SF) that it uses and the bandwidth (Bw) that it covers. With these two parameters, a symbol period is found $T_s = \frac{2^{SF}}{Bw}$. To encode a symbol, the method used by LoRa consists in adding a starting offset to the frequency sweep, F_{offset}. Then to recover the integer value encoded in each chirp the method used consists in multiplying each received chirp by the complex conjugate of the chirp encoding the value 0, which corresponds to a down chirp. Finally, a Discrete Fourier Transform (DFT) is applied in the dechirped symbol version in order to find the main frequency present, which is equal to F_{offset}.

Figure 13 – Sine sweep signal example



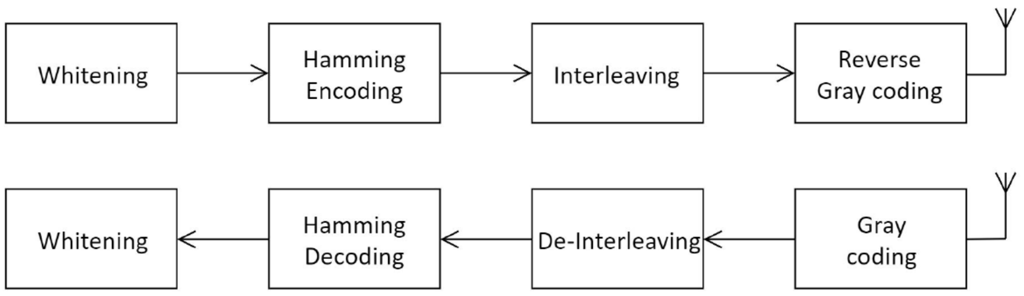
Source:www.recordingblogs.com/wiki/sine-sweep

The LoRa packets are initiated with a header (preamble), composed by non-modulated chirps, which has information for frame synchronization. Therefore, the header has transmission detection symbols, frame synchronization symbols and frequency synchronization symbols. In the encoding/decoding process of a message (e.g. codeword data bits), LoRa has four main blocks that are presented in Figure 14.

In the encoding process, the *whitening* block is firstly applied in order to remove the

DC-Bias[1] thought a XOR operation in the data bits, with a pseudo random sequence. This operation provides the advantage (compared with Manchester coding) of keeping the same data rate, but on the other hand, the process doesn't offer a guarantee that the DC-bias will be removed, only a high probability of it happening. The *Hamming* encoding block is an error correction code that will correct errors, adding redundancy in each codeword. The *interleaving* block spreads the symbols that constitute a codeword between multiple LoRa symbols. It helps in error correction in the reception, because when a symbol is transmitted it can be corrupted by noise or fading, which leads to multiple bit errors on demodulation. Once the same symbol generates those errors, they are highly correlated, making more difficult the task of the error correction codes, (designed to correct random errors). Then the interleaves break this correlation, spreading the errors over multiple code words. This process increases error correction code effectiveness but has the cost of increasing latency, once it is necessary to receive multiple symbols before being able to recover the full codeword.

Figure 14 – Encoding/decoding process used by LoRa



Source: (JOACHIN, 2020)

Finally, the *gray coding* block maps a numeric symbol to a binary sequence, adding one extra bit in the original numeric representation. Gray code is very useful once the data is decoded, and is more likely to misinterpret an adjacent symbol than a random symbol of the codeword. Gray code occurs in reverse order, but it holds its properties, then a mistake leads only to one bit being wrong. In the LoRa decoding process, the reverse encoding process is applied.

[1]DC-Bias is the average amplitude of the waveform. In some communications systems the aim is to have DC-Bias equal to zero called (DC-balanced) (SCHOUHAMER IMMINK; PATROVICS, 1997), to prevent bit errors when passing through circuits.

3 RELATED WORK

In the focused migration of hardware to software in recent network architectures, as well as to provide services in cloud environments, virtualization has proven to be a key technology for flexibility and ease of service management. Nevertheless, when the communication takes the path towards wireless communication, it reaches the radio elements such as SDR's, as transmission/receiver device, and at this edge point, virtualization schemes and its integration can become more complex, combining device diversity, hardware and SDR implementation constraints.

This chapter aims to show a set of works that benchmarks the differences between containerization and other virtualization technologies, as well as draws a clear picture of the similar architectures that explore virtualization and SDR integration. The purpose is to highlight the evaluation metric adopted by these solutions, and main differences when compared to the solution proposed by this thesis.

3.1 Virtualization techniques

An overview of the enabling technologies for network reconfiguration, such as NFV, SDN and SDR, are presented in (LIU *et al.*, 2020), showing the potential to improve the network's flexibility with radio virtualization and network virtualization as an alternative to scarce radio spectrum resources.

Once flexibility brought by virtualization is also intended for radio access technologies, some benchmarks of virtualization techniques have been performed at the network edge, and have demonstrated that lightweight containerization has some benefits when compared with VM or Kernel Virtual Machines (KVM). Applying a queue model to virtualized radio access networks architectures, (GOPALASINGHAM *et al.*, 2017) has shown that container virtualization architecture outperforms VM architecture in inter-arrival time and average waiting time as function of the number of virtual eNodeB launched. To bring NFV to the edge of the network, (CZIVA; PEZAROS, 2017) proposes a low latency container-based platform model to run and orchestrate containers. They argue that a container-based virtualization scheme has not the heavy footprint of traditional or specialized NFV implementation via hypervisors.

There are a lot of studies in literature that evaluate virtualization schemes. (LINGAYAT; BADRE; GUPTA, 2018) compares the virtual machine and Docker container-based hosts performance in terms of CPU performance, memory throughput, disk I/O, load test, and operation speed measurement, observing that Docker containers overcome VM performance in every test. (CHAE; LEE; LEE, 2017) executes a performance benchmark of container versus KVM, showing that Docker uses hardware resources such as CPU, HDD, RAM faster and more efficiently than KVM. This paper also evaluates the performance of both virtualization techniques in a web-server, showing that KVM CPU utilization is higher than docker containers and its RAM memory usage, which is greater 3.6-4.7 times than containers.

Taking advantage of container facilities, such as easy of management, rapid deployment, scalability etc., the usage of this virtualization technology is widely adopted in a broad range of network service, leveraged by cloud computing (Microsoft Azure, AWS Lambda, Google Cloud, etc.), and industry production. In spite of significant momentum, container virtualization is still considered a new technology, with limitations, shortcomings and improvements to take into account. By sharing the host OS, containers don't provide isolation as VMs, being more exposed to security issues. Furthermore, there is a lack of tools for real-time container management (STRUHÁR *et al.*, 2020), and also a need for real-time communication effective mechanism among containers (MOGA; SIVANTHI; FRANKE, 2016). However, containers has great adoption in real-time applications thanks to boot-up process (DUA; RAJA; KAKADIA, 2014), which takes much less time than VMs.

3.2 SDR virtualization

(KIST *et al.*, 2017) has shown that physical parts of the SDR can also be virtualized. They propose Hydra, an SDR virtualization layer that creates virtual RF front-ends. Using GNURadio blocks and applying FFTs and IFFTs in the signal, Hydra can slice the available radio spectrum of an SDR in multiple virtual RF front-ends that works as individual TX/RX channels. Being developed as a hypervisor for mobile networks, Hydra acts splitting IQ signal samples from different virtual RF front-ends into a single transmit/receive vector. In (KIST *et al.*, 2018), the authors present a proof-of-concept which applies LTE and NB-IoT protocols, compressing to transmit IQ samples of both protocols into a single waveform. One of the evaluation metrics of this solution compares the overhead imposed by Hydra in terms of CPU utilization using different IFFTs lengths.

In microservices provision context at wired networks, (MARTINS *et al.*, 2020) proposed an easy-to-use container-based architecture that provides network management using Netflow traffic collection. Applying a model based on container virtualization, this paper shows that a network service, (a Web-based client-server service using HTTP protocol), can be deployed together with traffic collection lightweight containers. This architecture doesn't include radio access, but it is an example of how a lightweight container-based architecture can be controlled by a virtualized layer, i.e. a control plane. The evaluation of this solution analyses the overhead imposed by the architecture when implementing the service, compared with the case where the service is implemented without the architecture. Computational resources as RAM memory and CPU utilization in heavy stress (i.e. increasingly user requests) are monitored.

(CARPIO; DELGADO; JUKAN, 2020) proposed an architecture to provide services in the IoT context for cloud computing. This solution presents a benchmark study of latency, resource utilization and scalability, from an end-to-end IoT service provided in three scenarios: cloud-only, edge-only and edge-cloud. The architecture uses lightweight containerization, embedding the IoT radio, (Raspberry Pi 3), in a docker container. The event streaming platform Kafka is employed in data streaming and Firebase as an open source cloud. This architecture is very similar to the proposed solution of this thesis, which also uses lightweight containerization to embed the SDR radio and its DSP script. But the main differences concern:

- *The architecture scope purpose*: presented solution aims to evaluate end-to-end IoT cloud services, (wired network communication), this thesis aims to provide end-to-end general radio services limited in edge scope (Web-based client-server service in local network), including wireless communication, but also focusing in how different services can be created using available decoded data.
- *The data stream method*: instead of Event Streaming Process (ESP), this thesis applies an embeddable networking library ZeroMQ to speed up the data stream between the containers.
- *The use cases*: the authors show an IoT use case, where the final service is accessed by Json variables in the server. This thesis presents ADS-B (web page services) and LoRa (file transfer service) use cases.

(AHMED; ALLEG; MARIE-MAGDELAINE, 2019) investigates the provision of NFV in IoT systems in order to provide services in a more flexible and active way. For this purpose, it presents a reference architecture, based on ETSI guideline for NFV reference architecture framework (ETSI, 2013). The decoupling of hardware and software exposes a new set of model, suited for IoT systems. This architecture creates lightweight containers as virtualized network functions (VNF) from IoT and links them to the Physical Network Functions (PNF), applied at radio hardware. Then an IoT Service is represented by a sequence of PNFs and VNFs instances. As a proof-of-concept, they present a case of service implementation using a set of Raspberry Pi 3 model, acting as IoT devices.

In industrial context (TASCI; MELCHER; VERL, 2018) proposes a container-based architecture for real-time control applications, measuring the impact of running containers in embedded devices. In the architecture, evaluation of round-trip time and latency of container-to-container communication (TCP, UDP and IPC) are performed, showing that real-time requirements can be achieved through kernel manipulation and fast network library support implementation (ZeroMQ).

(IMRITH *et al.*, 2020) present a container-based architecture to provide services from IoT devices, considering security aspects. This architecture launches containers as instances of security Intrusion Detection and Prevention Systems (IDPSs), that analyses the traffic from IoT devices on a fog infrastructure. This infrastructure is emulated via a Raspberry Pi-4, which has an eligible number of IDPS built-in containers. The evaluation takes into account container resource utilization (RAM, CPU usage), dropped and alerted container attack percentage. Besides the architecture, the paper provides a strategy of orchestration for security services management in the edge node.

Table 2 summarizes the main architectures that apply virtualization, showing mainly aspects and differences in comparison with this thesis.

Table 2 – Platform virtualization architectures

Authors	**Context**	**Virtualized elements**	**Contributions**	**Main differences**	**Plat-form**
KIST et al. (2017, 2018)	Mobile Networks	SDR RF Front End	Hypervisor for SDR Front End virtualization	- SDR RF front-end virtualization - Use cases: LTE, NBIoT	USRP
IMRITH et al. (2020)	IoT	RPi-4	Container-based architecture for security services management at the edge	- RPi-4 virtualized as a security tool - Use case: IoT traffic emulation	RPi-4
AHMED et al. (2020)	IoT	VNFs for IoT	Container-based architecture to provide IoT services, based on ETSI NFV	- ETSI NFV based architecture - Use case: IoT	RPi-3
CARPIO et al. (2020)	IoT (cloud)	RPi-3, Application	Container-based architecture to provide services from IoT devices	- Communication: Kafka - Use case: IoT (cloud)	RPi-3
This thesis (2021)	General Radio projects	SDR, Application	Container-based architecture to provide services from SDR devices	- Communication: ZeroMQ - Use case: ADS-B, LoRa (local)	USRP Lime RTL

4 PROPOSED ARCHITECTURE SOLUTION

In this chapter, the container-based proposed solution is presented. This solution takes into account a client request of a service where an SDR device is used to receive data. It is based on OOT GNURadio block (HOSTETTER, 2019). It was developed based on the *direct solution* which consists in running the necessary scripts (python, java, GNURadio, etc.) to provide an end-to-end service from an SDR device directly, without any container architecture. This solution considers that the signal will face a sequence of treatment stages, (or blocks), that performs signal or data processing. The implementation of these stages for ADS-B use case is discussed in detail in Chapter 5.

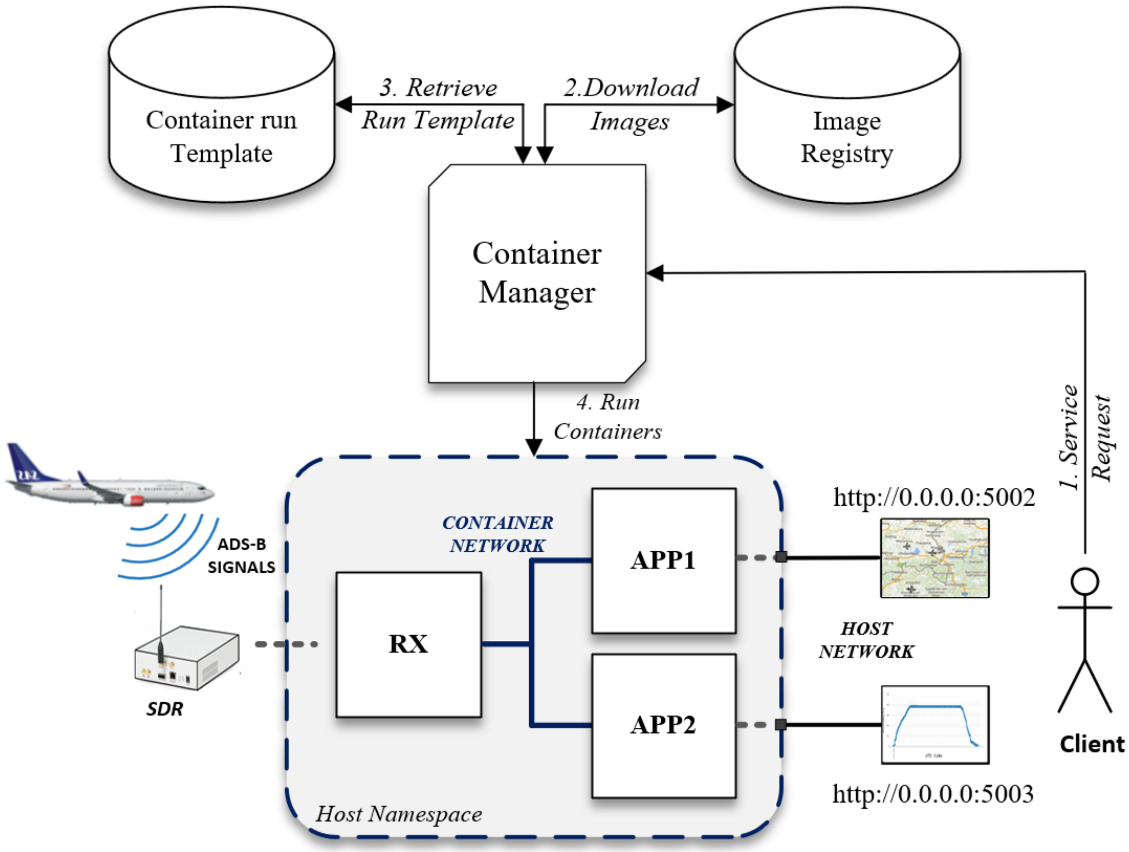
Firstly, all the components of the OOT module (that has an airplane tracker service) were embedded into containers. Then an architecture to manipulate and orchestrate the containers was proposed, with the addition of a new container-based service, the airplane altitude tracker. Once the container-based architecture is validated for ADS-B services, the solution can then be generalized for other RF projects. Therefore, in a second scenario, the architecture explores LoRa modulation technique to provide the services.

The proposed architecture applies container virtualization to embed the identified stages and their functions. The architecture aims to achieve the container management necessary to provide the service. It manages and orchestrates the container creating process according to the client input. The architecture brings the possibility to create an internal and isolated network between the receiver and application containers, or even launch the containers in a host network.

In the scope of this work, the provision of ADS-B services is limited to real-time tracker services, in this case, HyperText Markup Language (HTML) pages. For this first proposition, it is limited to a local network, therefore the client can access the service through a web browser, accessing a local HTTP server at a host computer port. The main motivation to apply container virtualization is to take benefit of its technologies features among which stands out their configuration flexibility, scalability and portability, as discussed in Section 2.3.

Proposed solution, composed by three main blocks, is shown in Figure 15, illustrating a scenario where ADS-B real-time services, (flight and altitude tracker) are provided. The architecture can provide a service thought fourth main blocks interaction steps. These interactions are presented in the sequence diagram in Figure 16, and explained below:

Figure 15 – Proposed Solution



Source: the author

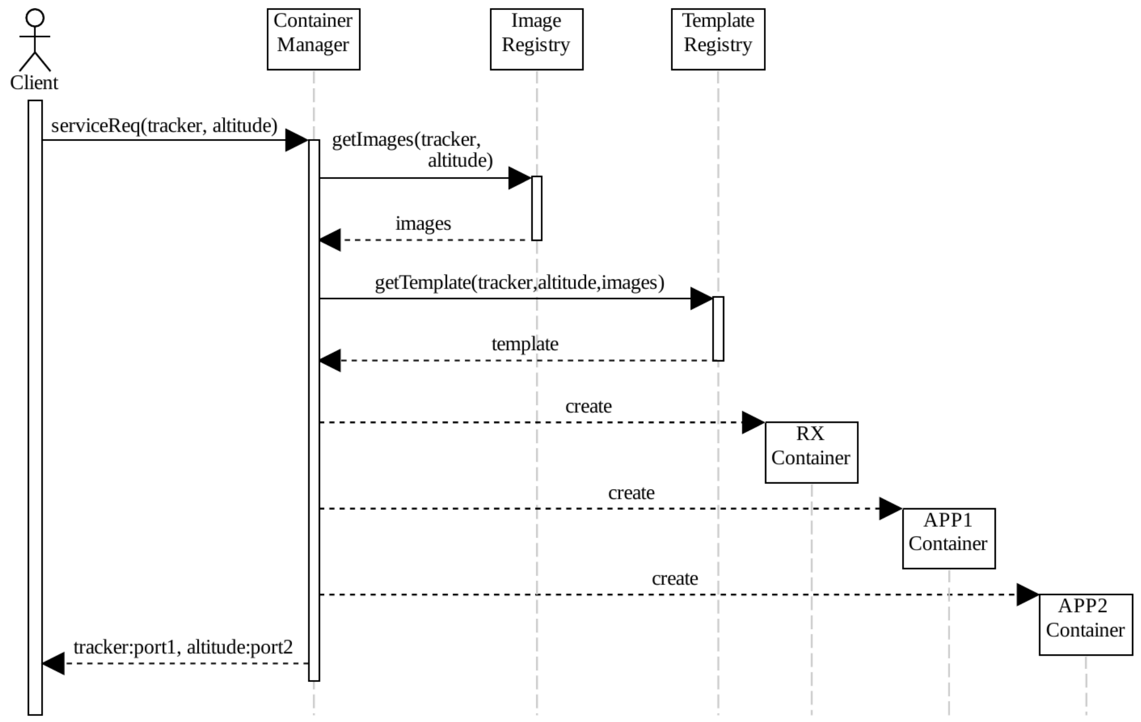
1. The *client* requests a service. In this step, the client sends a message to the Container Manager requesting *tracker* and *altitude* services. This step is represented by *serviceReq(tracker, altitude)* message in Figure 16. It is possible to specify a host port to access each service. If it is not specified, the architecture will mount in default ports, (5002 to *tracker* service and 5003 to *altitude* service).
2. The *Container Manager*, applied through a bash script, will require the necessary images to the *Image Registry* block, (*Docker Hub* was used, but a local image registry can also be mounted). This step is represented by *getImages(tracker, altitude)* message in Figure 16. For the illustrated example, the *USRP RX container*, *tracker* and *altitude* container images are downloaded from the repository.
3. The *Container Manager* will require to the *Container run Template* block a run template, (a docker-compose file, or a docker run CLI directive), for the corresponding service. This step is represented by *getTemplate(tracker, altitude, images)* message in Figure 16. This template corresponds to the container's configuration setup, with

network definition, start policies, environment variables, container run order, etc. Examples of templates are show in APPENDIX A, B, C and D.

4. Finally, the *Container Manager* will run the obtained template which create the containers. This step is represented by *create* messages in Figure 16. In ADS-B use case, the containers are launched in the following order:
    4.1 *Receiver* container, (RX in Figure 15).
    4.2 *Tracker* container, (App1 in Figure 15).
    4.3 *Altitude* container, (App2 in Figure 15).

Once the containers are correctly created, services can be accessed through a web browser on the specified ports, (or default ports if not specified). The RX container is responsible to receive and decode the data is the first container that should be launched, because it provides de data to the application containers to generate their services. Therefore, after starting the receiver container, the order of the application containers is not critical and can be changeable.

Figure 16 – Sequence diagram of ADS-B service request



Source: the author

Besides the *direct solution*, i.e. the traditional method to provide the services, using the proposed architecture topology presented, SDR based services can be generated by different ways, which are related to the container network mode employed. Henceforth, three modes of service provision are defined and listed below:

1. *Direct solution*: this is the traditional mode whereas the SDR programming platform (GNURadio), as well as all the applications and dependencies needed to gen-

erate the service, have to be in installed and run directly in the host machine.

2. Using the *architecture host* mode: containers are launched in host networking mode, as explained in Section 2.3.2, this mode uses the containers to generate the services sharing the host machine network. To facilitate the terminology, this mode is called hereinafter as *architecture host*.
3. Using the *architecture net* mode: containers are launched in a custom bridge network, i.e. an isolated container network, as explained in Section 2.3.2. To facilitate the terminology, this mode is called hereinafter as *architecture net*.

The architecture presented explores a containerized solution to generate the service. In this case, all software components needed to provide an end-to-end service from SDR devices are embedded in the containers. The containers are the only components needed to generate these services. Once the containers are created and validated, the architecture manages and orchestrates the service deployment for a client request.

In order to implement a container-based architecture to provide end-to-end services using SDR devices, some questions arise. These questions, listed below, reflect the main difficulties in carrying out this work:

- How to embed SDR applications into containers? What functionality each container will perform in the end-to-end service provision chain? How to define the communication system between the containers and which data will be shared between them?
- How to make the transition between direct and container implementation of DSP script keeping the possibility of setting the RF communication parameters, (such as gains, frequency and threshold adjustments, etc.)?
- And finally, once the containerization is performed, how to define the architecture topology in a primary local implementation, considering that this implementation can serve as the basis for more complex implementation such as distributed systems, in an external network context, where the containers collaborate to generate the final service.

The next chapter will show how these questions can be answered with the architecture implementation in the ADS-B service provision use case.

4.1 Proposed architecture and distributed systems integration

A distributed system is a system where multiple components (located at different machines) communicate and coordinate actions to fulfill a goal. Actually a lot of web services involve distributed systems, such as Amazon platforms, Google Search engine, Netflix, electronic bank services, gaming, etc. When a service is the goal, the distributed system appears as a single coherent system to the end-user.

A distributed system is composed by modular elements called nodes. A node can be a hardware, a software, a container, or any other element that have memory, and which can connect and communicate through the network. Highlight characteristics of distributed systems are: the nodes run asynchronously and concurrently, the nodes are independent, i.e. all components fail independently of each other.

Provide a service using a distributed system can be an attractive solution once it can bring some positive aspects, such as *scalability*: a node is independent then it is easy to add additional nodes to create new functionality; *Reliability*: most of distributed systems are fault-tolerant, it means that if a problem occurs with a node another node will be launched to replace the failed node, then the nodes will still work together and individual node fails are usually transparent to the end-user service; *Performance*: in distributed systems workloads can be broken down and distributed to multiple nodes, (which can be run on different machines), increasing efficiency when compared to a non-distributed system.

Examples of SDR integration with distributed systems are ADS-B tracker services, such as FlightAware© and FlightRadar24©. The live worldwide airplane tracker service is a collection of airplanes position, that are decoded by multiples SDR-based reception stations. These stations compose the distributed network and continuously send decoded information to feed a central server.

The container-based architecture is not primary focused on distributed systems integration, but further applications can explore this possibility. To make the containers applicable as a distributed system nodes, the following considerations should be taken into account:

- *The local architecture scope must be changed to an external scope*. In this case, the containers will be launched in different machines and the function performed by the *Container Manager* must be replaced by another container orchestration tool, such as Swarm and Kubernets.
- *Create reliable containers with their own health check and fault-tolerance systems*. The services created by the architecture has services verifier to assert end-user service availability. This mechanism is described in detail in Section 5.4. Basically, this method exchange messages between the containers and the *Container Manager* block. These messages are added in step 4 of architecture interaction steps (shown in figures 15 and 16). Applying this message-based method brings simplicity (desired for this first architecture implementation) but as a consequence it turns the architecture integration with distributed systems unfeasible, since the verification of the service requires interaction between the container and the administrator, (it is not performed by the container itself). Furthermore, a fault detection system is not implemented. As alternatives for further improvements, Docker container engine as well as Swarm container health check and fault detection mechanisms can be

explored instead of the presented service verifiers.

4.2 Proposed architecture, NFV and SDN technology integration

Following the hardware to software trend, the adoption and exploitation of softwarization technologies such as NFV and SDN is becoming increasingly common in network environments.

Traditionally NFV is structured according to a NFV Infrastructure NFVI model, which in practice is usually based on monolithic VM and the use of a hypervisor. However, Linux containers are gaining ground in NFV implementation. Thanks to their lightness of resource utilization, containers can provide elasticity at runtime to facilitate the auto-scaling of VNFs. In addition, containers don't need a hypervisor, and management tools applied to clusters and distributed systems, such as kubernets, bring attractive functions for the development of container based VNFs. These functions include container self-healing, auto replacement, continuous monitoring and restart policies. In the light of the growing implementation of container-based VNFs, the industry has sought to reach a consensus of a framework for this implementation. Recently ETSI has published the first normative specification for NFV on "Cloud-native VNFs and Container Infrastructure management", (ETSI, 2020), the first set of Cloud-Native VNF orchestration specifications. This effort demonstrates the container potential and insertion in NFV development.

In turn, SDN also applies resource abstraction, but it is focused on the separation of control and data plane of network communication. The control plane controls how data packets are forwarded, selecting which packet will be sent and, applying all necessary functions and process, it determines which path will be used to send the packet. Some network process, such as implementing the router network protocol, creating the routing tables, for instance, are considered part of control plane. In contrast, the data plane is a low level plane that is responsible for moving packets from source to destination, based on the information provided by the control plane. There is also a great effort in the implementation and development of SDN with container technology. Service providers are focusing on the integration of container and SDN to generate application and microservices in distributed systems.

On the other hand, container implementation in the context of NFV and SDN is still in maturity stage and presents several challenges to be explored. The main bottlenecks are the well known security (isolation) and OS limitation issues.

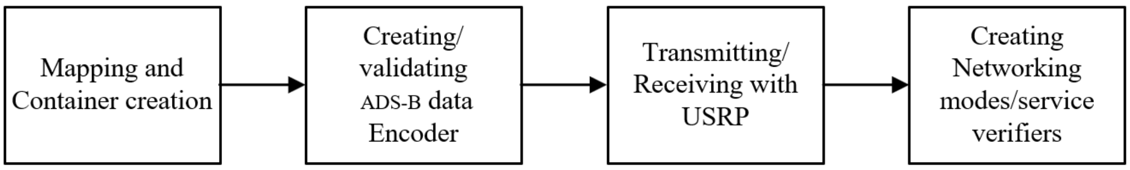
The common point of the technologies presented and the proposed architecture consists in the convergence of hardware abstraction and virtualization in network environments. These technologies, as well as cloud computing and cognitive radio, are been considered key enablers for the promising software-defined systems in information technology. The hardware abstraction proposed by SDS aims to bring management facilities and flexibility in service provision.

This work has a limited scope, presenting an architecture for SDRs integration with network environments to provide end-to-end services. However, it demonstrates a possible method for SDR function containerization, while preserving the radio parameters configuration option. In accordance with SDS, the architecture can be considered as a start point for further implementation with NFV and SDN technologies in distributed systems.

5 IMPLEMENTING THE ARCHITECTURE

In order to implement the architecture with the ADS-B use case, and provide *tracker* and *altitude* services, an implementation sequence of four project phases was defined, as shown in Figure 17.

Figure 17 – Implementation phases for ADS-B service provision


Mapping and Container creation
Creating/ validating ADS-B data Encoder
Transmitting/ Receiving with USRP
Creating Networking modes/service verifiers

Source: the author

The first phase consists in a mapping and container creation phase. Mapping step decouple the solution in modular blocks, pointing out their inputs and outputs, and the container creation step, embedding these blocks into containers. After the container creation, in order to be able to generate and transmit ADS-B data, a data encoder was created in phase 2. This encoder was validated with a simulation DSP script. After that, in phase 3, the ADS-B data was transmitted and received in an experimental setup. The transmission and reception of ADS-B data is validated with the generation of the services. Finally, in phase 4, architecture networking options were created, and service verifiers implemented to ensure the service availability. After conclude these implementation phases, sections 5.1 to 5.4, the architecture is validated through visual inspection of the services, in Section 5.5.

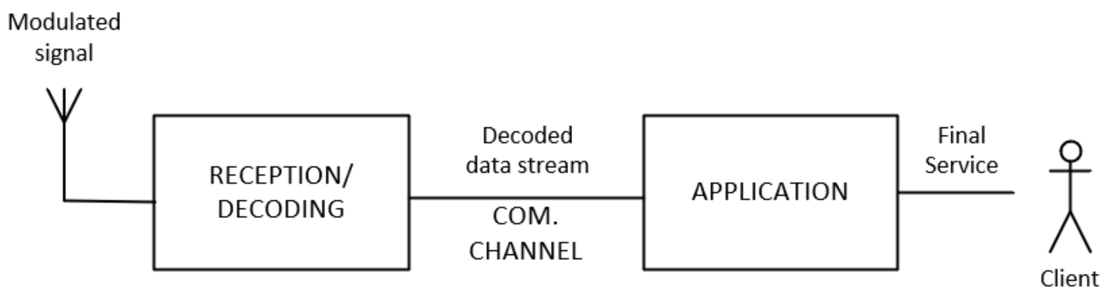
5.1 Mapping and container creation phase

The modular blocks resulting from the mapping step, and the container creation, are explained as follows.

Mapping signal path blocks

First and foremost two stages were identified in the signal path: the reception and decoding stage and the application stage, as seen in the Figure 18. In general terms, the input of the reception and decoding stage is the modulated signal, being represented by the SDR and the DSP decoding script, whereas its output is the decoded data stream. The decoded data stream is the input of the application stage, which generates as output, the final service. The service output depends on the client request.

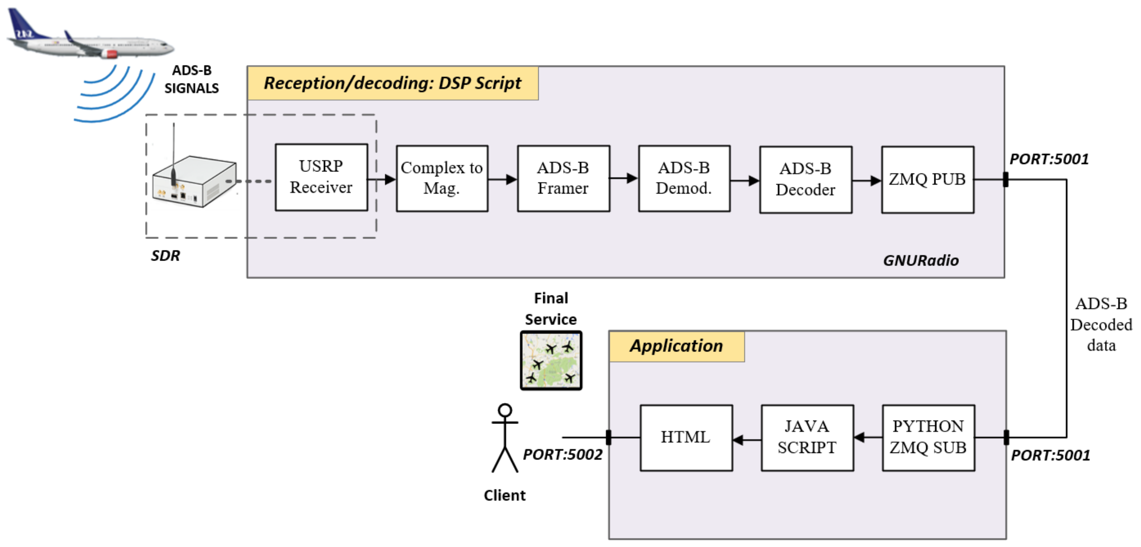
Figure 18 – Signal path stages for service provision with SDRs



Source: the author

Figure 18 presents the main stages of the end-to-end service chain. A deep-picture of the signal path stages with internal blocks is given in Figure 19, with an ADS-B use case. This case shows the example of airplane tracker service, generated from the ADS-B airplane signal reception from the SDR. The *reception/decoding* block is implemented with a GNURadio script, based on (HOSTETTER, 2019) out-of-tree GNURadio project.

Figure 19 – Signal path stages for ADS-B service provision with USRP



Source: the author

In the above scheme from Figure 19, the reception/decoding, and application stages have internal blocks that perform operation in the streaming data. These blocks and operations are briefly summarized below:

Receiving/decoding stage: composed by the DSP GNURadio flowgraph.

1. *USRP Receiver*: this block represents the SDR in use, in this case the USRP receiver. The output of this block consists of I/Q digital samples in complex format.
2. *Complex to Mag.*: ADS-B is modulated with PPM which has non-coherent reception, the phase component is not used in reception. Then this block selects only the real component (called *In-phase* component I), of the complex data signal stream.
3. *ADS-B Framer*: identifies the start and end of an ADS-B information packet, at a level of symbols, add tags and forward the packets to the ADS-B Demodulator block.
4. *ADS-B Demod.*: the demodulator receives the symbolic version of the packet and extracts its binary version.
5. *ADS-B Decoder*: working at a bit level, this block decodes the whole ADS-B packet and stream the aircraft information data: ICAO code, callsign, latitude, longitude, altitude, vertical rate, velocity, heading, timestamp and the number of messages received by the aircraft.
6. *ZMQ PUB*: this block implements a communication socket, being responsible for forwarding the decoded information to a specific IP:PORT address using TCP protocol. It's called ZMQ PUB, abbreviation for publisher. Therefore, this block will establish a communication channel allowing aircraft information data to be retrieved at the IP:PORT pair specified.

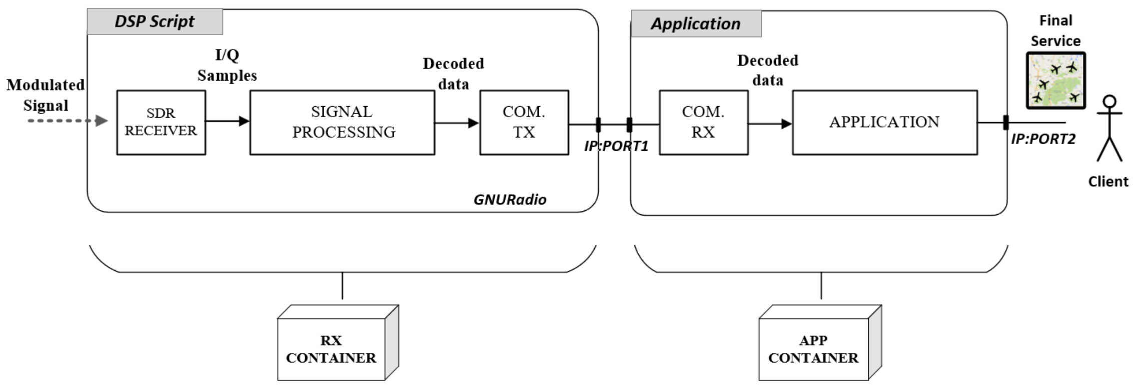
Application: the application is responsible to process the decoded data and offer the final service. This topology presents a REST (representational state transfer) web page API. The following blocks are used:

1. *Python ZMQ Sub*: a python script that establishes a subscriber. It receives and transfers the aircraft information to the javascript file. This script also enables the HTTP server. Then the client is able to follow the service, at the address specified, (e.g. http://localhost:5002).
2. *Java Script*: the javascript file is responsible for manipulating the objects created by the HTML page. This block triggs the actions in the HTML page. Then, once the HTML page is created, the javascript is the means by which it is possible to interact with it.
3. *HTML*: generates the page static visual elements (i.e. the objects).

By introducing the main stages in the signal path, and through the analysis of the presented example, some notes about the topology of the *direct solution* are highlighted. The choice of the SDR device only affects the first block in the DSP script, once the SDR block outputs I/Q complex samples. Therefore, it is possible to switch SDR device without affect the block chain; to implement another RF project the DSP script can be replaced and adapted. Finally, once the service is verified and validated, it is possible to transfer its functionality to the containers.

With these remarks, a general block scheme can be drafted for the signal stages, as seen in Figure 20. This view generalizes the functions. Pointing out their inputs and outputs. SDR receiver block receives the modulated signal, extracting its I/Q digital samples. After processing the I/Q samples, the signal processing blocks output the decoded data, which is transmitted from COM. TX to the COM RX block (using IP:PORT pair 1). Finally, the application blocks will process the decoded data and generate the final service at IP:PORT pair 2. Based on this block's topology, two kinds of containers are proposed: a receiver container (RX CONTAINER), for reception/decoding, and an application container (APP CONTAINER), which will perform the application stage functions.

Figure 20 – General block stages for service ADS-B service provision



Source: the author

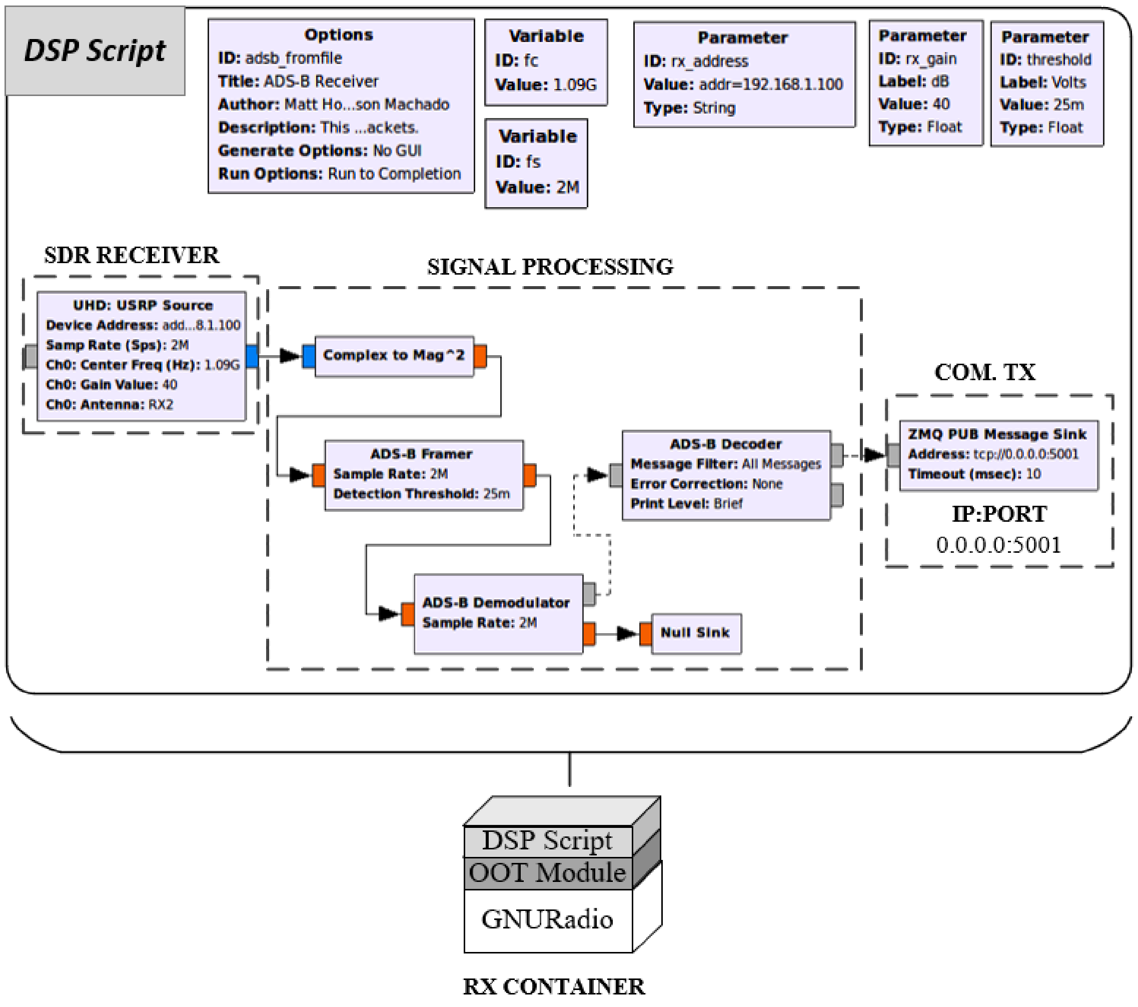
Container creation

Once the communication path is mapped, the container creation was started for the two container types. The process of the receiver container and application container creation for ADS-B service provision, have followed the image creation and container running steps, cited in subsection 2.3.2. For this use case, a receiver container and two application containers were developed: a real-time flight tracker, henceforth called just *tracker* container, and a real-time altitude tracker, henceforth called *altitude* container, that shows an altitude graph of an aircraft. The container creation process is briefly described below:

Receiver container: image creation adds GNURadio framework, gr-ADSB OOT module, and the GNURadio flowgraph as DSP reception script. To make the main RF parameters, (SDR RX gain, USRP address, and threshold level), available to be configured at container runtime an environment variable assignment technique [1] was applied.

Figure 21 presents the DSP script which was passed on to the receiver container.

Figure 21 – GNURadio DSP reception script for ADS-B

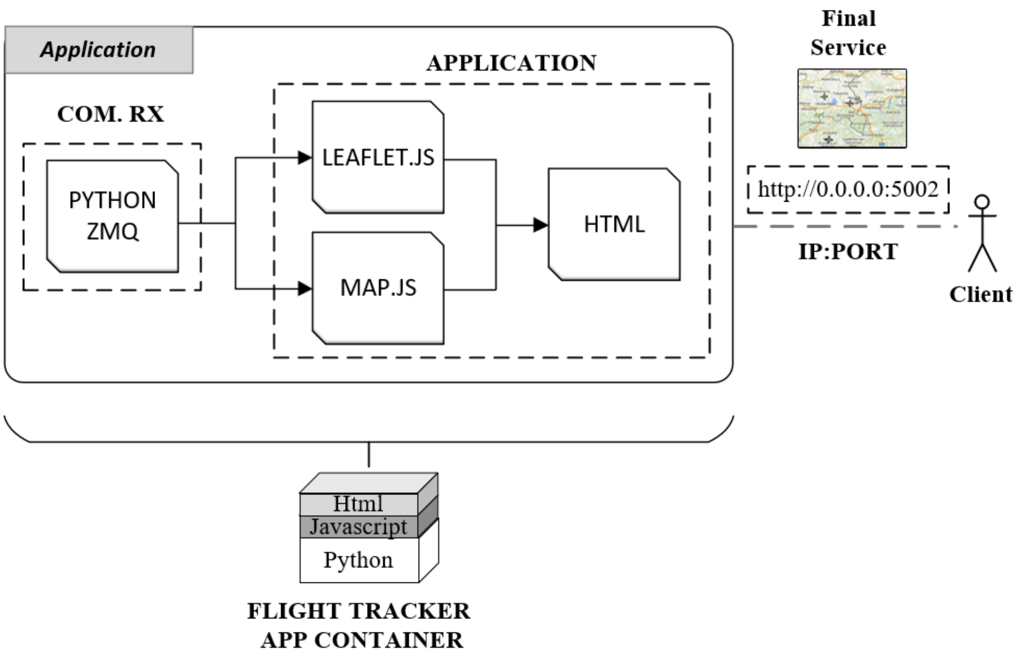


Source: the author

Application container: image creation for the *tracker* and *altitude* containers adds the necessary blocks to generate the service from the decoded data. (python, javascript, HTML file). Figures 22 and 23 show examples of application block files for ADS-B *tracker* and *altitude* services, respectively.

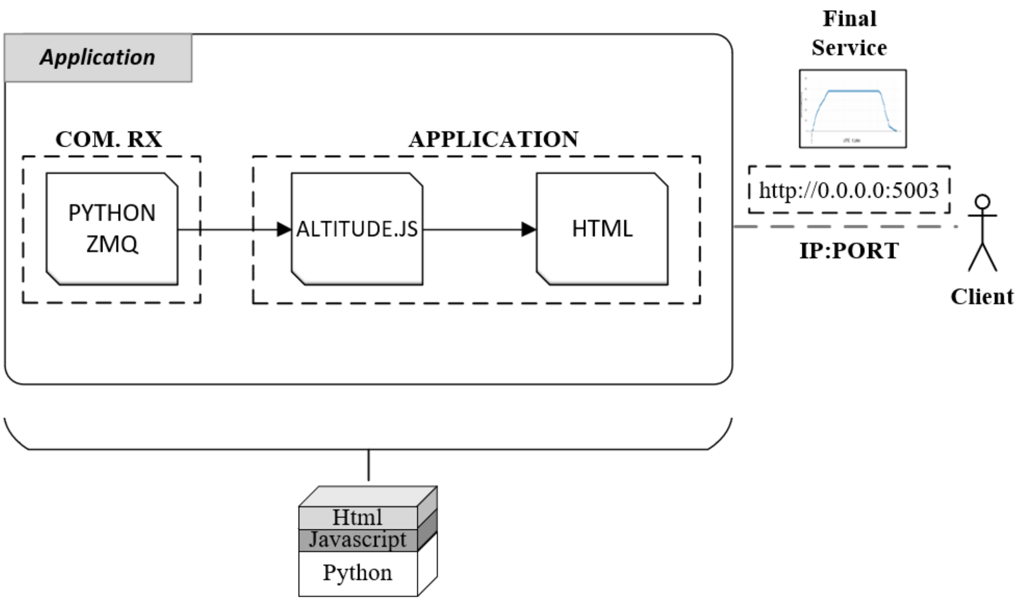
[1] An environment variable is a variable whose value is set outside some program. Environment variables are stored in the system, being composed by a name/value pair. This name/value pair is retrieved and accessed by a program which aims to use its value.

Figure 22 – Application files embed on to the *tracker* container



Source: the author

Figure 23 – Application files embed on to the *altitude* container



Source: the author

5.2 Creating and validating ADS-B data encoder

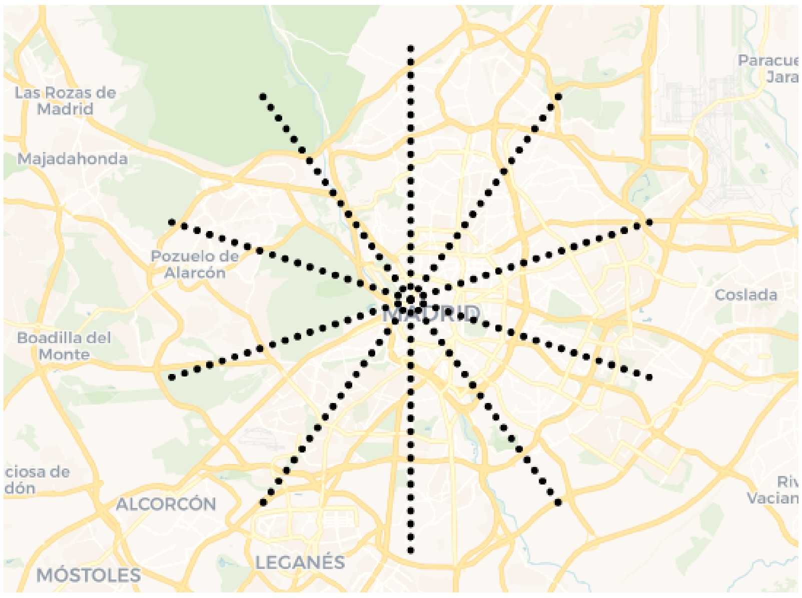
The architecture focuses service generation trough data reception. Nonetheless, to receive ADS-B data, it was necessary to generate an ADS-B transmitter. The transmitter is composed by a GNURadio DSP script that reads airplane dummy data (I/Q samples from a bin file) and sends it through an USRP. The transmitter creation helps to standardize the evaluation tests, creating and sending continuously an aircraft data set. The transmitter also helps to avoid being influenced by ADS-B signal scarcity, in the case of real aircraft reception. To generate the bin files for the ADS-B transmitter, an encoder was created, and validated by a simulation script using GNURadio.

Creating ADS-B data encoder

Initially an OOT GNURadio block developed in an earlier project, that encoded two static airplanes (lat, lon) positions was tested for ADS-B transmission, (MACHADO, 2020). After that, in order to control the number of airplanes created and vary their positions in the map, creating routes as real aircraft, a tailored ADS-B encoder version was created. The encoder creates and stores the I/Q samples in a bin file, later used in SDR ADS-B transmission.

The encoder creates routes, a set of lat, lon points, that each aircraft will travel until reaches the final destination, a reference lat, lon pair point. Then as input, the encoder receives four entries: a *lat, lon pair* for the reference point, *the number of airplanes* (that will be distributed in a circle), the *initial distance* from the airplanes to the reference point, (i.e. the radius of the circle), and the *number of lat, lon pairs points* that each aircraft will travel until arrive at the reference point. Figure 24 shows the lat, lon pair points generated for a number of 10 "dummy" airplanes, with a reference point at $40.4218, -3.7132$, (near to Madrid), with a trajectory of $10km$ covered by 20 points per aircraft.

Figure 24 – Trajectory generation for 10 airplanes by encoder ADS-B



Source: the author

Using *haversine* formula and bearing angle information, the encoder calculates the lat, lon pair points for the number of aircraft specified and generates the rest of aircraft information fields: altitude, heading, ICAO identifier, vertical rate, speed and *callsign* [2].

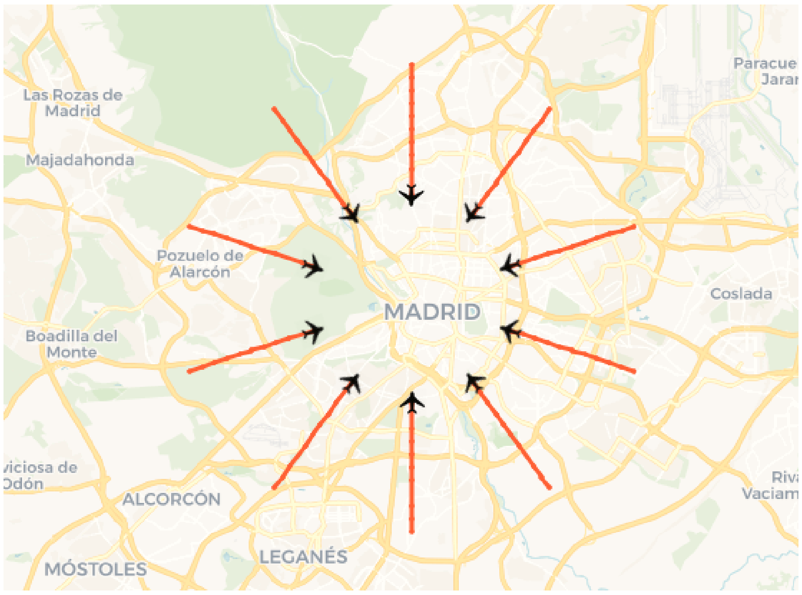
Validating ADS-B data encoder

To evaluate the encoder and also to check the decoding process, a simulation GNU-Radio script shown in Figure 26 was implemented.

The bin file containing 10 dummy aircraft data is loaded in a file source block. The bits of this file are extracted with a sample rate of $2Msamples/sec$ and then unpacket 8 by 8 bits. The following decoding block sequence is the same presented in Section 5.1, Figure 19. Running this GNURadio script and the *tracker* service (with *direct solution*) has produced the following service output, presented in Figure 25. The red trace represents in the map the route path that the aircraft have already covered.

The I/Q data from the binary file is continuously streamed, which gives movement to the dummy airplanes like real airplane flights. By inspecting the visual results, the encoder was considered validated.

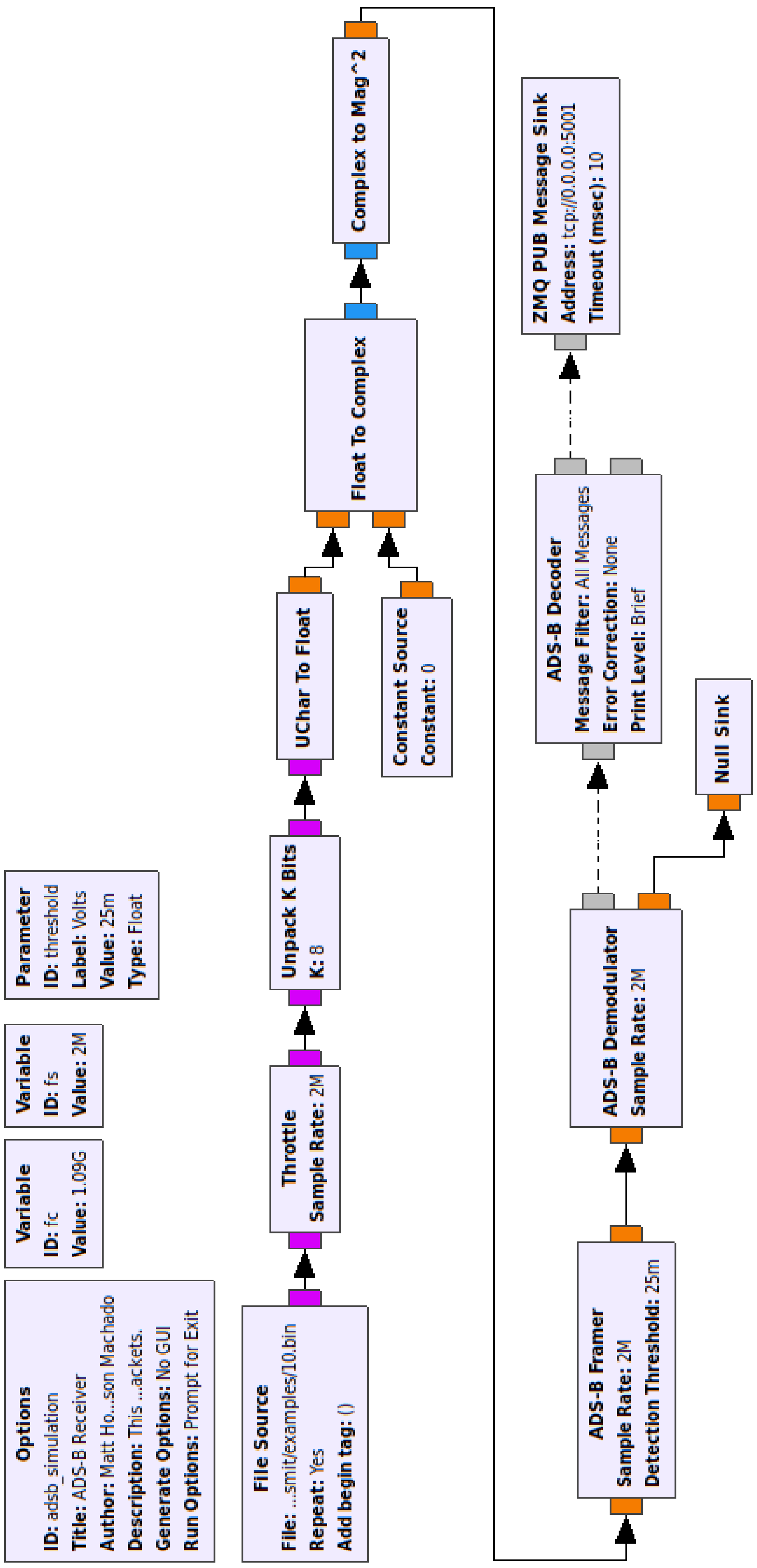
Figure 25 – Tracker service using GNURadio simulation script



Source: the author

[2]a *callsign* is a group of alphanumeric characters that is used to identify an aircraft in air-to-ground communications. Example:*VNN980*, (KUMAR; DEREMER; MARSHALL, 2005).

Figure 26 – GNURadio simulation script



Options
ID: adsb_simulation
Title: ADS-B Receiver
Author: Matt Ho...son Machado
Description: This ...ackets.
Generate Options: No GUI
Run Options: Prompt for Exit

Variable
ID: fc
Value: 1.09G

Variable
ID: fs
Value: 2M

Parameter
ID: threshold
Label: Volts
Value: 25m
Type: Float

File Source
File: ...smit/examples/10.bin
Repeat: Yes
Add begin tag: ()

Throttle
Sample Rate: 2M

Unpack K Bits
K: 8

UChar To Float

Constant Source
Constant: 0

Float To Complex

Complex to Mag^2

ADS-B Framer
Sample Rate: 2M
Detection Threshold: 25m

ADS-B Demodulator
Sample Rate: 2M

Null Sink

ADS-B Decoder
Message Filter: All Messages
Error Correction: None
Print Level: Brief

ZMQ PUB Message Sink
Address: tcp://0.0.0.0:5001
Timeout (msec): 10

Source: the author

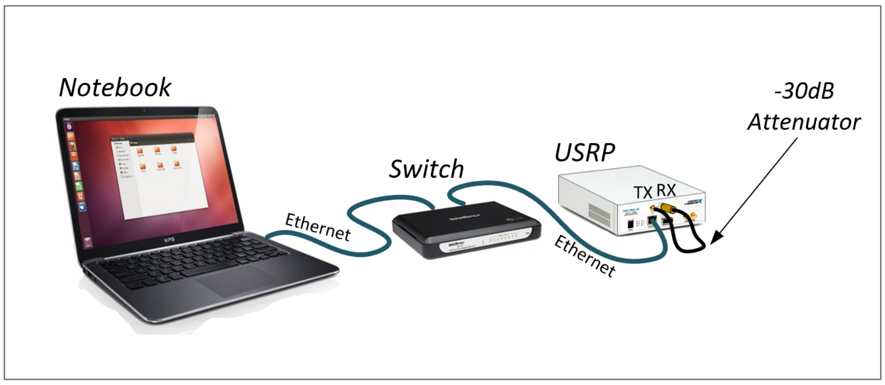
5.3 Transmitting and receiving ADS-B data with USRP

Once the simulation script has validated the ADS-B encoder, the next step concerns the transmission of ADS-B packets with the USRP device using the containers created. It used a channel of USRP for ADS-B transmission and another for reception, both directly connected by an $-30\ dB$ attenuation cable, as shown in 27. The direct connection between the transmitting and receiving channel was choose by the following reasons:

- To avoid ADS-B data scarcity. The ADS-B packets received in a region depends on the airplane traffic of this region.
- To standardize (as well as possible) the communication between the channels, and then be able to perform benchmark tests using the architecture.
- To avoid undesired ADS-B data transmission. Depending on transmitted power, the "dummy" packets could be received by control towers and airplanes in the vicinity, which is clearly not the objective and should be avoided.

Once the USRP 2932 has full duplex capability, for transmit/receive simultaneously, the receiver block presented in Figure 21 was combined with the transmission part of the simulation script shown in Figure 26, resulting in receiver and transmitter stages in a single GNURadio script, as seen in Figure 28.

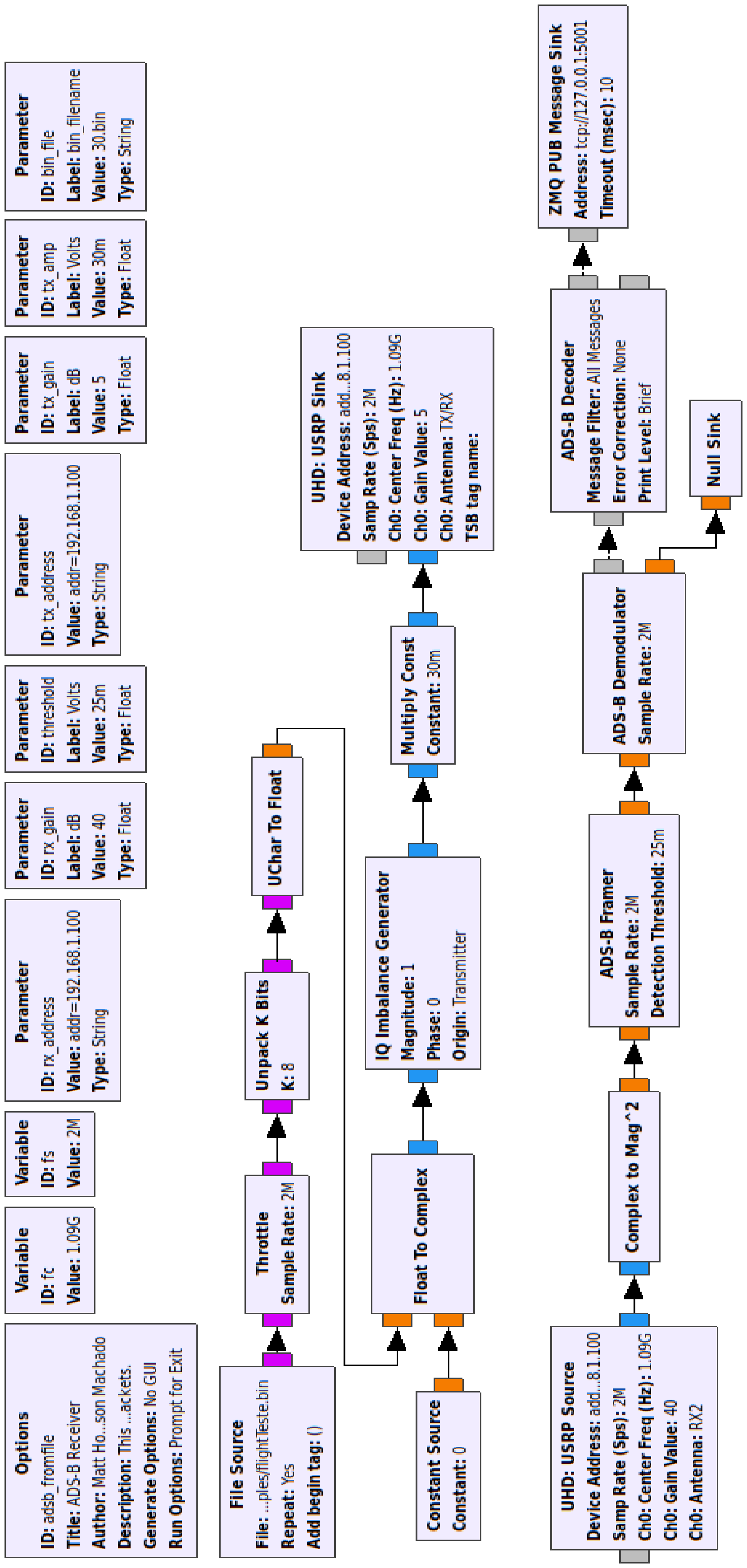
Figure 27 – Experiment setup: transmitting/receiving ADS-B signals



Source: the author

In this case, for the receiver container instead of using the receiver GNURadio script a receiver/transmitter GNURadio script was added, with the possibility to set the transmitter parameters, such as transmission gain and TX Address. The setup for this implementation, which is also used in the architecture evaluation, is shown in Figure 27. It connects a notebook to the USRP 2932 through a switch with gigabit Ethernet capability. The USRP TX channel is connected directly to the RX channel through the attenuation cable 28.

Figure 28 – GNURadio script for ADS-B transmit/receive with an USRP


Options
ID: adsb_fromfile
Title: ADS-B Receiver
Author: Matt Ho...son Machado
Description: This ...ackets.
Generate Options: No GUI
Run Options: Prompt for Exit

Variable
ID: fc
Value: 1.09G

Variable
ID: fs
Value: 2M

Parameter
ID: rx_address
Value: addr=192.168.1.100
Type: String

Parameter
ID: rx_gain
Label: dB
Value: 40
Type: Float

Parameter
ID: threshold
Label: Volts
Value: 25m
Type: Float

Parameter
ID: tx_address
Value: addr=192.168.1.100
Type: String

Parameter
ID: tx_gain
Label: dB
Value: 5
Type: Float

Parameter
ID: tx_amp
Label: Volts
Value: 30m
Type: Float

Parameter
ID: bin_file
Label: bin_filename
Value: 30.bin
Type: String

File Source
File: ...ples/flightTeste.bin
Repeat: Yes
Add begin tag: ()

Throttle
Sample Rate: 2M

Unpack K Bits
K: 8

UChar To Float

Constant Source
Constant: 0

Float To Complex

IQ Imbalance Generator
Magnitude: 1
Phase: 0
Origin: Transmitter

Multiply Const
Constant: 30m

UHD: USRP Sink
Device Address: add...8.1.100
Samp Rate (Sps): 2M
Ch0: Center Freq (Hz): 1.09G
Ch0: Gain Value: 5
Ch0: Antenna: TX/RX
TSB tag name:

UHD: USRP Source
Device Address: add...8.1.100
Samp Rate (Sps): 2M
Ch0: Center Freq (Hz): 1.09G
Ch0: Gain Value: 40
Ch0: Antenna: RX2

Complex to Mag^2

ADS-B Framer
Sample Rate: 2M
Detection Threshold: 25m

ADS-B Demodulator
Sample Rate: 2M

ADS-B Decoder
Message Filter: All Messages
Error Correction: None
Print Level: Brief

Null Sink

ZMQ PUB Message Sink
Address: tcp://127.0.0.1:5001
Timeout (msec): 10

Source: the author

To implement a service from SDR, in practice, the DSP script generally will pass by a manual tuning stage, where the adjustment of receiver main parameters, such as gain, signal threshold, SDR channel address, take place. It can be achieved by adding two DSP scripts in the receiver container, one for tuning the parameters with a GUI mode, and another, which will run a final script with fine-tuned parameter values. The choice of which script to run can be made using main container command substitution at runtime.

After the parameter setting procedure, the final values are shown in the Table 3.

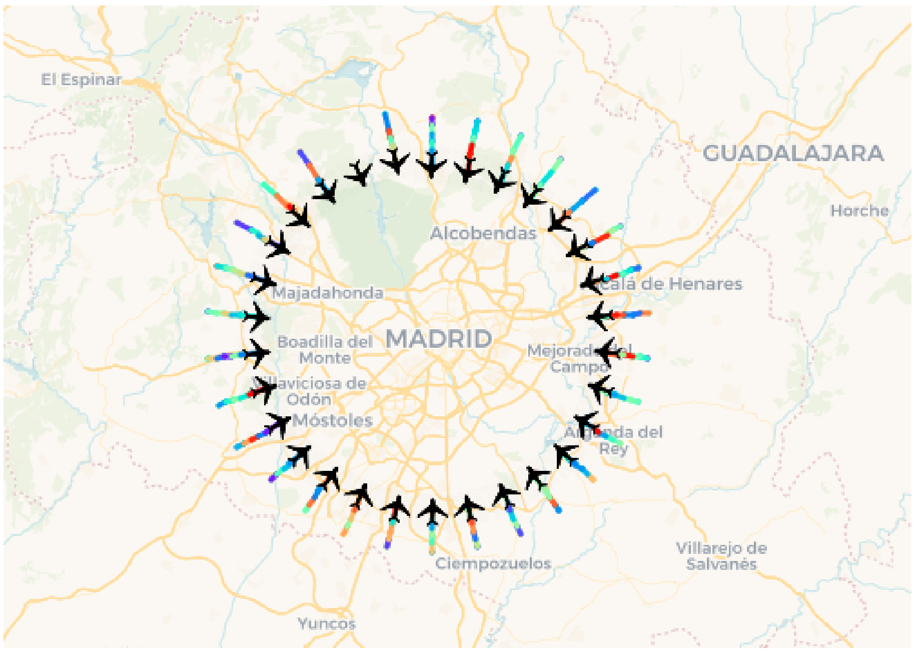
Table 3 – SDR configuration parameters TX and RX

Receiver		**Transmitter**	
rx-address	198.162.1.100	tx-address	198.162.1.100
rx-gain (dB)	40	tx-gain (dB)	5
threshold (V)	0.025	tx-amp (numerical)*	0.030

* tx-amp is a numerical value used to fine-tunning TX power.

Running the *receiver* (now with also transmitting capability), *tracker* and *altitude* containers, with the setup presented in Table 3, has produced the *tracker* and *altitude* services, shown in Figures 29. In this verification, the ADS-B data are created from a bin file generating 30 dummy aircraft. Those aircraft are moving towards the reference point at $lat, lon = 40.4218, -3.7132$.

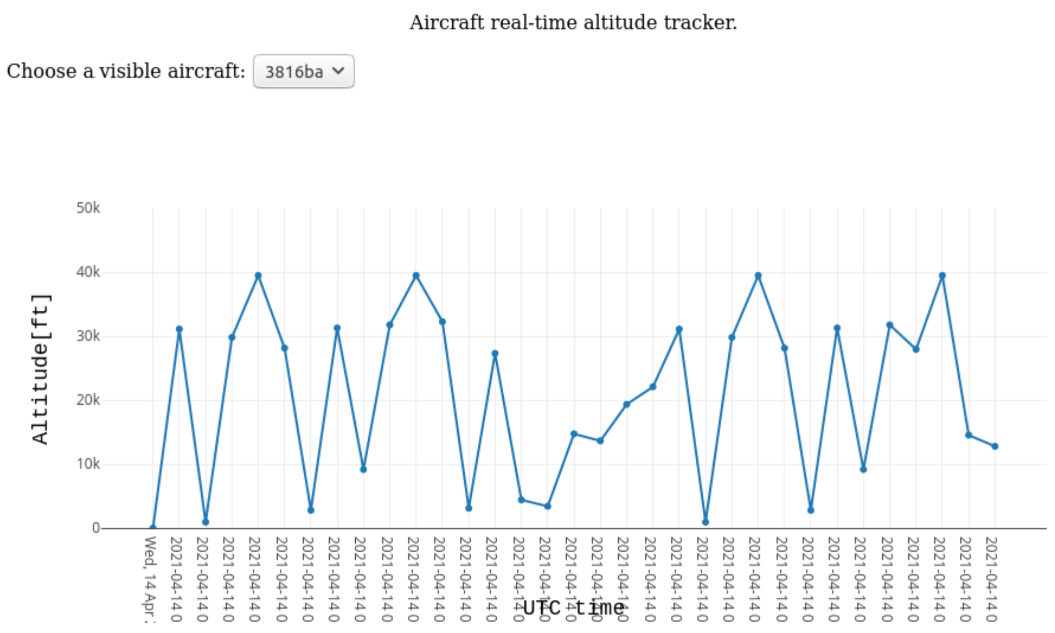
Figure 29 – Aircraft tracker service generated at port 5002



Source: the author

Once the ADS-B encoder creates random altitudes for the aircraft, their paths do not follow a real trajectory, as seen in the Figure 30, for *altitude* service of aircraft identified by ICAO code $3816ba$,

Figure 30 – Aircraft altitude service generated at port 5003



Source: the author

These results show that, alternatively to the *direct solution*, the container-based solution is a viable option to generate end-to-end services from the SDR. Once it was implemented by CLI commands, the main difference with the architecture implementation consist in automation. The architecture automatically orchestrates the service generation and verification, for a client request. The next section will present two networking modes implemented by the architecture and the services verifiers.

5.4 Creating architecture networking modes and service verifiers

Once the receiver and application containers for ADS-B end-to-end services generation were already developed and checked out, two network mode options from the architecture were created. In addition, to guaranteeing the service provision to the end user, service verifiers were created.

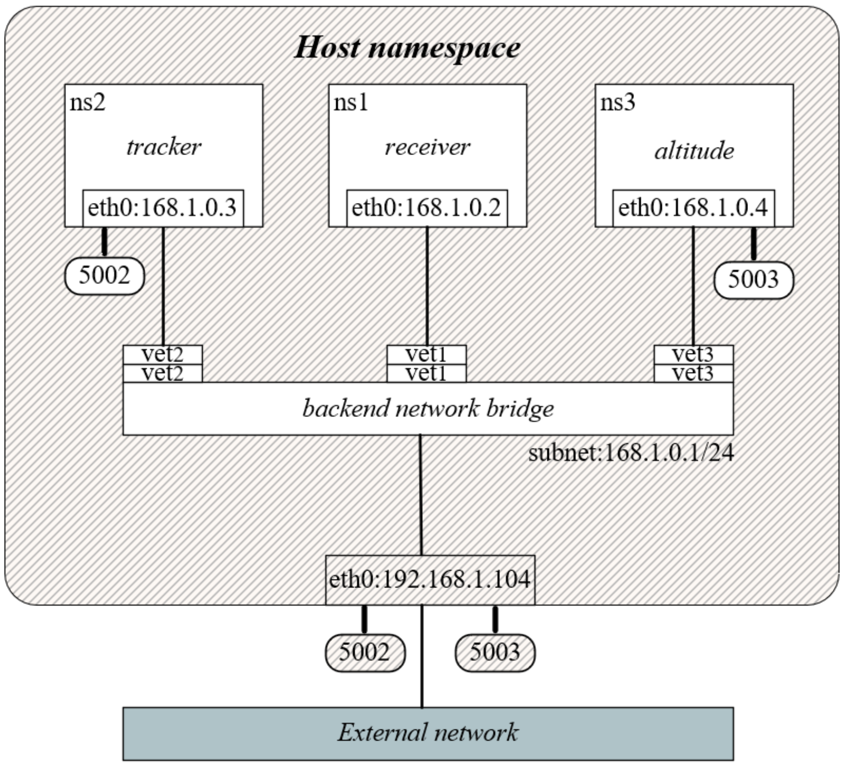
Creating networking modes

These modes are selected through run templates (at *Container Run Template* block), the choice of which template will be used for generating a service is taken in the *Container*

Manager, in Figure 15, according to the service and network isolation required. These architecture modes are explained below:

- *Architecture host*: in this mode, the containers share the same network namespace from the host machine without any network isolation. The APPENDIX A presents a docker-compose template that runs the architecture in this mode. Application containers were developed with a variable called *HOST_MODE*, that when set to *true* will specify this mode. Output ports can be also specified. Default values are *tracker* service output in host port 5002, and *altitude* service output in host port 5003.
- *Architecture net*: this mode creates a custom bridge network called *backend* for the container communication. A docker-compose template used to run this mode is presented in APPENDIX B. Each container has its own network namespace, $ns1$, $ns2$ and $ns3$, isolated from the host namespace, as seen in Figure 31. The subnet *backend* bridge network has the address 168.1.0.1/24, created with default docker driver. The containers have their IP assigned to this network and are connected through veth pairs, being visible and able to communicate to each other. The *backend* network bridge is then connected to the host interface (eth0:192.168.1.104). Tracker container exposes the service output at its port 5002, and *altitude* container exposes the service output at its port 5003. Then the *tracker* and *altitude* services are published at host port 5002 and 5003, respectively, where the client can access the service. Both containers output ports and host publish ports can be configured in the template files.

Figure 31 – Architecture net mode namespace details



Source: the author

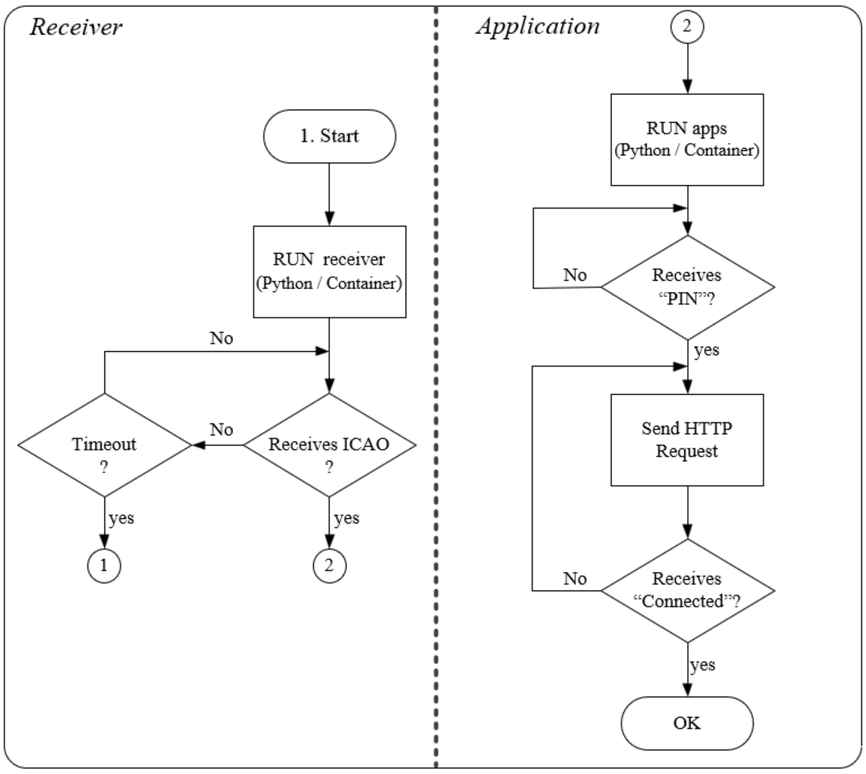
Creating service verifiers

When a container runs, there is no guarantee that its function is actually being executed as expected. For instance, when the *receiver* container is launched sometimes internal USRP errors occur and its expected output, (decoded ADS-B data), is not produced. To assure that a service is effectively working, and also to standardize the startup test (discussed in Section 7.1), some service verifiers were created. The verifiers are managed by the *Container manager* block, and can be applied individually, (for instance to check if just the receiver is producing its output as expected), or for the complete container implementation, i.e. receiver plus application containers.

Possibly, there is no consensus on which point defines that a service was correctly created, but some definitions could be made. For ADS-B service generation, it was defined that the *receiver* container is executing its function when an "ICAO" string is printed at the container terminal. It means that the first decode packet was received. For the service containers, it was considered that the service was launched correctly when the application containers (HTTP servers) answered a connection HTTP request.

Both receiver and application containers, as well as the complete end-to-end service, (*receiver* plus *application* containers), verifiers were implemented through bash files by the *Container Manager* block, following the flowchart presented in Figure 32.

Figure 32 – Flowchart for receiver/application verifiers

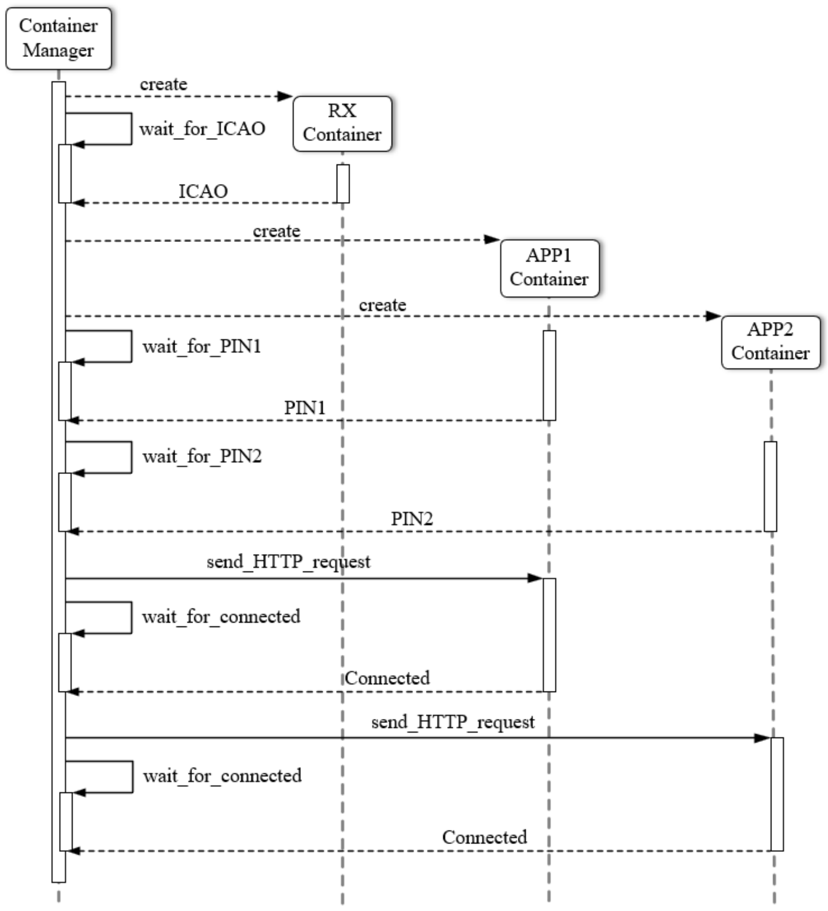


Source: the author

Consider the case that the complete solution (*receiver* plus *application*) will be verified. It starts by the receiver verifier flowgraph, the first step consists in running the python script (*direct solution*) or the receiver container (in host or network modes). Then a message containing "ICAO” is expected at the terminal (host machine terminal or container terminal). If any "ICAO" message is received and a timeout is reached, then the python script or container is running again. It’s also possible to set a maximum number of attempts. If the "ICAO" code is received, then the application flowchart starts. In the first step the *tracker* and *altitude* python scripts or container will be running, until a "PIN" message, (a server debug terminal output), which indicates that the server was built, is received. Then the *Container Manager block* will send HTTP connection requests to the services output ports (default values 5002 and 5003) and wait until a "Client Connected message" from both services is received. At this point, the complete solution is considered checked out.

The interactions between the *Container Manager* block, receiver and application containers or scripts, with service verifiers are shown in the diagram presented in Figure 33, considering the steps from container implementation (step 4) onward.

Figure 33 – Block interaction with service verifiers

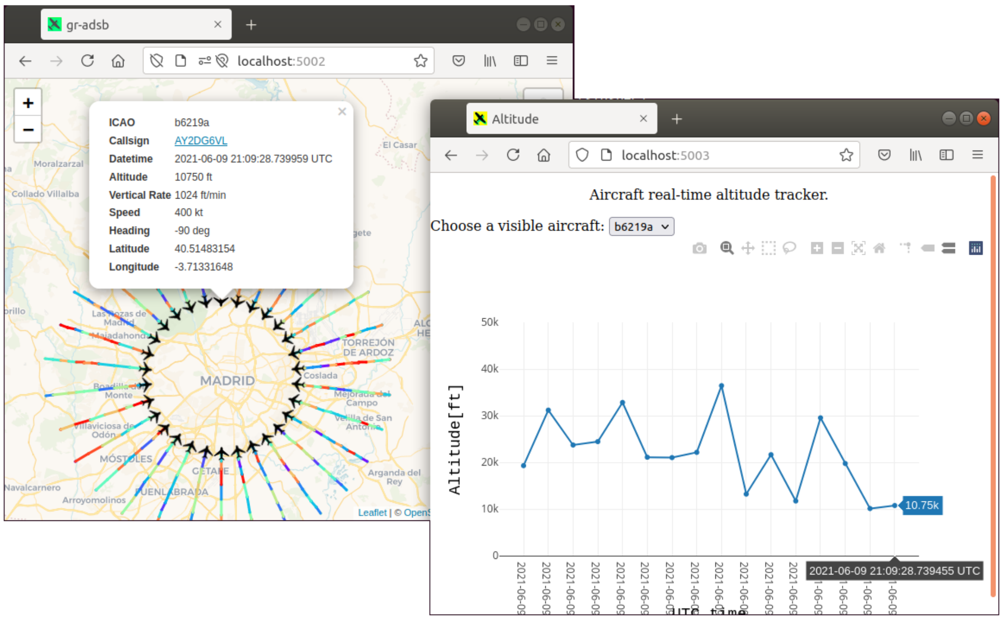


Source: the author

5.5 Validating the architecture

Finally, after proceeding with the implementation steps, the architecture implementation for ADS-B service is validated over *tracker* and *altitude* services through visual inspection, as seen in figures 34 and 35, for *architecture host* and *architecture net* modes, respectively.

Figure 34 – *Architecture host* mode validation



gr-adsb

localhost:5002

ICAO	b6219a
Callsign	AY2DG6VL
Datetime	2021-06-09 21:09:28.739959 UTC
Altitude	10750 ft
Vertical Rate	1024 ft/min
Speed	400 kt
Heading	-90 deg
Latitude	40.51483154
Longitude	-3.71331648

Altitude

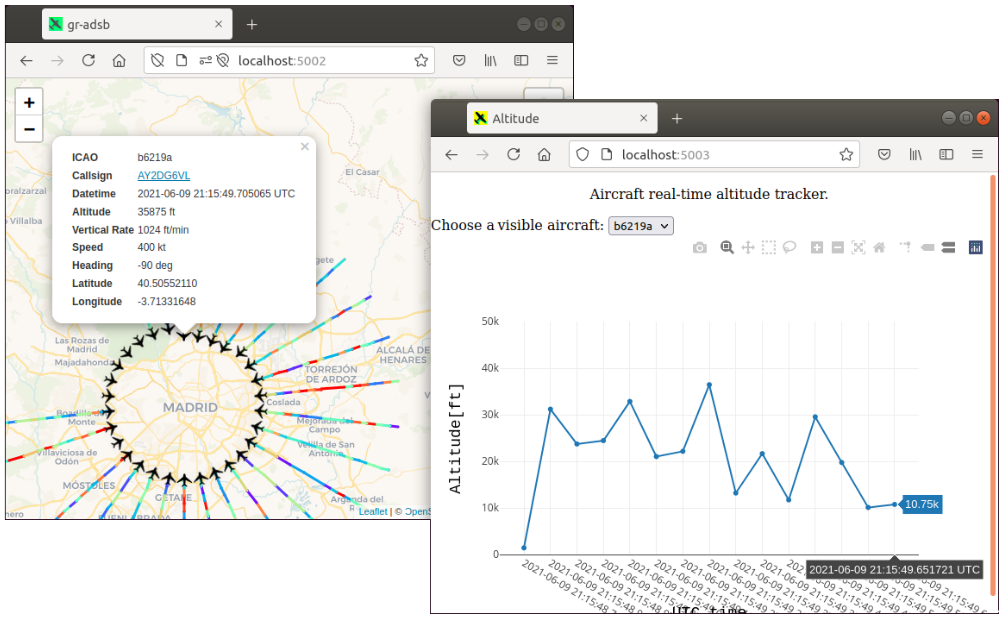
localhost:5003

Aircraft real-time altitude tracker.

Choose a visible aircraft: b6219a

Source: the author

Figure 35 – *Architecture net* mode validation



gr-adsb

localhost:5002

ICAO	b6219a
Callsign	AY2DG6VL
Datetime	2021-06-09 21:15:49.705065 UTC
Altitude	35875 ft
Vertical Rate	1024 ft/min
Speed	400 kt
Heading	-90 deg
Latitude	40.50552110
Longitude	-3.71331648

Altitude

localhost:5003

Aircraft real-time altitude tracker.

Choose a visible aircraft: b6219a

Source: the author

6 EXPLORING ARCHITECTURE FEATURES

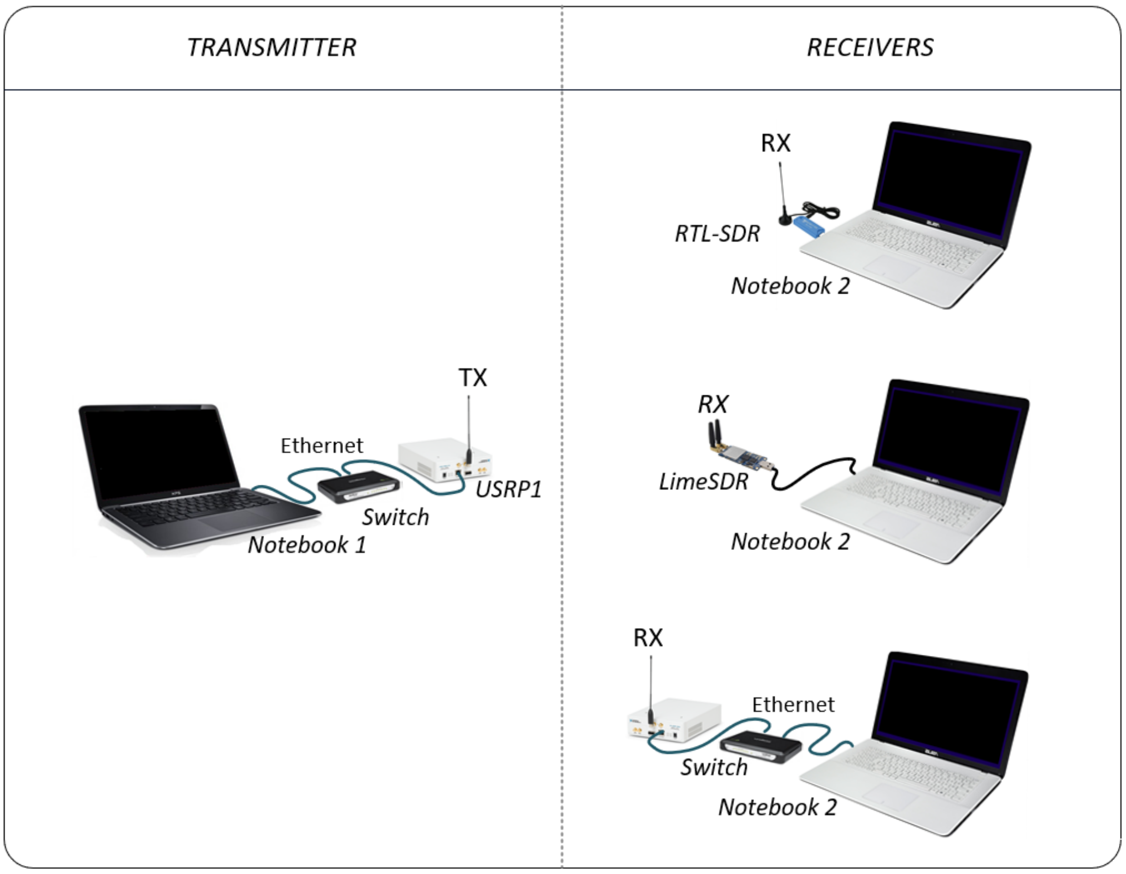
Once the architecture is validated for ADS-B use case, in this chapter some features resulting from the architecture implementation are highlighted. In Section 6.1 the implementation of the architecture with a different radio project is explored, with an example of service using LoRa modulation technique. The second feature demonstrates container reusability and the architecture utilization with different SDR devices types. Finally, scalability of the services are tested and demonstrated in Section 6.3.

6.1 Implementing LoRa service

To demonstrate architecture flexibility to provide different services, the LoRa modulation technique was also implemented. The service consists in data reception of text files, using gr-LoRa GNURadio OOT module (KNIGHT, 2017). This module and the GNURadio DSP script are added to the top layer of a generic RX container. The RX container as well as its run template are then stored in the *Image* and *Container Registry* blocks, (shown in Figure 15), respectively. Finally, by the same token as ADS-B service provision, the *Container Manager* block orchestrates the service implementation.

LoRa transmitter consists of an USRP 2932. For reception, LimeSDR, RTL-SDR and other USRP 2932 SDRs were used. The setup for the three communication cases are presented in Figure 36.

Figure 36 – Communication setup for LoRa/ADS-B services

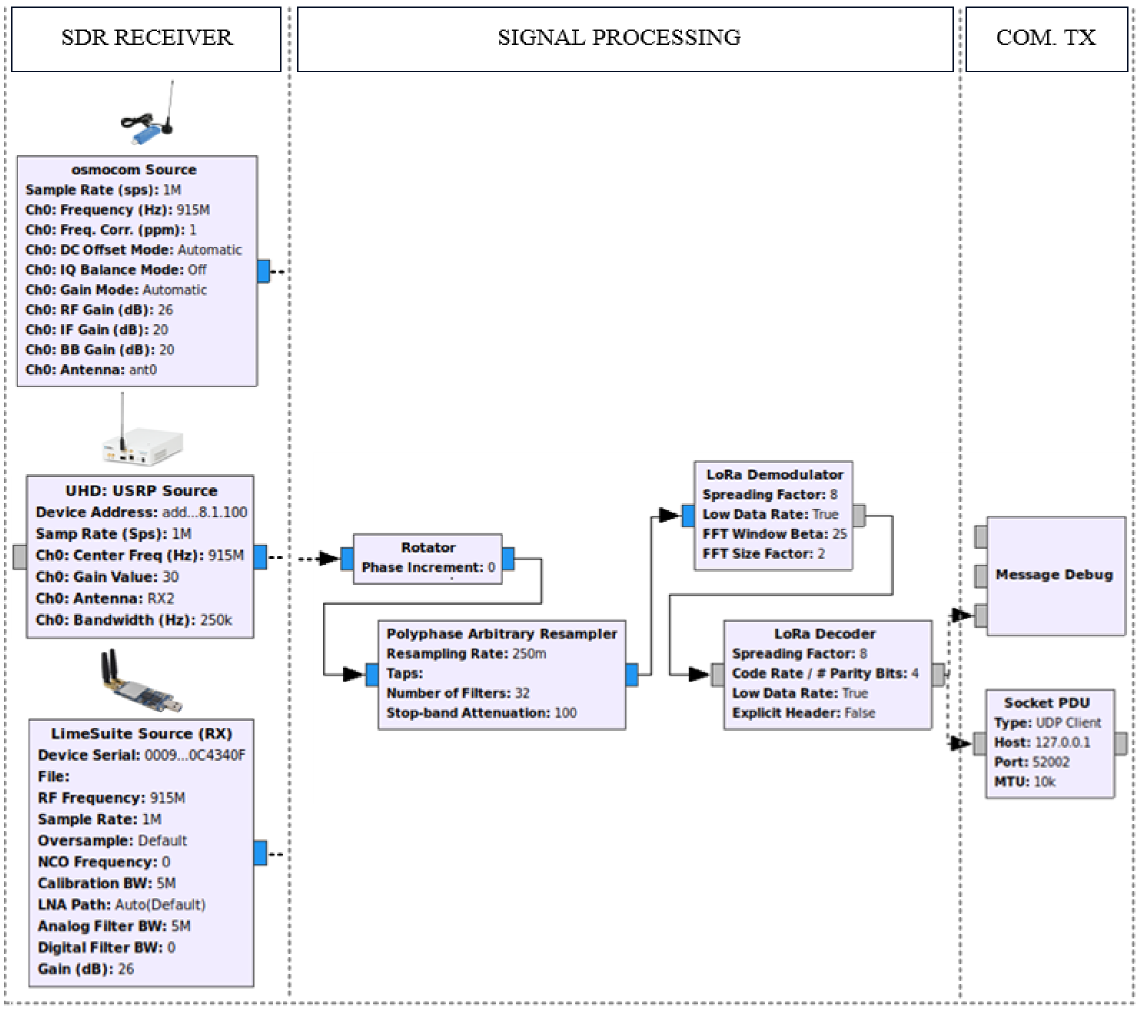


Source: the author

The main blocks of GNURadio DSP script used for the three communication cases are presented in Figure 37. It is possible to receive the data with different SDR devices, changing the SDR receiver block and adjusting its communication parameters. The docker template used for LoRa reception with LimeSDR is presented in APPENDIX C.

Running the service, a *helloword.txt* text file containing two lines with the message "hello world", as well as computer arts text files, *dog.txt* and *slug.txt*, are sent by the LoRa transmitter to the RTL-SDR, LimeSDR and USRP receivers, respectively. Using the *architecture host* mode the data is recovered on the receiver side, (notebook 2), at port 52002, through a *Netcat* Linux command. The files received are presented in Figure 38.

Figure 37 – GNURadio DSP script implemented with SDR receivers



osmocom Source
Sample Rate (sps): 1M
Ch0: Frequency (Hz): 915M
Ch0: Freq. Corr. (ppm): 1
Ch0: DC Offset Mode: Automatic
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Automatic
Ch0: RF Gain (dB): 26
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20
Ch0: Antenna: ant0

UHD: USRP Source
Device Address: add...8.1.100
Samp Rate (Sps): 1M
Ch0: Center Freq (Hz): 915M
Ch0: Gain Value: 30
Ch0: Antenna: RX2
Ch0: Bandwidth (Hz): 250k

LimeSuite Source (RX)
Device Serial: 0009...0C4340F
File:
RF Frequency: 915M
Sample Rate: 1M
Oversample: Default
NCO Frequency: 0
Calibration BW: 5M
LNA Path: Auto(Default)
Analog Filter BW: 5M
Digital Filter BW: 0
Gain (dB): 26

Rotator
Phase Increment: 0

Polyphase Arbitrary Resampler
Resampling Rate: 250m
Taps:
Number of Filters: 32
Stop-band Attenuation: 100

LoRa Demodulator
Spreading Factor: 8
Low Data Rate: True
FFT Window Beta: 25
FFT Size Factor: 2

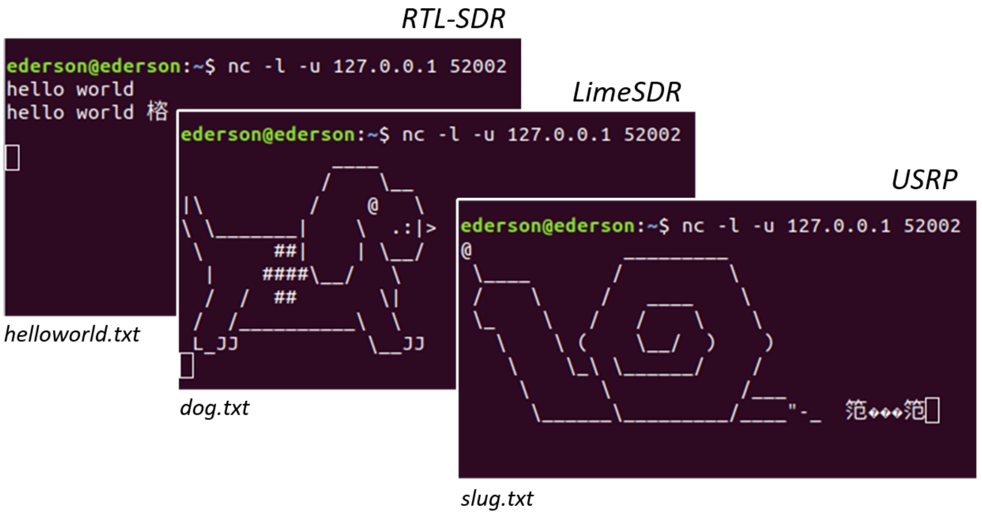
LoRa Decoder
Spreading Factor: 8
Code Rate / # Parity Bits: 4
Low Data Rate: True
Explicit Header: False

Message Debug

Socket PDU
Type: UDP Client
Host: 127.0.0.1
Port: 52002
MTU: 10k

Source: the author

Figure 38 – Text files received with LoRa communication



```
ederson@ederson:~$ nc -l -u 127.0.0.1 52002
hello world
hello world 榕
```

```
ederson@ederson:~$ nc -l -u 127.0.0.1 52002
             ____
            /    \__
|\         /    @   \
\ \_______|     \  .:|>
 \      ##|     | \__/
  |    ####\__/    \
  /  /  ##         \|
 /  /_____________\  \
 L_JJ               \__JJ
```

```
ederson@ederson:~$ nc -l -u 127.0.0.1 52002
@             _________
 \____       /         \
 /    \     /   ____    \
 \_    \   /   /    \    \
   \    \ (    \__/  )    )
    \    \_\ \________/  /
     \      \           /____
      \______\_________/____"-_  筢���筢
```

Source: the author

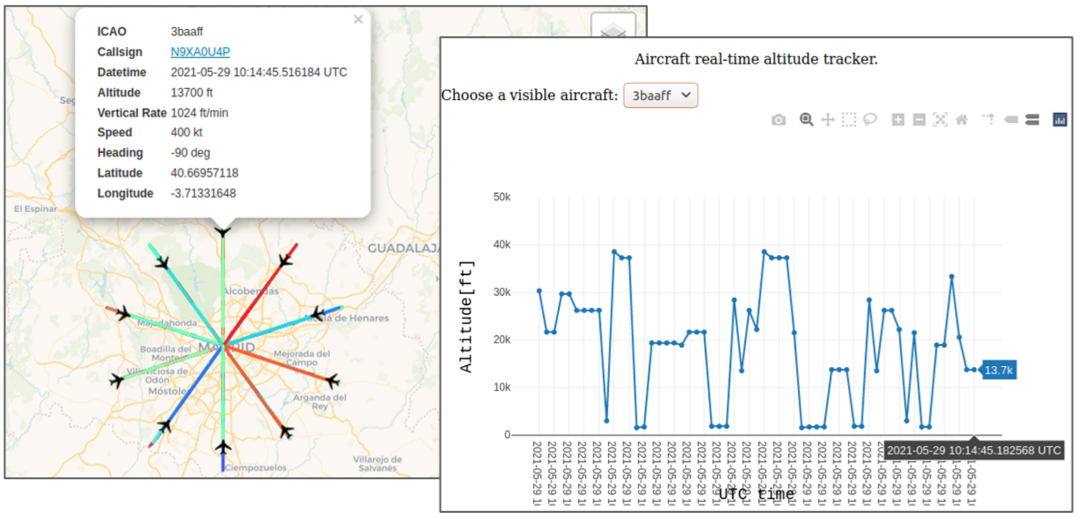
The results show that the architecture can be applied to different radio projects through the selection of the DSP reception script used by the receiver container. In this example, the LoRa modulation technique was implemented and the communication was verified for the aforementioned SDR devices.

6.2 ADS-B services with different SDR devices

Architecture flexibility for a variety of SDR devices can also be demonstrated to the ADS-B services. This test uses the same LoRa service setup presented in Figure 36, to generate ADS-B *tracker* and *altitude* services from RTL-SDR, LimeSDR and USRP SDRs. In the signal reception, the GNURadio receiver script presented in Figure 21, has been applied, changing the SDR receiver block and adjusting its respective communication parameters. The *architecture net* is applied and the receiver and application containers launched to generate the service. Then the final visual results obtained (*tracker* and *altitude* services), are shown in figures 39, 40 and 41, for RTL-SDR, LimeSDR and USRP reception cases. APPENDIX D presents the docker template file used to launch ADS-B tracker and altitude services with RTL-SDR.

The figures 39, 40 and 41, evince the *tracker* and *altitude* services for an aircraft identified as ICAO *3baaff*, at a given instant of time.

Figure 39 – ADS-B *tracker* and *altitude* services with RTL-SDR



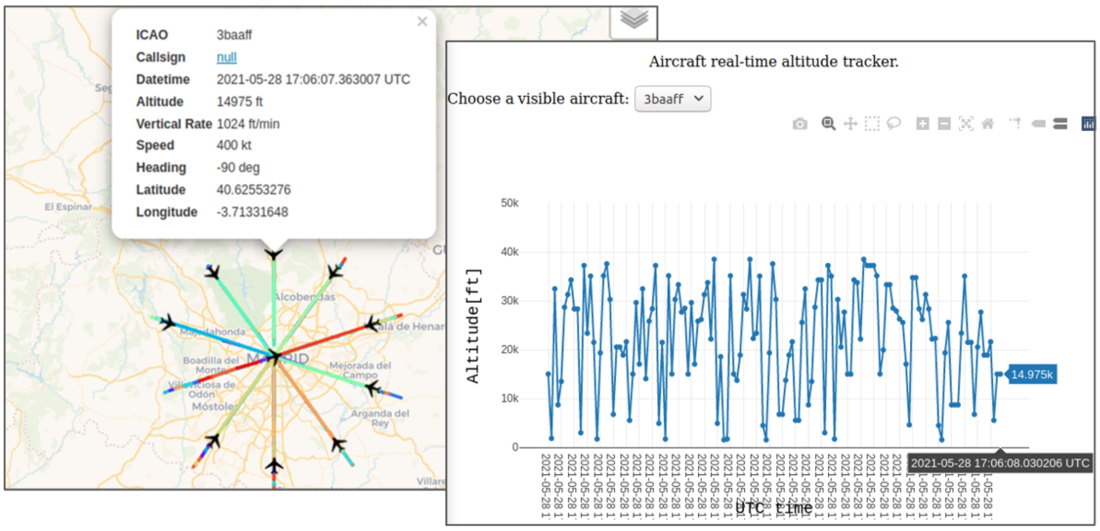
ICAO	3baaff
Callsign	N9XA0U4P
Datetime	2021-05-29 10:14:45.516184 UTC
Altitude	13700 ft
Vertical Rate	1024 ft/min
Speed	400 kt
Heading	-90 deg
Latitude	40.66957118
Longitude	-3.71331648

Aircraft real-time altitude tracker.

Choose a visible aircraft: 3baaff

Source: the author

Figure 40 – ADS-B *tracker* and *altitude* services with LimeSDR



ICAO	3baaff
Callsign	null
Datetime	2021-05-28 17:06:07.363007 UTC
Altitude	14975 ft
Vertical Rate	1024 ft/min
Speed	400 kt
Heading	-90 deg
Latitude	40.62553276
Longitude	-3.71331648

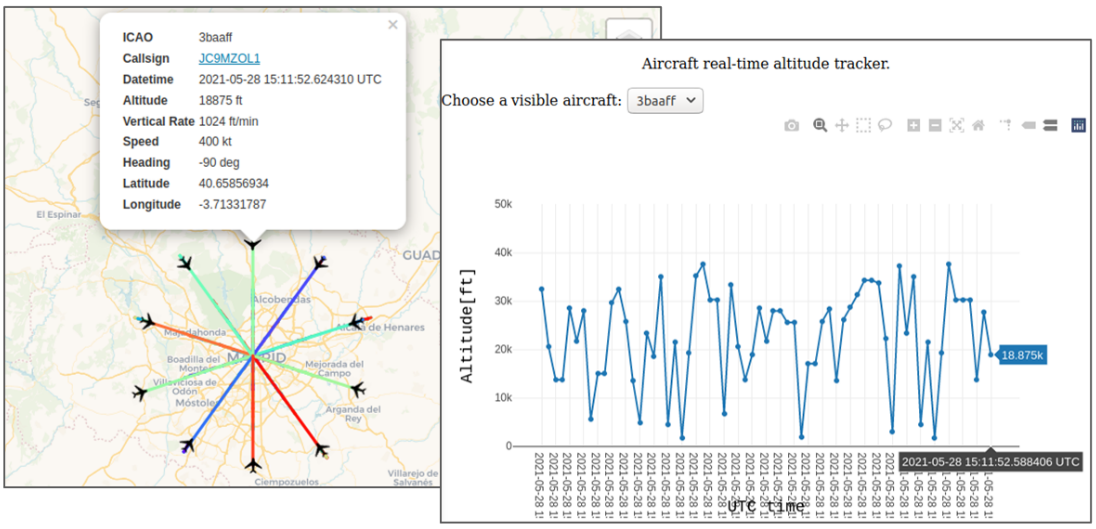
Aircraft real-time altitude tracker.

Choose a visible aircraft: 3baaff

Source: the author

The results show that the proposed architecture achieves modularity of communication blocks, by decoupling receiver and application services into independent containers, and SDR flexibility, once with few changes in the reception script, it is possible to apply the architecture to provide services from different SDR devices.

Figure 41 – ADS-B *tracker* and *altitude* services with USRP



ICAO	3baaff
Callsign	JC9MZOL1
Datetime	2021-05-28 15:11:52.624310 UTC
Altitude	18875 ft
Vertical Rate	1024 ft/min
Speed	400 kt
Heading	-90 deg
Latitude	40.65856934
Longitude	-3.71331787

Aircraft real-time altitude tracker.

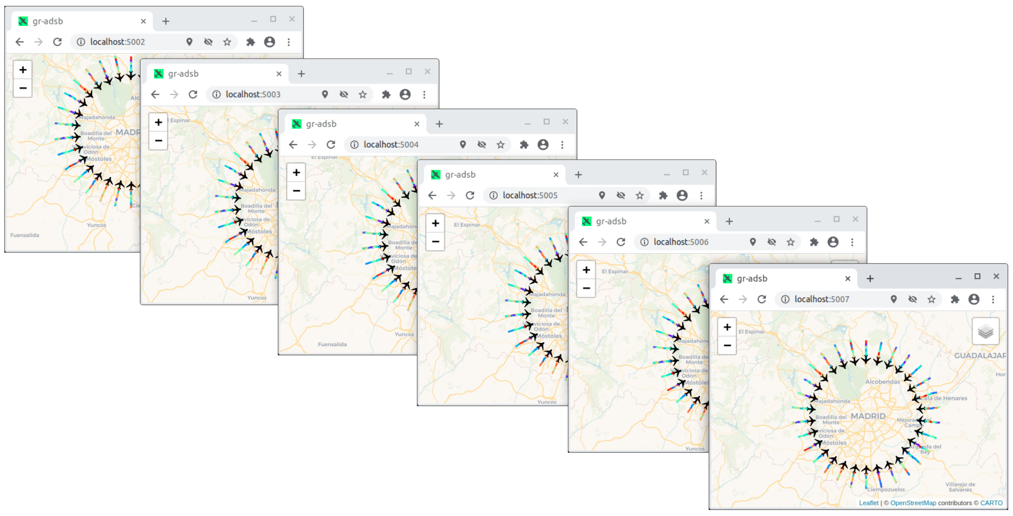
Choose a visible aircraft: 3baaff

Source: the author

6.3 Reproducing the services

One of the benefits brought by the implementation of containers with the architecture is the ease of reproducing the services. In order to check this feature, a simple test was implemented. This test evaluates the free RAM memory for an increasing number of services launched. An illustration of service reproducing with output ports 5002 to 5007 is shown if Figure 42.

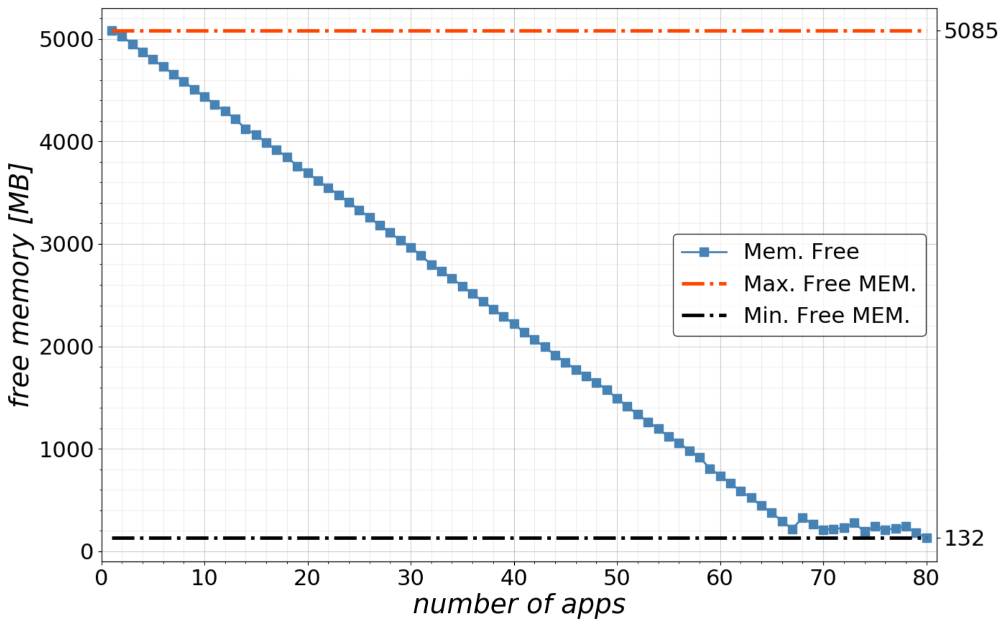
Figure 42 – Tracker reproducing process output example



Source: the author

Using the USRP with attenuation cable between TX and RX, the setup presented in Figure 27, A notebook with 7.45 GB of RAM, the receiver container was launched in *architecture net* mode, and the *tracker* service (HTTP server) was reproduced until the system reaches its limit, (in terms of free RAM memory). The results for this evaluation is presented in Figure 43, where free RAM memory is measured in Megabytes (MB).

Figure 43 – Free RAM memory x number of *tracker* services



Source: the author

The system reaches its limit with 80 service containers, starting from $5085MB$ of free memory with 1 *tracker* service, and arriving at $132MB$ with 80 simultaneous *tracker* services. In the linear area of the graph (1-67 services), the decreasing rate of free memory has the mean of $-73.74MB$ per tracker service. It was applied with a bash script that can be implemented and automated by the *Container Manager*. Other implementations options are also possible, such as *docker run* command, docker-compose files, as well as containers orchestration tools (e.g. docker Swarm and Kubernets). It is noted that the system resources bound the number of services that can be generated. This evaluation helps to clarify how it is possible to take advantage of the ease of reproducing services using containers.

7 EVALUATION

The main objective of the evaluation is to compare the performance of the proposed solution using the containerized architecture with the *direct solution* in end-to-end service generation from SDR devices. Therefore, the overhead in terms of resource utilization, start up time, and server response time, is verified. For this purpose, the aforementioned three modes of use: 1. *direct solution*; 2. with the container in *host networking* mode - *architecture host*; and 3. with the containers in an internal custom bridge network - *architecture net*, are evaluated and compared for the ADS-B use case. For this investigation (all tests), the setup presented in Section 5.3, shown in Figure 27, was applied. This setup considers the USRP as an SDR device with a transmission channel connected directly to the receiver channel, by an $-30dB$ attenuation cable. The following tests were evaluated:

- *Startup time test*: compares the startup time to launching the services, or individual containers/scripts.
- *Resource utilization test*: this test compares the CPU and RAM memory utilization to provide the services under stressed conditions (increasing the number of HTTP client requests).
- *Response time test*: this test compares the waiting time request response for the services, under an HTTP request overflow.

Following, the sections 7.1, 7.2 and 7.3 present the test results and discussion.

7.1 Startup time test

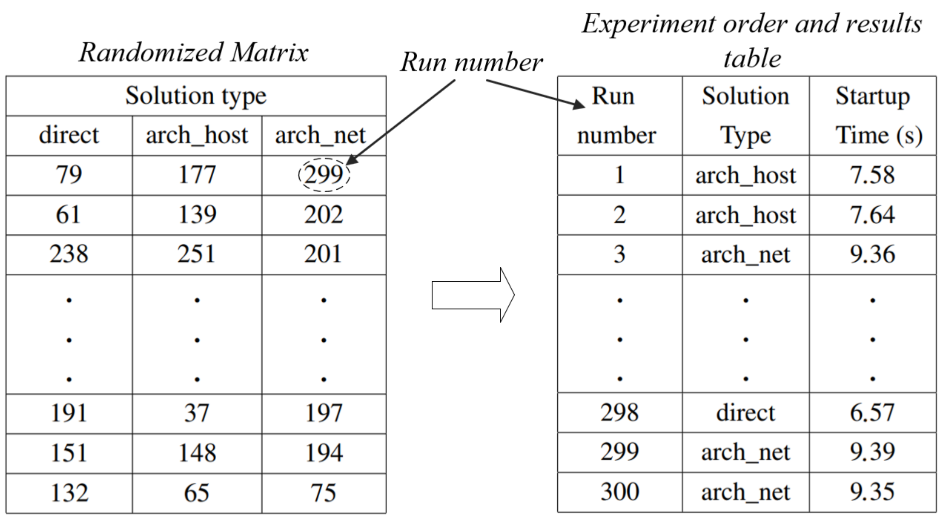
The *startup time test* evaluates the time in seconds, necessary to run individually a container/python script, or the complete sequence of containers that generate the ADS-B end-to-end service.

Statistical design of the experiment

Following guidelines to designing experiments (MONTGOMERY, 2013), the Analysis of variance (ANOVA) method is initially applied to check out, by a hypothesis test, if the startup time means of the three groups of solution, (*direct*, *architecture host* and *architecture net*) differ. Once this first study is conducted, respecting the ANOVA method premises, more specific parametric mean tests, (such as Tukey, Fisher, etc.), can be applied to group and evaluate the mean confidence intervals (CI) of the three solutions.

To apply the ANOVA method, the randomization principle (MONTGOMERY, 2013) was adopted, thus a complete randomized matrix was generated determining the repetition test order, an example of randomized matrix is presented in Figure 44. The *startup time test* is characterized as single-factor experiment (factor=solution type), with 3 levels (or treatments: *direct*, *architecture host* and *architecture net*), being the response variable the *startup time*, measured in seconds. It was firstly defined to investigate the results for a number of observations per level $N = 100$, (i.e. balanced data).

Figure 44 – Experimental Randomized matrix and order table



Randomized Matrix

Solution type		
direct	arch_host	arch_net
79	177	299
61	139	202
238	251	201
. . .	. . .	. . .
191	37	197
151	148	194
132	65	75

Experiment order and results table

Run number	Solution Type	Startup Time (s)
1	arch_host	7.58
2	arch_host	7.64
3	arch_net	9.36
. . .	. . .	. . .
298	direct	6.57
299	arch_net	9.39
300	arch_net	9.35

Source: the author

The ANOVA power and sample size analysis for the *receiver startup time test*, has shown that with $N = 100$, the statistical power[1] is 85% to detect a mean difference of $0.274s$ at treatment levels. This difference is considered appropriate because the minimum mean difference between treatment levels observed in the *startup time tests* were greater than $1s$. Therefore, the following hypotheses were checked: $H_0 : \mu_1 = \mu_2 = \mu_3$ no differences in treatment means, against the alternative H_1 : some means are different. For the *receiver*, the results of the ANOVA procedure, with $P - Value$ approach, and 95 percent confidence interval, (significance $\alpha = 0.05$), are summarized in Table 4.

[1]The statistic power of a hypothesis test is a very important measure, it relies on the probability of not making a type II error, i.e. when a false hypothesis is not rejected being it false.

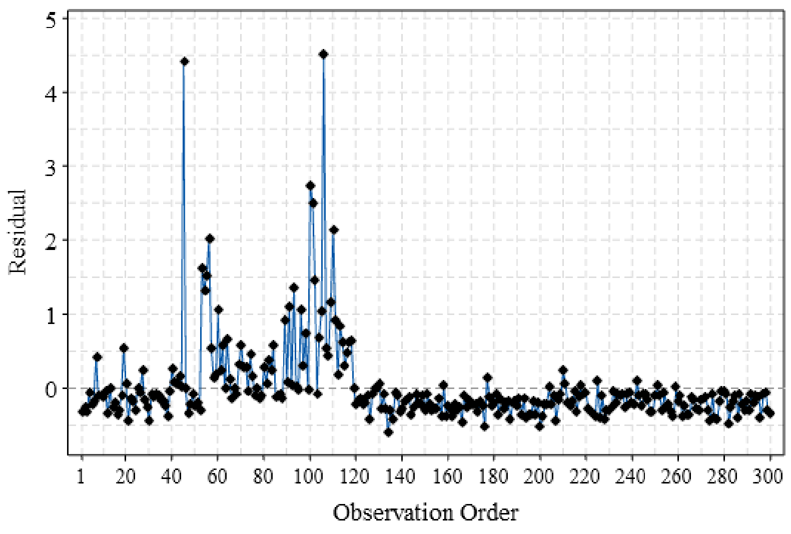
Table 4 – ANOVA for the *receiver startup time test*

Source of variation	**Sum of Squares**	**Degrees of Freedom**	**Mean Square**	**F_0**	***P-Value***
Solution Type	471.1	2	235.55	691.25	0.00
Error	101.2	297	0.34		
Total	572.3	299			

Where F_0 is the Solution Type mean square divided by the *Error* mean square. The $P - Value = 0.00$, ($P - Value < 0.05$), tells that F_0 value is significant, in this case H_0 is rejected, and then it is concluded that the treatment means differ, that is, the solution type significantly affects the mean startup time.

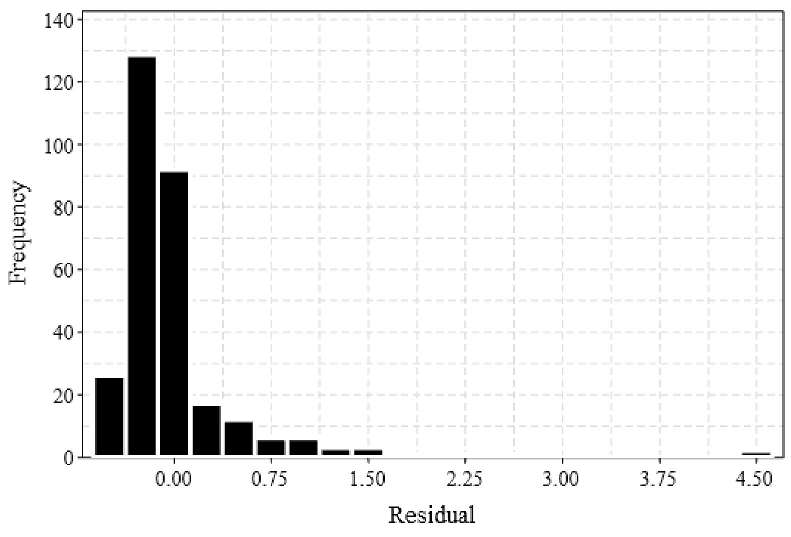
However, to assure that the ANOVA test is applicable, in spite of the primary results, the procedure premises should be verified. The ANOVA procedure considers that the model errors are independently and normally distributed random variables with mean zero and variance σ^2. Then the variance σ^2 is assumed to be constant for all treatments of the factor. To check these premises, and validate the ANOVA model performed, the residual error can be evaluated. A residual, performed by Minitab© software, is the difference between an observed value and its corresponding fitted value. In turn, the fitted values for the model, (e.g. represented by the red straight line in Figure 47), are calculated using the regression equation and variable settings. The error independence was checked by residual versus order plot, shown in Figure 45. The random behavior of residual error indicates that there is no evidence that this premise was violated. In contrast, the histogram of residuals, presented in Figure 46, infers that the shape of residuals do not follow a normal distribution. To verify if the residual error distribution is normally shaped a Ryan-Joiner test was conducted, resulting in violation ($P - Value < 0.05$, i.e. reject H_0, the data do not follow a normal distribution), as seen in Figure 47. The treatment constant variance premise was also checked by Levene's Test, (resulting in $P - Value = 0.021$), also indicating a premise violation. Thus, it is concluded that the ANOVA procedure is not applicable for the data analysis, once its premises were not respected.

Figure 45 – Residual error x observation order for *receiver*



Source: the author

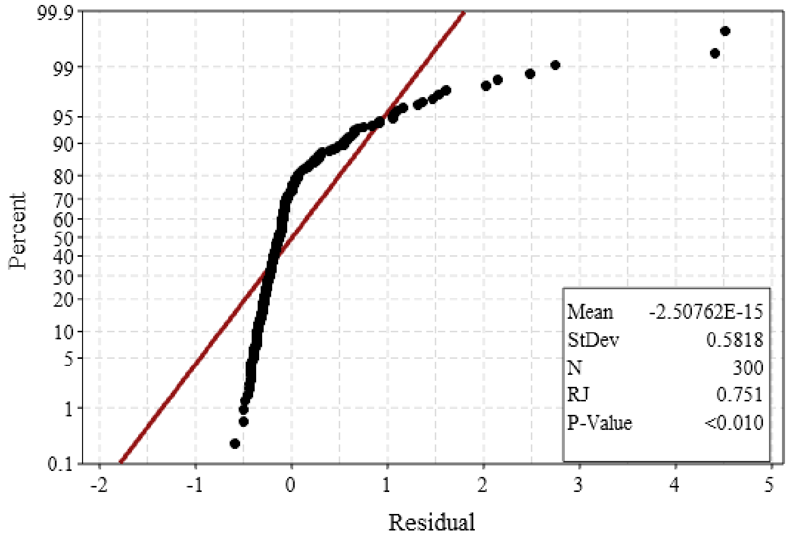
Figure 46 – Histogram of residuals for *receiver startup time test*



Source: the author

It is evidenced in the normality probability plot of residual, Ryan-join test, Figure 47, that the model does not fit the data around the red straight line as expected when the residual follows a normal distribution.

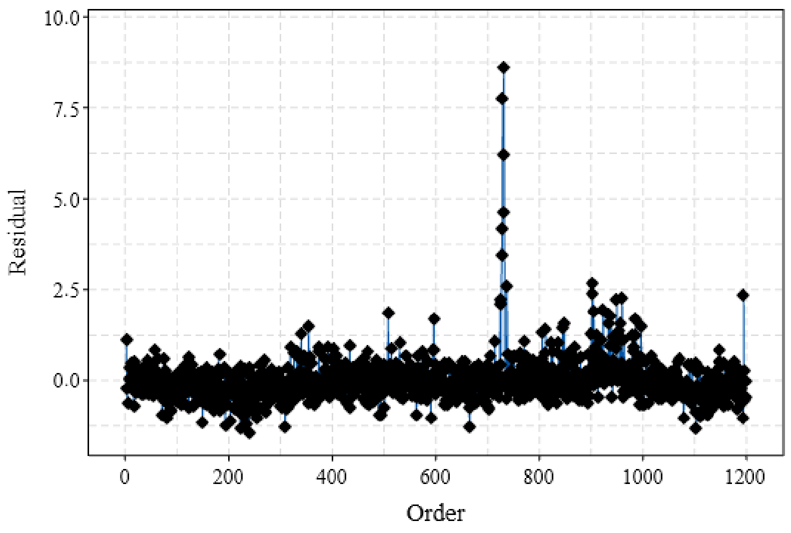
Figure 47 – Normality probability plot of residual *receiver startup time test*



Source: the author

Apart from the *receiver startup time test*, the violation of normality distribution of residuals premise for all *startup time tests* was observed, even with increasing the sample size from $N = 100$ to $N = 400$, in a new *receiver+tracker+altitude* test attempt. Figure 48 presents the residual error versus observation order for *receiver+tracker+altitude* test with $N = 400$. The random distribution of the residual indicates an independent behavior, but there are the presence of some residuals that have abnormal distance from other values in the random sample population, which can contribute to the non-normal distribution of the population.

Figure 48 – Residual error x observation order for *rec.+tracker+altitude*



Source: the author

None of the *startup time tests* conducted have fulfilled the premises for the parametric statistical analysis. As alternative to a parametric model some solutions that include Generalized Linear Models (GLM), non-parametric and transformation models can be applied, once these models allow that the residual error from the response variable follows a distribution other than a normal distribution. For complexity reasons, once the interpretations of results may be less convenient if a transformation is applied, it was decided to use a nonparametric rank basis model, and thus compare the median instead of mean value. Among nonparametrics hypothesis tests, Kruskal-Wallis test can be used to compare two or more medians of groups when the groups have similarly shaped distributions. Observing the treatment distributions (applying an equal variance Levene's test) this premise was not noticed. Instead, the Mood's Median Test was applied. This test determines whether the median of two or more groups differ, not implying that the group's distribution have similar shape. The results for the *receiver startup time test* applying Mood's Median Test are shown in Table 5. For all tests, the $P - Value$ has resulted in $P < 0.05$ ($P - Value = 0.00$), indicating that H_0 is rejected and thus the treatment medians differ.

Table 5 – Mood's Median Test for the *receiver startup time test*

Solution Type	**Median(s)**	**N $\leq$ overall median**	**N > overall median**	**Q3 - Q1**	**95% CI**
direct	6.53	97	3	0.097	(6.53;6.54)
arch_host	7.73	53	47	0.287	(7.68;7.81)
arch_net	9.41	0	100	0.420	(9.39;9.47)
Overall	7.74				

Mood's median test uses chi-square statistics, in conjunction with the chi-square distribution, to calculate the $P - Value$. The test performs the overall median value and then ranks the observations above and below this value. The interquartile range ($Q3 - Q1$) measures the dispersion of data in each group. The range is the distance between the $75th$ percentile ($Q3$) and the $25th$ percentile ($Q1$). The test considers three premises: the data distribution is not normal; the test is applied for small number of observations N < 20 or the data is better represented by median values (the latter was considered in the evaluation); and finally, the response variable must be continuous.

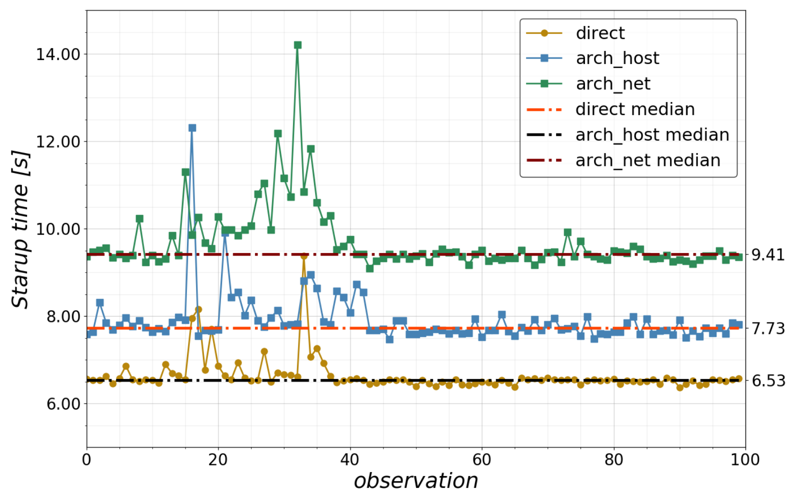
Table 6 summarizes the Mood's Median Test results for all implementations. The mean, median and median CI (95%) are presented, as well as the sample size, N, for each implementation. Increasing the sample size, from $N = 100$ to $N = 400$ for the *receiver+tracker+altitude* implementation, decreases the median 95% CI values range, (for *direct* solution from 0.15 to 0.08, *architecture host* from 0.20 to 0.09 and *architecture net* from 0.25 to 0.14 seconds.

Table 6 – Mood's Median Test for all containers/scripts - *startup time test*

Container/script	**Solution Type**	**N**	**Mean(s)**	**Median(s)**	**Median 95%CI**
	direct	100	6.63	6.53	(6.53;6.54)
Receiver	arch_host	100	7.89	7.73	(7.68;7.81)
	arch_net	100	9.69	9.41	(9.39;9.47)
	direct	100	6.76	6.72	(6.66;6.76)
Tracker	arch_host	100	7.52	7.45	(7.38;7.54)
	arch_net	100	10.88	10.75	(10.68;10.90)
	direct	100	6.81	6.66	(6.59;6.73)
Altitude	arch_host	100	7.45	7.39	(7.30;7.48)
	arch_net	100	10.83	10.71	(10.54;10.87)
	direct	100	13.70	13.44	(13.32;13.37)
Receiver+Tracker	arch_host	100	16.30	15.78	(15.60;16.05)
	arch_net	100	20.82	20.56	(20.33;20.78)
	direct	100	13.26	13.19	(13.16;13.24)
Receiver+Altitude	arch_host	100	15.64	15.60	(15.50;15.62)
	arch_net	100	20.06	19.96	(19.91;20.01)
	direct	100	15.78	15.66	(15.59;15.74)
Receiver+Tracker+Altitude	arch_host	100	18.28	18.18	(18.10;18.30)
	arch_net	100	24.10	24.04	(23.91;24.16)
	direct	400	15.76	15.67	(15.63;15.71)
Receiver+Tracker+Altitude	arch_host	400	18.19	18.12	(18.09;18.18)
	arch_net	400	24.06	24.02	(23.95;24.09)

The individual results for the *receiver*, *tracker*, *altitude*, and *receiver+tracker+altitude* ($N = 100$ and $N = 400$) tests, are presented in the figures 49, 50, 51, 52 and 53, respectively. Figure 49 shows startup time observations for the *receiver* considering the three solution modes. In this case, the median of *architecture host* is $1.20s$ and architecture host of $2.88s$, when greater than the direct solution median.

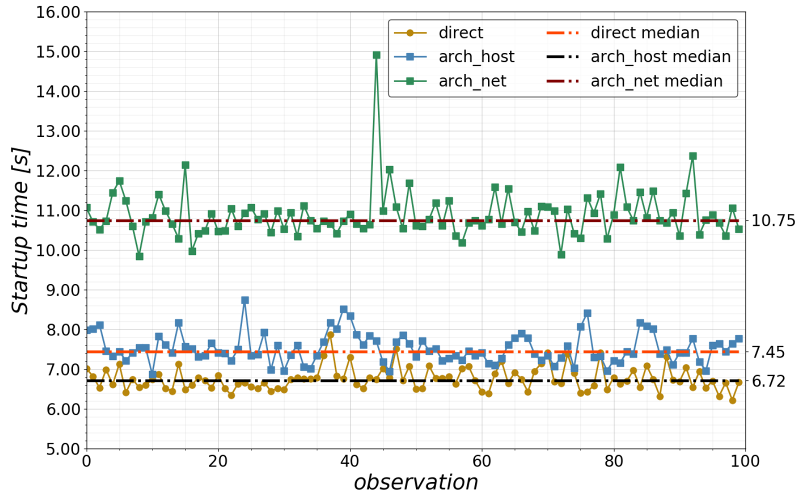
Figure 49 – Deploy time overhead for the receiver



Source: the author

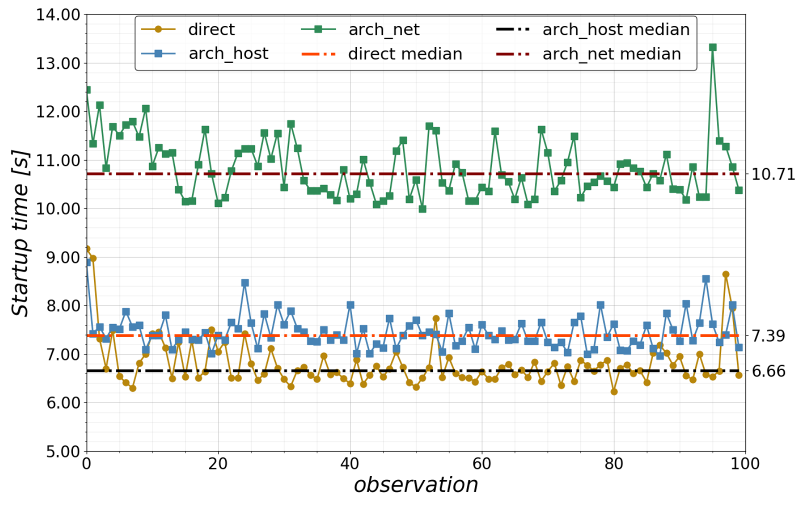
Look into individual *startup time test* results for *tracker* and *altitude* containers/script, showing in Figure 50 and 51 respectively, it has been observed that the *tracker* container when used in *architecture host* mode, exceeds $0.73s$, and $4.03s$ in *architecture net* mode the *direct solution* median. Similar results are observed for *altitude* container which exceeds $0.73s$ the *direct solution* in *architecture host* mode, and $4.05s$ in *architecture net* mode.

Figure 50 – *Startup time test* for the *tracker*



Source: the author

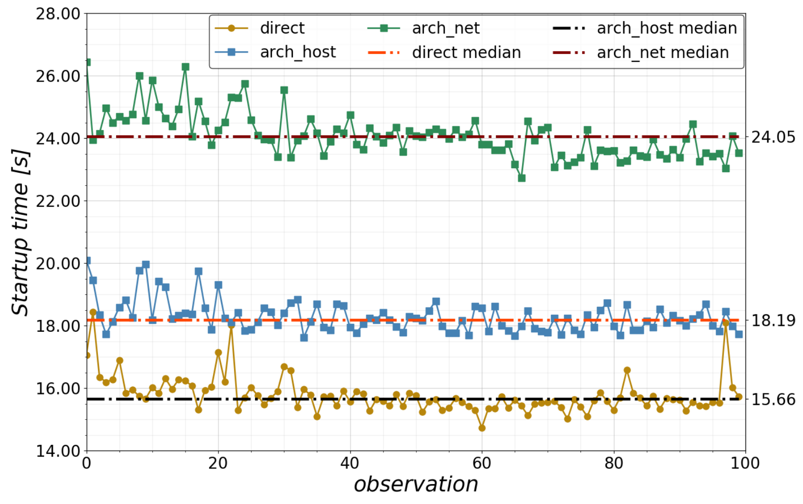
Figure 51 – *Startup time test* for the *altitude*



Source: the author

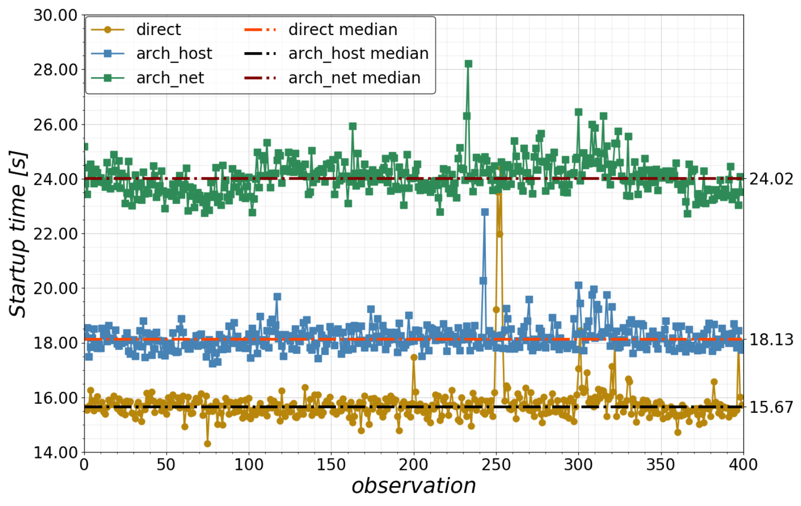
For $N = 400$ observations, examining the *startup time test* for the complete solution, *receiver+tracker+altitude*, Figure 53 the median value of the *architecture host* and *architecture net* exceeds $2.47s$ and $8.35s$ respectively, the *direct solution* median. With $N = 100$ the median of *architecture host* and *architecture net* surpass the *direct solution* median in $2.52s$ and $8.38s$, showing that in terms of median, There weren't significant differences increasing the number of observation from $N = 100$ to $N = 400$.

Figure 52 – *Startup time test* for the *receiver+tracker+altitude* with *N=100*



Source: the author

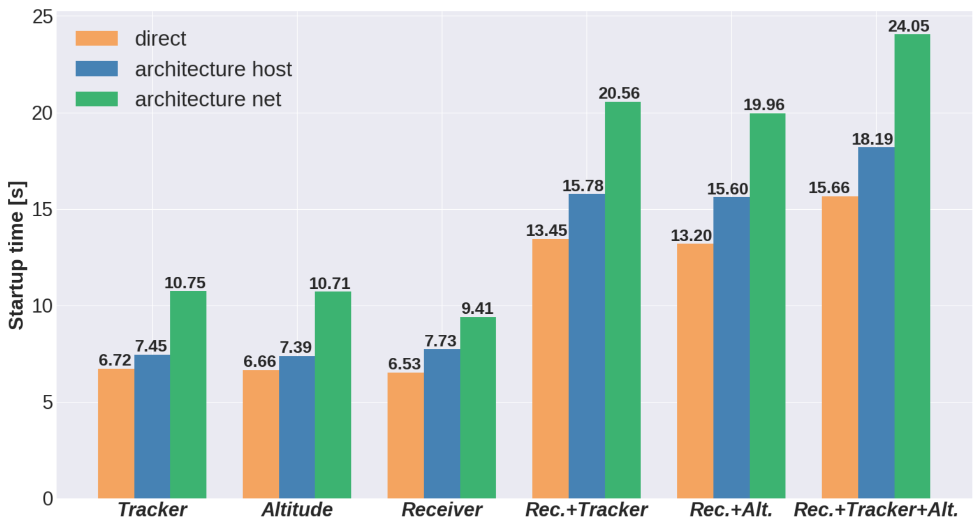
Figure 53 – *Startup time test* for the *receiver+tracker+altitude* with *N=400*



Source: the author

The overall results of the *startup time test* in terms of median values, including the individual results presented above, as well as the complete solution (*Rec.+Tra.+Alt*) and compositions, (*Rec.+Tra* and *Rec.+Alt*), are shown in Figure 54.

Figure 54 – *Startup time test* overall results - median values



Source: the author

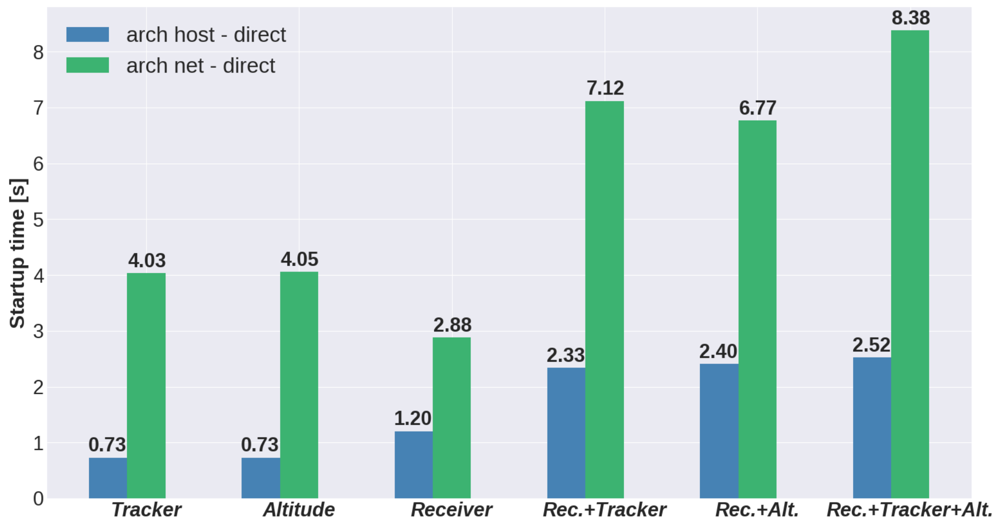
The long delay of the *startup time test*, (that, for instance, comes from around 15s to around 24s for the complete solution), is related to the checker implementation. The verifier could be seen as an essential functionality which asserts the proper operation of the container or script in each observation. Thus, the test takes into account the whole time spent by the verifier interactions, presented by the flowchart in Figure 32.

Finally, to summarize the startup time differences between using the architecture and the *direct solution*, the Table 7 and Figure 55 are presented.

Table 7 – Summary overhead time added by architecture modes - median

Container	***architecture host*** (s)	***architecture net*** (s)
Receiver	1.20	2.88
Tracker	0.73	4.03
Altitude	0.73	4.05
Rec.+Tra.	2.33	7.12
Rec.+Alt.	2.40	6.77
Rec.+Tra.+Alt $(N = 100)$	2.52	8.38
Rec.+Tra.+Alt $(N = 400)$	2.47	8.35

Figure 55 – Overhead median time add by the architecture, *N=100*



Source: the author

Observing the results, Table 7 and Figure 55, it is noticed that the greatest startup time differences between using the architecture and *direct solution* is observed for the complete solution. Maximum median value for *architecture host* is $2.52s$ and $8.38s$ for *architecture net*, showing that the *architecture net* exceeds more than three times the *architecture host* when compared with *direct solution*. For individual container evaluation, the maximum time difference value was $1.20s$ for *architecture host* and $4.05s$ for *architecture net*, ascertained to receiver and altitude containers respectively.

As a consequence of the custom network creation, it is noted that the *architecture net* adds more overhead time than the *architecture host*. The *backend* network must be launched before the containers, so that they can be attached to it. When attached to the network, each container will receive extra configuration, such as IP address and namespace assignment, demanding more time when compared with host networking.

In conclusion, after the statistical analysis of *startup time test*, it was verified that applying the architecture adds overhead time at startup of the solution. This overhead needs to be considered in light of architecture implementation, but, conversely, it must be noticed that the time for deployment is a one-time cost for the user, as discussed in (MARTINS *et al.*, 2020).

7.2 Resource utilization test

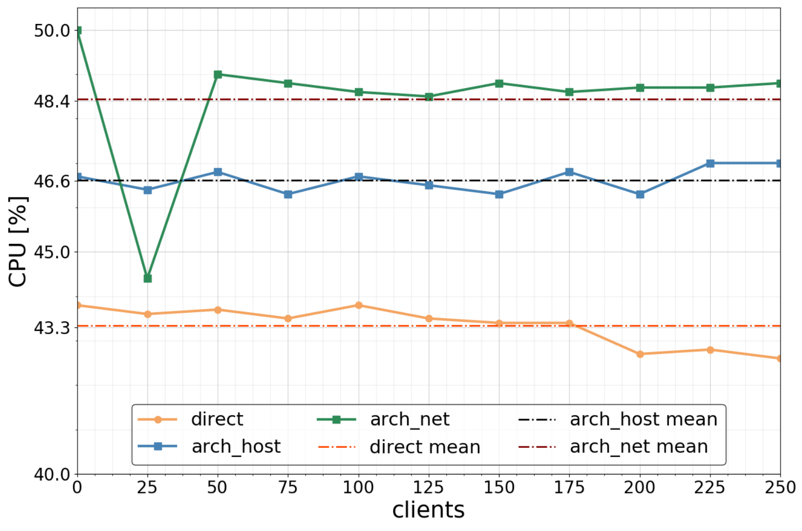
Resource utilization overhead was evaluated in terms of CPU processing and RAM memory. Using an Intel® Core^{TM} i7-4510U CPU @2.00GHz, with 4 physical cores and 8 GB of RAM (7.45 GB available).

In this test the computational resources are measured, in percentage ($0 - 100\%$) of total values, for an increasing number of concurrent clients that send HTTP requests for the *tracker* and *altitude* services.

The start point for this test is to define the metrical criteria. Considering that some HTTP benchmarking tools (*siege, wrk2, ab*) were applied, the constraints of these tools implies that the test must be sequential and not randomized by solution type, (different from what was executed with the *startup time test*). Then a randomized matrix of solution type measures is not feasible, and a parametric model, such as ANOVA, cannot be applied once it is not possible to assure the residual error independent premise. The metrical criteria chosen is then based on the similar work (CARPIO; DELGADO; JUKAN, 2020), which relies on increases the number of concurrent clients C, by 10 levels (e.g. $C = 100$, $C = 200$,..., $C = 1000$), and make at least 1000 measures for level, evaluating the mean and maximum, minimum values. The main metrical difference in this thesis, just as (MARTINS *et al.*, 2020), concerns the number of clients, that increases from 0 to 250 (respecting the boundaries of Linux Apache server), in a step of 25 clients. For each step, $N = 1000$ measures were then performed, one measure for every second, and then the average of the N measures was computed. The interval between the request of a client is set to a constant value $1s$, once the test benchmarks the performance of the solution modes and then a constant interval is preferred instead of random.

The results for the receiver CPU resource utilization, for the *direct solution*, *architecture host* and *architecture net*, are shown in Figure 56. Each level value [C, $CPU[\%]$], corresponds to the mean of $N = 1000$ measures.

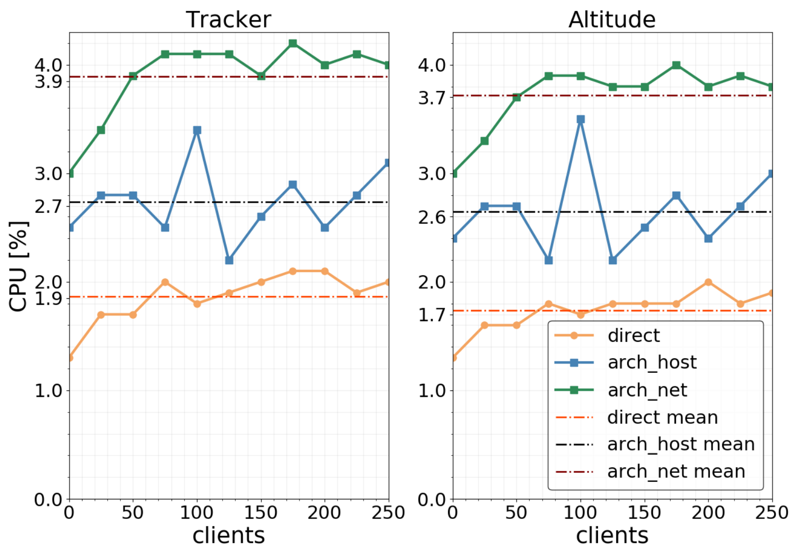
Figure 56 – CPU utilization for the receiver



Source: the author

The results for the *tracker* and *altitude* services CPU utilization resource utilization, for the *direct solution*, *architecture host* and *architecture net*, are shown in figure 56 and 57.

Figure 57 – CPU utilization for the applications

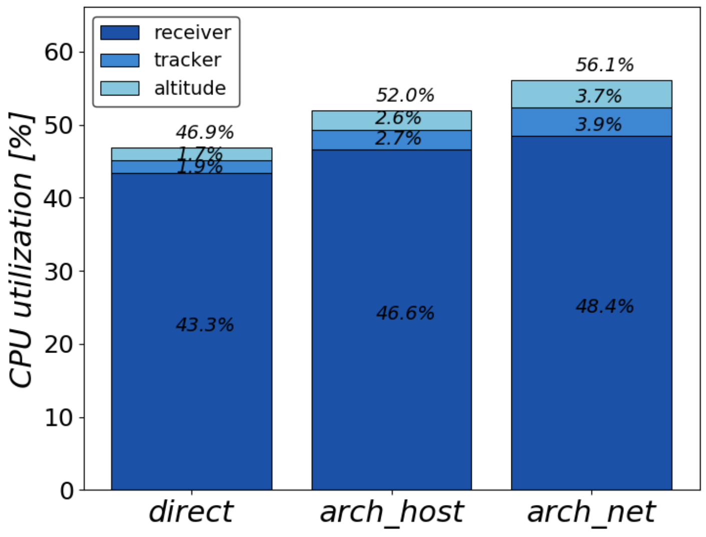


Source: the author

The CPU results, figures 56 and 57, show the differences between mean resource usage for each solution mode over an increase of 0 to 250 concurrent clients, and evince that the *architecture net* mode consume more CPU slices than *architecture host* mode, and both architecture modes consume more CPU slices than *direct solution*. The overall

CPU average values, consider all client levels, are presented in Figure 58.

Figure 58 – *Receiver*, *tracker* and *altitude* CPU utilization average



Source: the author

Looking at the average results for the receiver CPU utilization, it is found the values 43.3%, 46,6% and 48.4%, for the *direct solution*, *architecture host* and *architecture net*, respectively. These results demonstrate that the CPU consumption increases according to network isolating level, which indicates that architecture adds an overhead in CPU resource consumption. Despite the CPU utilization value of at $C = 25$ clients with *architecture net* mode, it is expected to be near to a constant CPU utilization value with an increasing number of clients. Hence the receiver CPU utilization should not be significantly interfered by client requests, and these requests are directed to the application containers. For the application container results, Figure 57, the solution modes results do not follow any exponential or linear behavior with the increasing number of the clients. This behavior can be an indicator that, for the given experiment setup, (i.e. each client added to make a request with a 1s interval), the application servers can handle the requests.

The Table 8 sum up the overhead CPU results of the architecture, compared to direct solution.

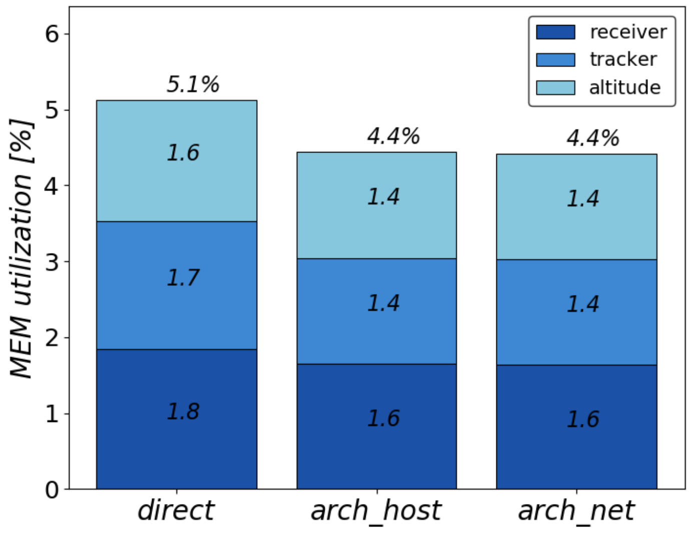
Table 8 – Summary of CPU overhead added by architecture - mean values

Container	***architecture host*** (%)	***architecture net*** (%)
Receiver	3.3	5.1
Tracker	0.8	2.0
Altitude	0.9	2.0

The table 8 shows that the overhead of receiver CPU utilization for the *architecture host* is 3.3% and for *architecture net* 5.1%, and for the application CPU utilization, the *architecture host* adds 0.8% and 0.9%, for the *tracker* and *altitude* services respectively, whereas the *architecture net* adds 2.0% for both services. The results show that there is a CPU cost to implement the architecture, and that this cost also increases with respect to the network implementation. The greater values correspond to receiver implementation of the architecture. In contrast, due to the architecture design, it is expected that the receiver CPU utilization remains constant, scaling the services. A complete scaling test is planned to evaluate this situation in further improvements.

Overall RAM memory utilization results, (mean values for all clients levels), for the three solution modes, are shown in Figure 59.

Figure 59 – *Receiver*, *tracker* and *altitude* RAM utilization average



Source: the author

The Table 9 sum up the overall differences of RAM memory utilization between architecture modes and *direct solution*.

Table 9 – Summary of RAM overhead of the architecture - mean values

Container	***architecture host*** (%)	***architecture net*** (%)
Receiver	-0.2	-0.2
Tracker	-0.3	-0.3
Altitude	-0.2	-0.2

It was observed a slight decrease of memory utilization mean, applying the architecture, (of -0.2% *receiver* and *altitude*, and -0.3% for the *tracker*). Further analysis could be conducted to check the memory consumption in detail with other tests.

7.3 Response time test

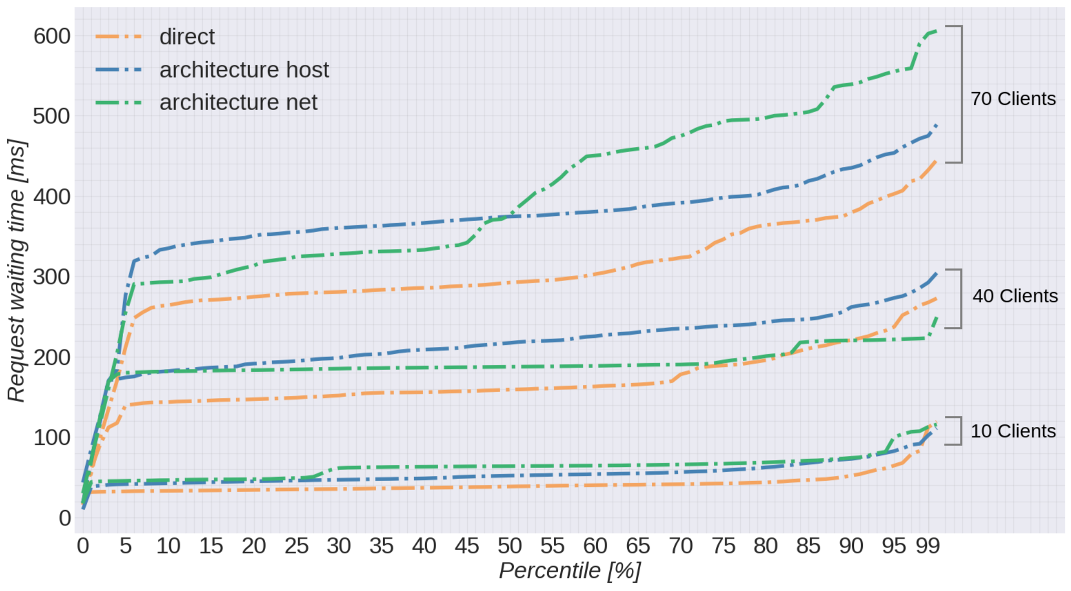
One of the metrics to evaluate web pages performances use the percentile of request waiting response time. The percentile represents a time value where a certain percentage of scores fall below that value. For instance, if the $50th$ percentile of a set of requests is equal to $30ms$, it means that 50% of the requests will experience a delay of $30ms$ or less. The response time test aims to evaluate the response time of the services under an overflow of HTTP requests. The test has two configuration options:

- the number of concurrent clients: C, that also represents the number of requests to be made at once.
- and the total number of requests: NR.

Considering that a certain number of clients C will do a number of requests NR to the HTTP web page, the test works as follows: at the time zero, a group of C requests will be done by the clients. The next C request will be done as soon as the first group receives their responses, until that the accumulated number of requests reaches NR. For instance, for $C = 10$ clients and $NR = 1000$ requests, at the time zero of the test, 10 requests will be done at once. When that 10 initial requests are answered, more 10 requests will be done, and so on, until the accumulated number of requests reaches the total $NR = 1000$ value.

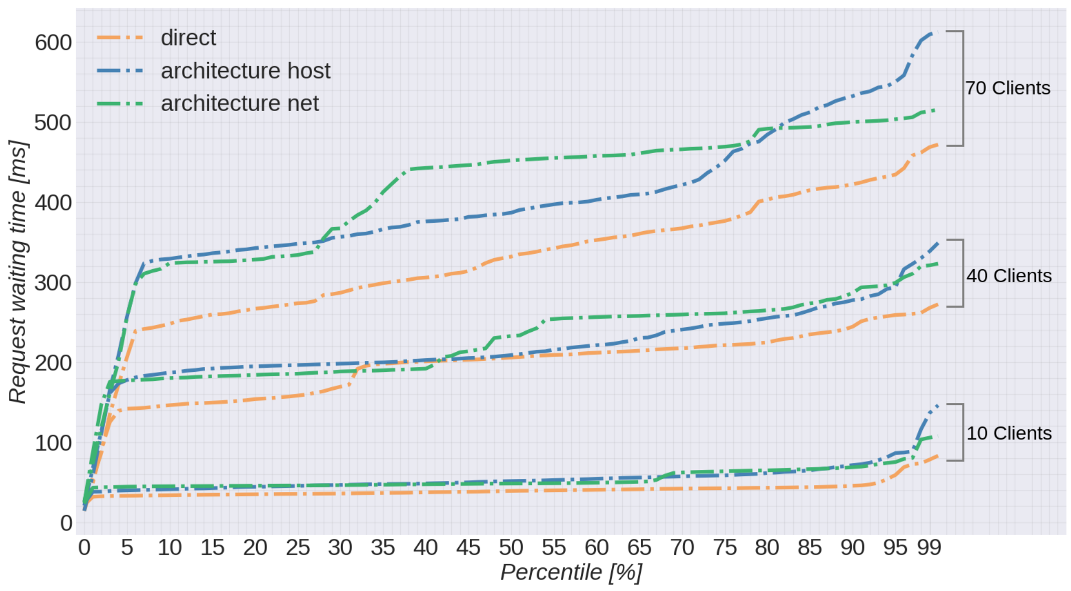
The first test evaluates the response for the solution types, *direct solution*, architecture*host* and *architecture net*, for $C = 10$, $C = 40$, $C = 70$, and $NR = 1000$ requests. Both *tracker* and *altitude* services are evaluated. The results are shown in figures 60 and 61, respectively.

Figure 60 – Request waiting time x percentile, *tracker* service



Source: the author

Figure 61 – Request waiting time x percentile, *altitude* service



Source: the author

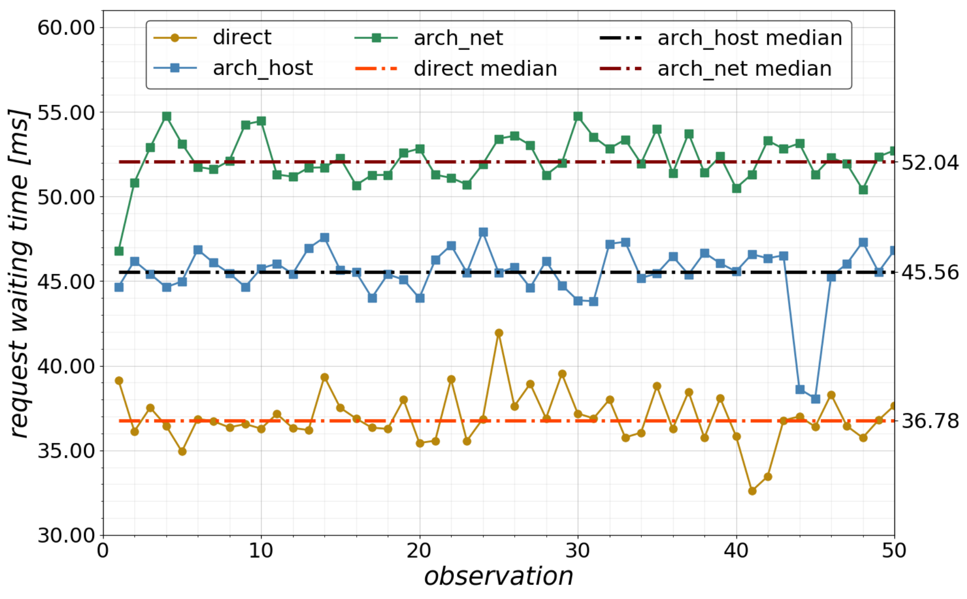
Observing the *response time test* results presented in figures 60 and 61, it is verified that increasing the number of the clients, the request waiting for the solution time increases. Even though observing the data variability, it is not possible to compare the values directly, with just one measure. Then, in order to be able to compare the measuring values of the *response time test*, an experiment plan was conducted including a statistical analysis. The strategy consists of taking some reference percentile values as reference ($10th$, $25th$, $50th$, $75th$, $90th$ and $99th$ were chosen), and make a number of measures for each one of them. Therefore, it is possible to proceed with a statistical analysis of the results. To simplify the analysis, it was also defined to verify the percentile for one of the applications, the *tracker*, then the simulated client requests target address http://localhost:5002/.

Statistical design of the experiment

Likewise the *startup time test*, the guidelines to designing experiments (MONTGOMERY, 2013), for the ANOVA was adopted. The *response time test* is a single-factor experiment (factor = solution type), with 3 levels (or treatments: *direct*, *architecture host* and *architecture net*). The response is the *request waiting time*, in milliseconds (ms), by percentile levels: $10th$, $25th$, $50th$, $75th$, $90th$ and $99th$. Adopting the randomization principle, a complete randomized matrix was generated determining the repetition test order, for an initial number of observations $N = 50$. The number of clients was defined to a fixed value, $C = 10$ and the total number of requests $NR = 1000$.

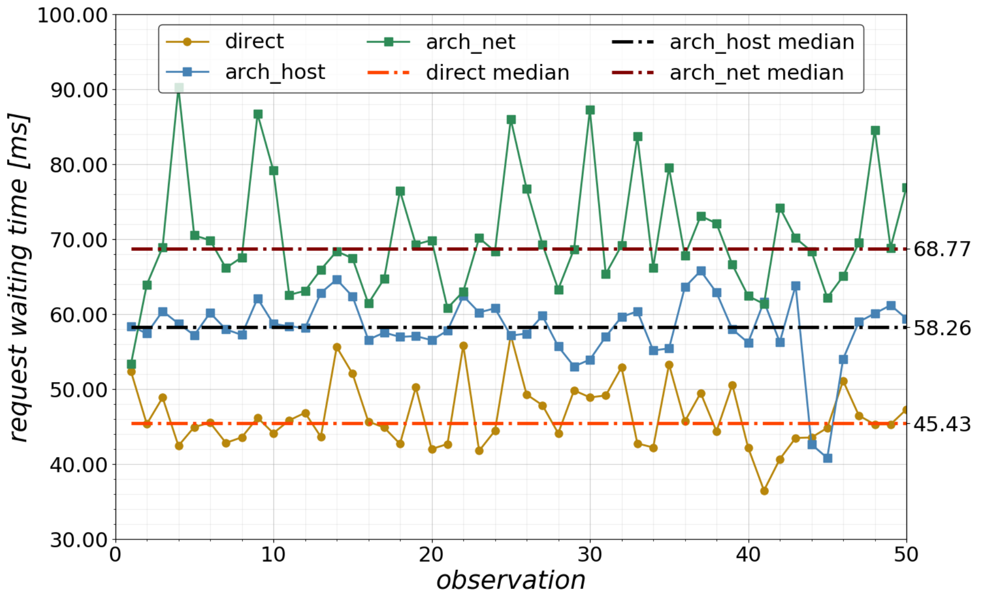
The results for percentile levels $25th$, $75th$ and $99th$ are shown in figures 62, 67 and 64, respectively.

Figure 62 – Request waiting time at $25th$ percentile, *N=50*, *C=10*, *NR=1000*



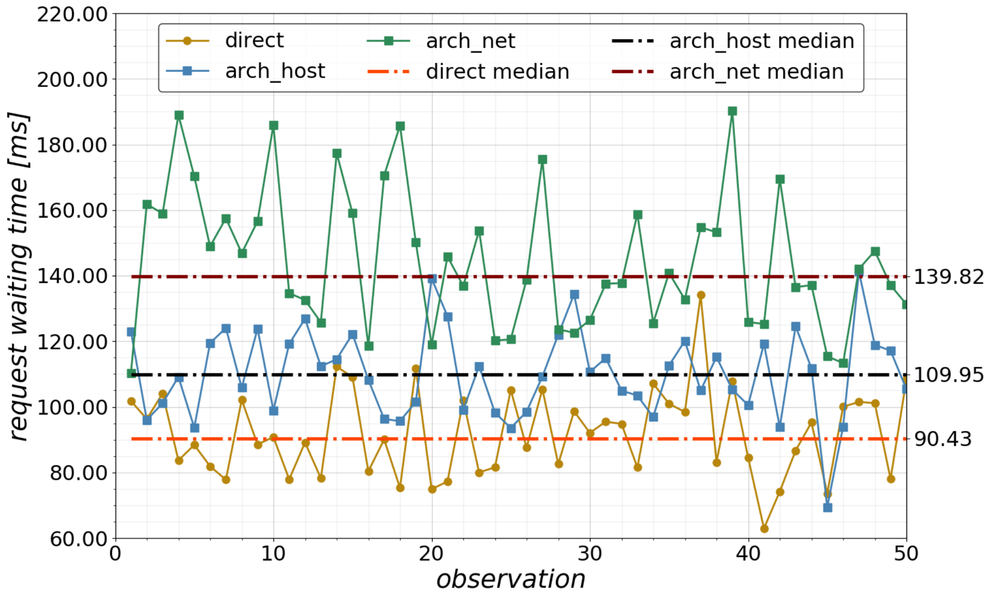
Source: the author

Figure 63 – Request waiting time at $75th$ percentile, *N=50*, *C=10*, *NR=1000*



Source: the author

Figure 64 – Request waiting time at $99th$ percentile, *N=50*, *C=10*, *NR=1000*



Source: the author

Observing the request waiting time, it can be noticed the great variability of the data for $N = 50$. Nevertheless, a first examination was carried out. In order to be able to apply the ANOVA method, supposing a normal residual error distribution, the procedure premises were checked. In addition, to verify if the number of the observations is adequate, the statistical power for ANOVA model with $N = 50$ was also evaluated. The following tests were performed, for confidence intervals of $85\%CI$ and $95\%CI$, to check ANOVA assumptions:

- Ryan-Joiner Test: to evaluate normality distribution of the residual error.
- Levene's Test: to evaluate the equal variance of the treatments

Once a randomized experimental matrix was applied in observation order, the residual error independent premise was checked, observing the random behavior of the *residual error x observation order* plot. Table 10 summarizes the obtained results.

Table 10 – Summary of ANOVA premise check and statistical power $N = 50$

Percentile	**CI** *(%)*	**ANOVA** *P-value*	**Normality** *P-value*[*]	**Equal Variances** *P-value*[*]	**Errors Independent**	**Max.Dif. Mean** *(ms)*	**Power** *(%)*	**N**
10th	85	0.00	< 0.010	0.984	ok	0.64	0.85	50
	95	0.00	< 0.010	0.984	ok	0.77	0.85	50
25th	85	0.00	< 0.010	0.977	ok	0.87	0.85	50
	95	0.00	< 0.010	0.977	ok	1.04	0.85	50
50th	85	0.00	< 0.010	0.744	ok	1.49	0.85	50
	95	0.00	< 0.010	0.744	ok	1.79	0.85	50
75th	85	0.00	< 0.010	0.003	ok	3.19	0.85	50
	95	0.00	< 0.010	0.003	ok	3.84	0.85	50
90th	85	0.00	< 0.010	0.353	ok	4.05	0.85	50
	95	0.00	< 0.010	0.353	ok	4.86	0.85	50
99th	85	0.00	< 0.010	0.001	ok	9.18	0.85	50
	95	0.00	0.05	0.001	ok	11.03	0.85	50

[*] The gray cells indicate test violation

Observing the results of the ANOVA premise check and statistical power for $N = 50$, in Table 10, it is seen that none of the tests has pass in the normal distribution of residual error, then the ANOVA method is not applicable for $N = 50$. It was tried to increase the number of observations to $N = 150$, decrease the confidence levels to $80\%CI$ and $95\%CI$, and then recheck the premises for the ANOVA method. The results are shown in Table 11. The statistical power obtained values, (85%), proved to be acceptable to detect differences between observed means, in all tests with $N = 50$ and $N = 150$. However, the ANOVA test results are inapplicable once the distribution of the error is not normal.

As it was done in *startup time test*, the nonparametric Mood's Median Test was applied. The results are presented in Table 12.

Table 11 – Summary of ANOVA premise check and statistical power $N = 150$

Percentile	**CI** *(%)*	**ANOVA** *P-value*	**Normality** *P-value**	**Equal Variances** *P-value**	**Errors Indepen-dent**	**Max.Dif. Mean** *(ms)*	**Power** *(%)*	**N**
10th	80	0.00	0.010	0.552	ok	0.34	0.85	150
	90	0.00	0.010	0.552	ok	0.39	0.85	150
25th	80	0.00	0.010	0.780	ok	0.45	0.85	150
	90	0.00	0.010	0.780	ok	0.52	0.85	150
50th	80	0.00	0.010	0.273	ok	0.75	0.85	150
	90	0.00	0.010	0.273	ok	0.82	0.85	150
75th	80	0.00	0.010	0.000	ok	1.47	0.85	150
	90	0.00	0.010	0.000	ok	1.69	0.85	150
90th	80	0.00	0.010	0.000	ok	1.98	0.85	150
	90	0.00	0.010	0.000	ok	2.28	0.85	150
99th	80	0.00	0.010	0.000	ok	5.12	0.85	150
	90	0.00	0.05	0.000	ok	5.90	0.85	150

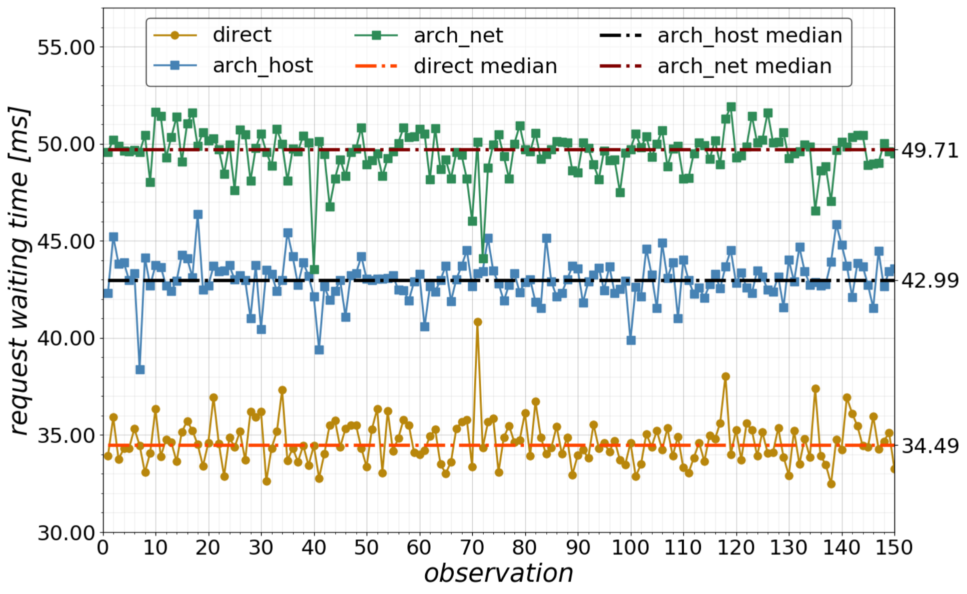
* The gray cells indicates test violation

Table 12 – Mood's Median Test percentiles for treatments - *response time test*

Percentile	**Solution Type**	**N**	**Mean(s)**	**Median(s)**	**Median 95%CI**
	direct	150	34.66	34.49	(34.32; 34.76)
10th	arch_host	150	43.01	42.99	(42.89; 43.17)
	arch_net	150	49.54	49.71	(49.57; 49.86)
	direct	150	37.44	37.10	(36.99; 37.29)
25th	arch_host	150	46.01	46.06	(45.78; 46.23)
	arch_net	150	52.28	52.40	(52.15; 52.53)
	direct	150	41.77	41.10	(40.86; 41.40)
50th	arch_host	150	51.91	51.77	(51.35; 52.44)
	arch_net	150	58.37	58.50	(58.29; 58.88)
	direct	150	47.52	46.18	(45.42; 47.29)
75th	arch_host	150	60.03	59.80	(59.12; 60.64)
	arch_net	150	70.15	69.06	(68.09; 70.35)
	direct	150	56.36	57.42	(56.23; 58.44)
90th	arch_host	150	71.73	72.24	(71.36; 72.91)
	arch_net	150	92.62	95.38	(93.53; 96.51)
	direct	150	94.84	93.49	(90.12; 96.23)
99th	arch_host	150	112.07	112.66	(109.89; 113.81)
	arch_net	150	143.72	144.44	(136.90; 148.74)

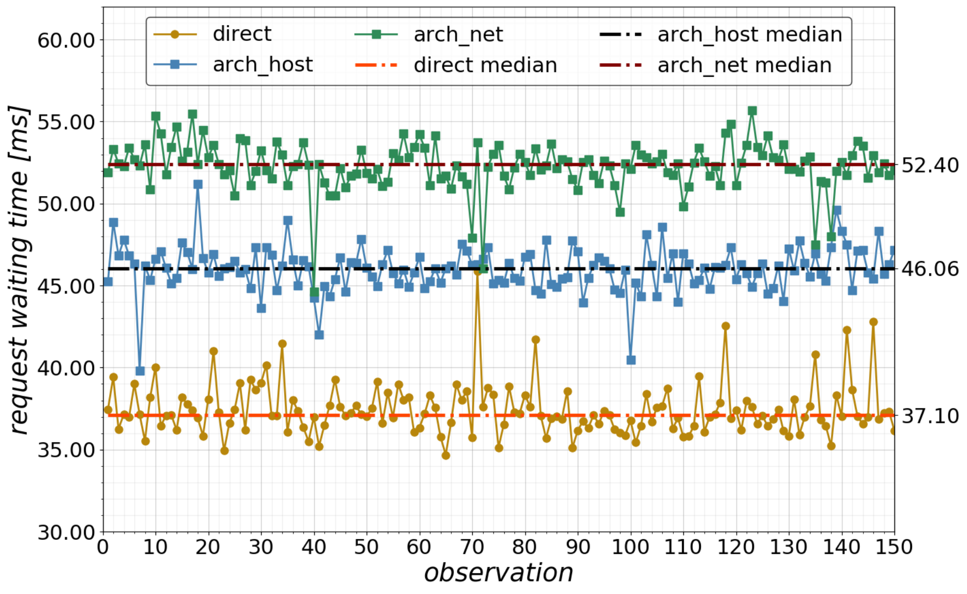
The results obtained at the considered percentile levels are presented in figures 65 to 70.

Figure 65 – Request waiting time at $10th$ percentile, *N=150*, *C=10*, *NR=1000*



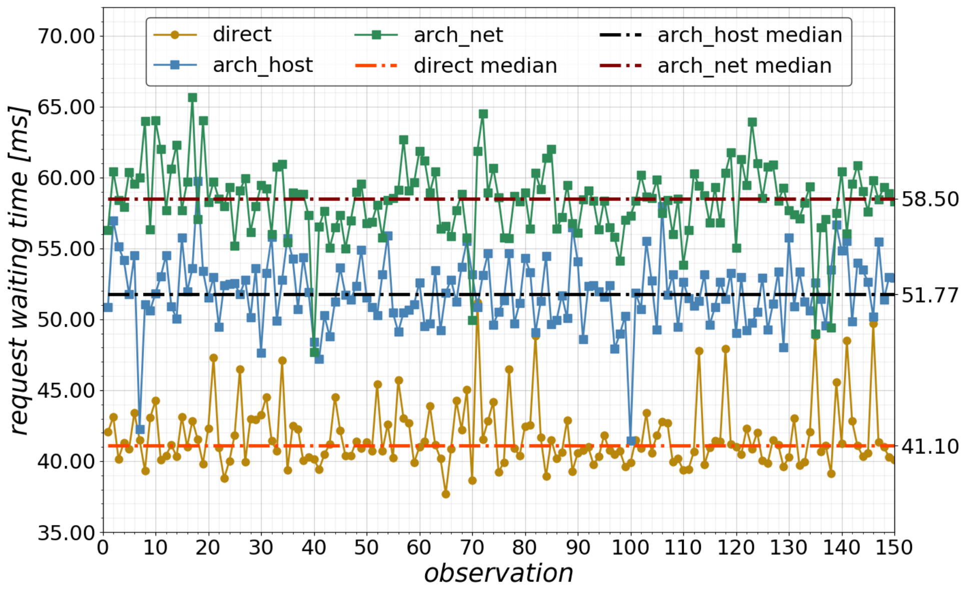
Source: the author

Figure 66 – Request waiting time at $25th$ percentile, *N=150*, *C=10*, *NR=1000*



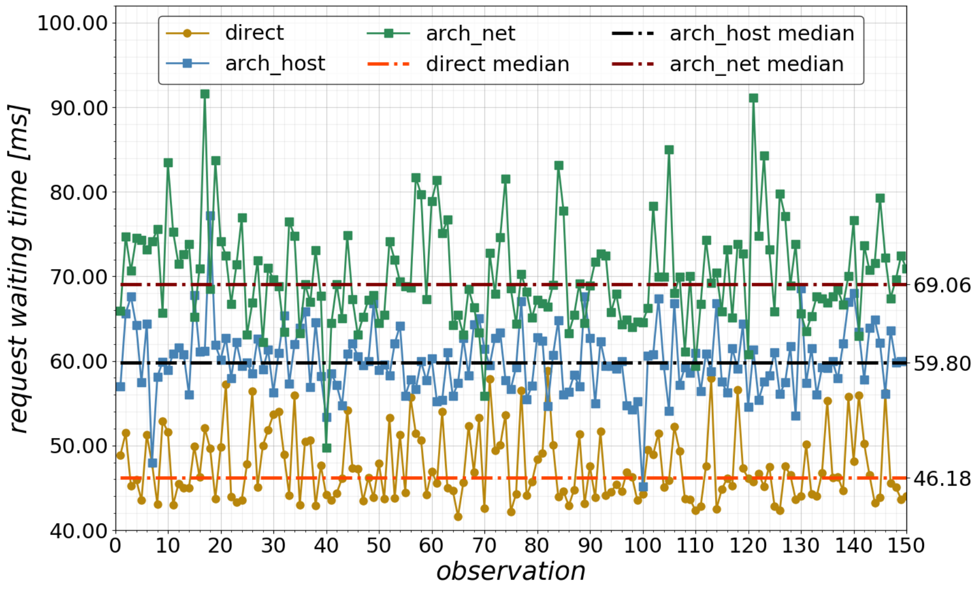
Source: the author

Figure 67 – Request waiting time at $50th$ percentile, *N=150*, *C=10*, *NR=1000*



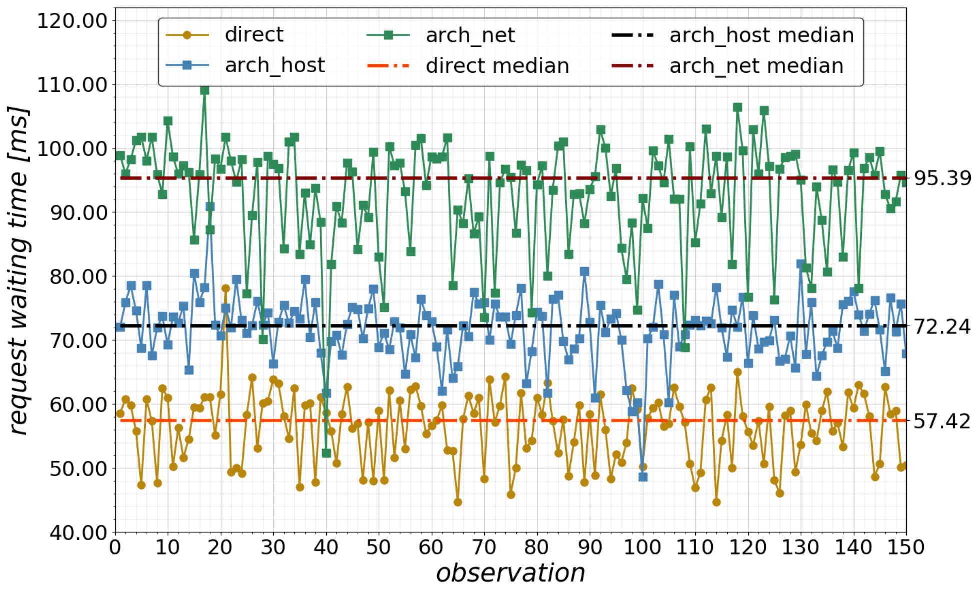
Source: the author

Figure 68 – Request waiting time at $75th$ percentile, *N=150*, *C=10*, *NR=1000*



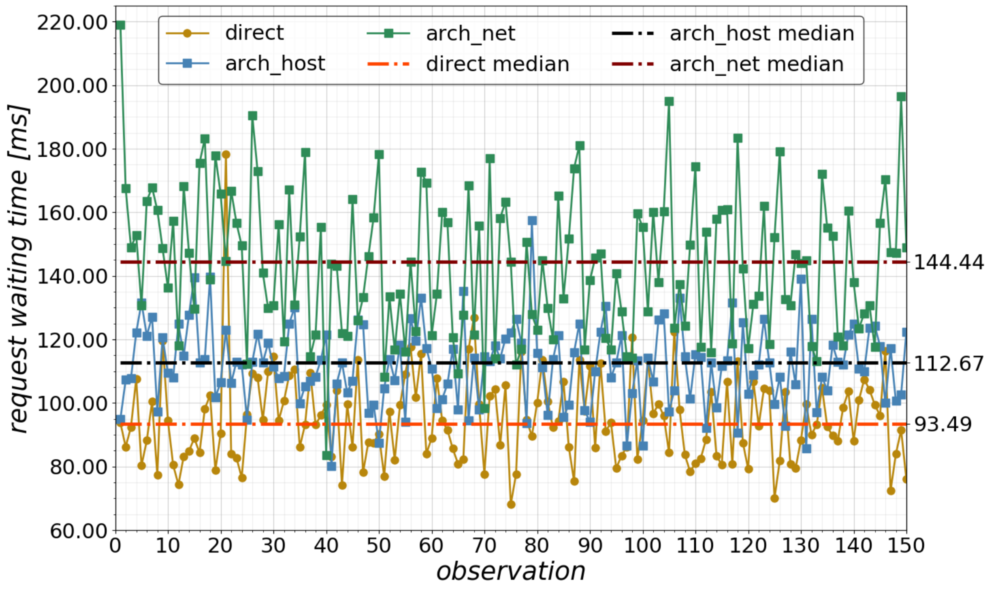
Source: the author

Figure 69 – Request waiting time at $90th$ percentile, *N=150*, *C=10*, *NR=1000*



Source: the author

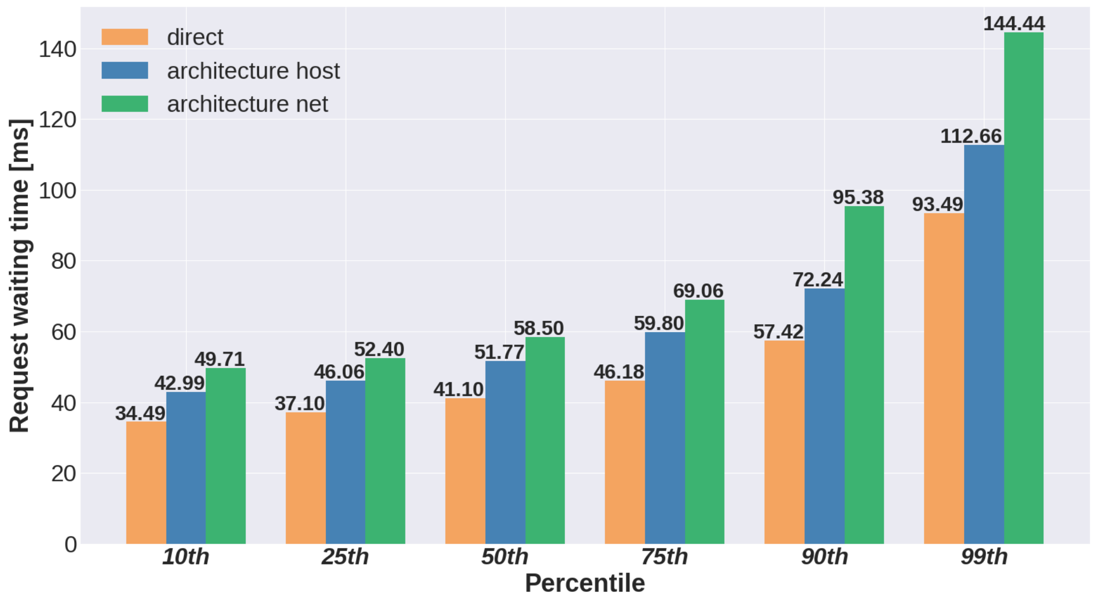
Figure 70 – Request waiting time at $99th$ percentile, *N=150*, *C=10*, *NR=1000*



Source: the author

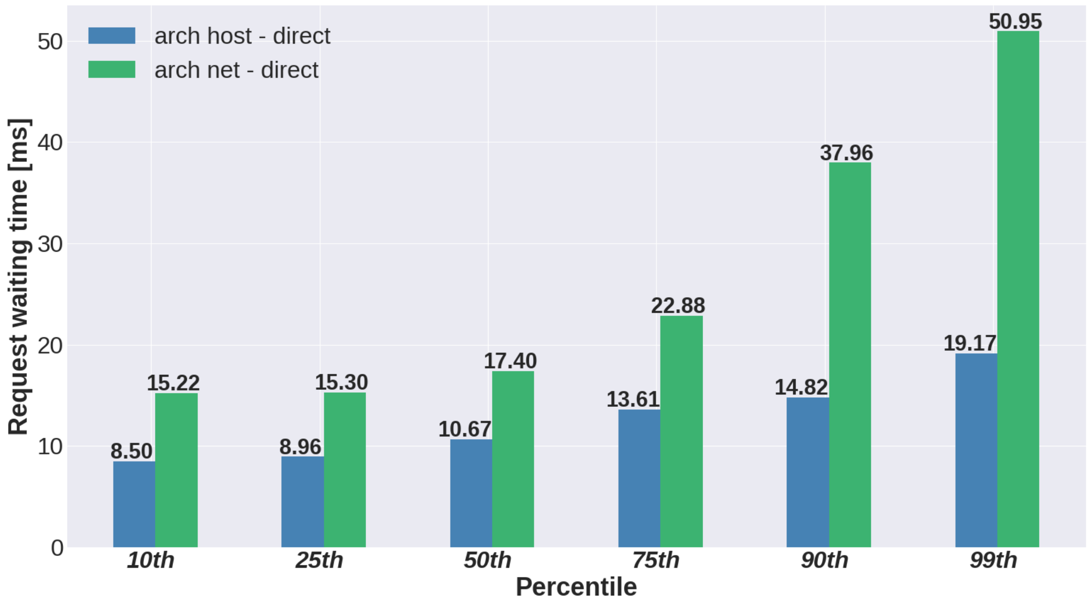
Figure 71, Table 13 and Figure 72 summarize the results for the three solution modes found in the *response time test*, in terms of median values of request waiting time.

Figure 71 – *Response time test* overall results - median values



Source: the author

Figure 72 – Overhead time (ms) added by architecture - median values, *N=150*



Source: the author

Table 13 – Summary of overhead time (ms) added by architecture

Percentile	***architecture host*** (ms)	***architecture net*** (ms)
10th	8.50	15.22
25th	8.96	15.30
50th	10.67	17.40
75th	13.61	22.88
90th	14.82	37.96
99th	19.17	50.96

The request waiting time for each percentile is a median value *ms*, that represents an estimate of the limit value of waiting time at the level. It is observed that the architecture has a cost in terms of response time. This cost comes from $8.50ms$ greater than direct solution, to $19.17ms$ with *architecture host*, and from $15.22ms$ to $50.96ms$ with *architecture net*. At $50th$ percentile level, the overhead is around $10.67ms$ and $17.4ms$ for the architecture host and architecture net, respectively. It means that, at the $50th\%$ percentile is possible that a client requests experiences a delay until $17.4ms$ greater than *direct solution*, using the architecture with isolated network, or $10.67ms$ greater the *direct solution*, using the architecture with low level of isolation, in mode host.

The results show that the higher the percentile, the greater the parameters of dispersion of the data, as seen in the $99th$ percentile.

In the initial test proposition, a verification of ANOVA implementation was conducted, but the data proved to have a nonparametric characteristic, and then the study was driven to the rank test. In further investigation, it is planned to verify the request response time by controlling the request rate. However, the *response time test* proposed in this thesis has its relevance as an initial step in terms of overhead investigation of response time. Further investigation must be led, with other strategies, which can focus on the higher percentiles levels.

8 CONCLUSION AND FURTHER IMPROVEMENTS

Finally, this chapter presents the conclusions of this work and points out further improvements that can be addressed in next implementations.

Conclusion

This thesis has presented a container-based architecture to provide end-to-end services from SDR devices. The architecture presents a possible topology for container storage, orchestration, and management that integrates SDR devices into network environments. Therefore, the proposed solution can be useful to automate network SDR service provision and to integrate SDRs with network environments that apply software defined technologies. The architectural design key relies on mapping the signal path processes to functional blocks, and then providing the blocks' operation through containers. As use cases, ADS-B and LoRa service generation were implemented, and an example of how to add functionalities to the architecture is shown, through the application of service verifiers.

The results show that the architecture implementation brings benefits and drawbacks. The benefits are concentrated in non-functional features, while the drawbacks were verified on system performance degradation, with an overhead introduced. The following non-functional features are enabled by the architecture:

Real-time capability: demonstrated by real-time airplane tracker services.

Control and management: by the architecture templates, which act as a top level control plane, it was shown the possibility to control low-level signal processing and communication parameters, much required in these RF structures. Moreover, the containers managed by the architecture inherit the control features of the Docker virtualization platform, such as: container restart policies, limitation of available resources (CPU quota, memory limit), volume sharing and networking options, etc.

Reusability and reproducibility: through the modularization of the service provision in functional blocks, and the inter-container communication system proposed, it is possible to reuse the created containers, and reproduce the services. A reproducibility test was presented where 80 aircraft tracking services (HTTP servers) were launched, using incoming data from a unique receiver container.

Flexibility and portability: through separating domain logic of DSP script from I/Q digital samples output of SDR platforms, it was possible to demonstrate the applicability of the architecture for different SDR devices types (LimeSDR, RTL-SDR and USRP). These devices were used to provide LoRa and ADS-B services, showing, in addition, that the architecture can also embrace generic RF projects.

Network isolation: in a local scope, two modes of implementing the architecture were presented with different isolation levels in terms of host namespaces. The cost and benefit of each mode were highlighted.

In contrast with the non-functional features added by the architecture, the remarkable drawbacks were verified in terms of overhead, when compared with direct solution. Using the architecture with host networking, *architecture host*, and custom bridge networking *architecture net*, modes for ADS-B service generation use case, the final overhead results are summarized as follows:

Startup time: it was observed an overhead maximum in startup time of $0.73s$ with *architecture host*, and $4.05s$ with *architecture net* to launch individual containers, and $2.47s$ and $8.35s$ for launch the complete solution (three containers).

Resource utilization: it was observed an overhead maximum in CPU utilization of 3.3% and 5.1% for the receiver container, and 0.9% and 2.0% for the application containers, with *architecture host* and *architecture net* modes, respectively.

Response time: analyzing the average request waiting time for the tracker service it was observed overhead, (delay), added by the architecture of $8.50ms$ and $15.22ms$ at the $10th$ percentile, $10.67ms$ and $17.40ms$ at $50th$ percentile, and $19.17ms$ and $50.96ms$ at the $99th$ percentile.

The overall results have pointed out the trade-offs between the architecture implementation modes and the direct solution in a local scope. Taking into account the extent of the topic discussed, and with the delimitation coverage defined for containerized SDR service provision, it was considered that this thesis has fulfilled the expectations, paving the way for further implementations that can use the presented topology and the containerized method as modular components of network environments.

Further improvements

Further improvements are related to the integration of this architecture with software defined technologies, such as NFV and SDN, and microservice implementation in distributed systems.

To make the implementation of this architecture applicable for distributed systems, a main modification have to be performed, as discussed in Section 4.1. Instead of use service verifiers to assert end-user service provision, the container health-check and continuous monitoring tools (available on the containerization platform) can be explored.

Once this modification is considered, it is possible to explore the architecture in an external network scope with multiple hosts. For this purpose, container management tools, such as Swarm and Kubernets, can be used. Therefore, the containers can become independent nodes, (once the communication with the *Container Manager Block* at startup is broken), and their coordination and collaboration in the provision of the service can be investigated.

Once the architecture is applicable for distributed systems, it is possible to extend the tests from a local scope to a microservice scope. Thus, a comparison of trade-offs between enabling non-functional features and performance degradation in distributed systems can be performed. Furthermore, besides reproducibility test, (as the one performed in Section 6.3), the container scalability becomes an interesting evaluation, once it can highlight features such as elasticity and resilience of the architecture components.

As seen in Section 4.2, an interesting subject for further research is the integration of the architecture with technologies such as SDN and NFV in network environments. In these environments, the proposed architecture introduces an edge element into play, the virtualized SDR. In this scenario, several research possibilities can be explored. On NFV context, for instance, the VNF placement problem takes another perspective with the insertion of edge containers that have virtualized SDRs. Due to hardware dependency, the SDR virtualized node could not be moved through the network, being a static element on the service chain. However, node collaboration strategies can be investigated, using multiple SDR and available nodes, in order to define a suitable configuration for the service provision.

APPENDIX A ARCHITECTURE HOST TEMPLATE ADS-B USRP

This file consist in the Docker-compose template applied with *architecture host* mode using USRP SDR for ADS-B *tracker* and *altitude* service provision.

```yaml
version: '3.4'

# build flight tracker service (app1) at host port 5002
# build flight altitude tracker service (app2) at host port 5003

services:
  receiver:
    image: aqualtune/receiver:rxtx
    environment:
      RX_ADDRESS: "addr=192.168.1.100"
      RX_GAIN: 40
      THRESHOLD: 0.025
      TX_ADDRESS: "addr=192.168.1.100"
      TX_GAIN: 5
      TX_AMP: 0.030
    tty: true
    network_mode: host

  app1:
    image: flighttracker:vf
    environment:
      HOST_NET: "True"
    network_mode: host

  app2:
    image: altitudetracker:vf
    environment:
      HOST_NET: "True"
    network_mode: host


```

APPENDIX B ARCHITECTURE NET TEMPLATE ADS-B USRP

This file consist in the Docker-compose template applied with *architecture net* mode using USRP SDR for ADS-B *tracker* and *altitude* service provision.

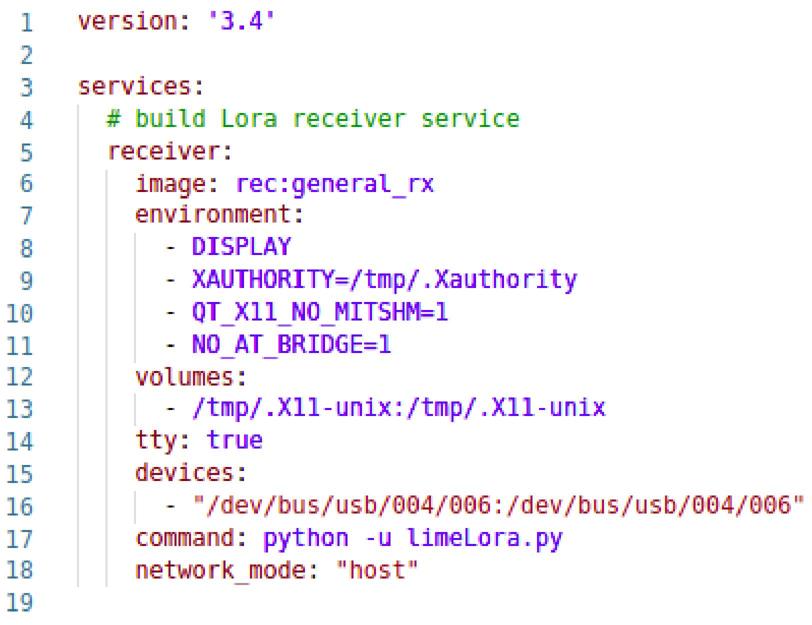
```yaml
version: '3.4'

services:
  # build receiver service
  receiver:
    image: aqualtune/receiver:rxtx
    restart: always
    environment:
      RX_ADDRESS: "addr=192.168.1.100"
      RX_GAIN: 40
      THRESHOLD: 0.025
      TX_ADDRESS: "addr=192.168.1.100"
      TX_GAIN: 5
      TX_AMP: 0.030
    tty: true
    networks:
      backend:
       ipv4_address: 168.1.0.2
  # build flight tracker service (app1) at host port 5002
  app1:
    image: aqualtune/flight_tracker_app
    depends_on:
      - receiver
    environment:
      RECEIVER_ADDRESS: 168.1.0.2
    ports:
      - "5002:5002" #host:container
    networks:
      - backend
  # build flight altitude tracker (app2) at host port 5003
  app2:
    image: aqualtune/flight_altitude_app
    depends_on:
      - receiver
    environment:
      RECEIVER_ADDRESS: 168.1.0.2
    ports:
      - "5003:5003" #host:container
    networks:
      - backend
# generates an internal network
networks:
  backend:
    ipam:
      driver: default
      config:
        - subnet: 168.1.0.0/24

```

APPENDIX C ARCHITECTURE HOST TEMPLATE LORA LIMESDR

This file consist in the Docker-compose template applied with *architecture host* mode using LimeSDR for LoRa *file transfer* service provision.



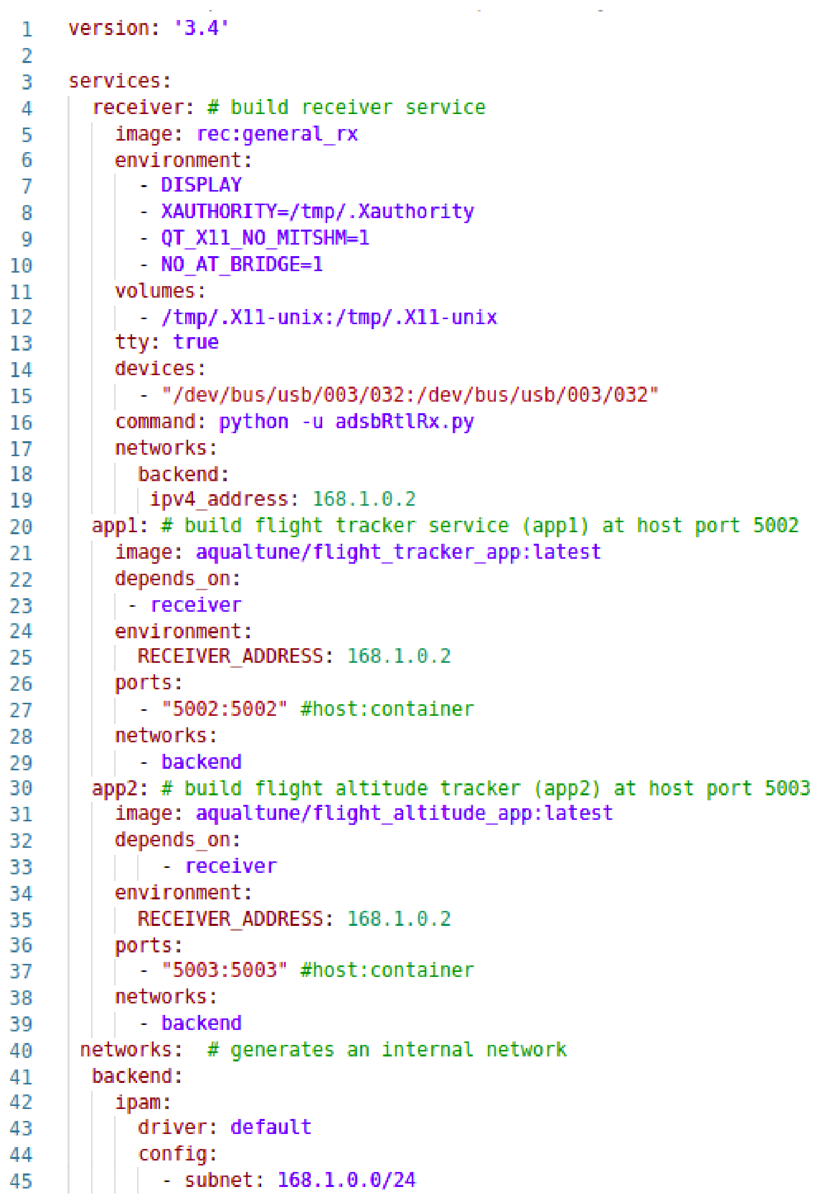
```
version: '3.4'

services:
  # build Lora receiver service
  receiver:
    image: rec:general_rx
    environment:
      - DISPLAY
      - XAUTHORITY=/tmp/.Xauthority
      - QT_X11_NO_MITSHM=1
      - NO_AT_BRIDGE=1
    volumes:
      - /tmp/.X11-unix:/tmp/.X11-unix
    tty: true
    devices:
      - "/dev/bus/usb/004/006:/dev/bus/usb/004/006"
    command: python -u limeLora.py
    network_mode: "host"

```

APPENDIX D ARCHITECTURE NET TEMPLATE ADS-B RTL-SDR

This file consist in the Docker-compose template applied with *architecture host* mode using RTL-SDR for LoRa *file transfer* service provision.



```
version: '3.4'

services:
  receiver: # build receiver service
    image: rec:general_rx
    environment:
      - DISPLAY
      - XAUTHORITY=/tmp/.Xauthority
      - QT_X11_NO_MITSHM=1
      - NO_AT_BRIDGE=1
    volumes:
      - /tmp/.X11-unix:/tmp/.X11-unix
    tty: true
    devices:
      - "/dev/bus/usb/003/032:/dev/bus/usb/003/032"
    command: python -u adsbRtlRx.py
    networks:
      backend:
       ipv4_address: 168.1.0.2
  app1: # build flight tracker service (app1) at host port 5002
    image: aqualtune/flight_tracker_app:latest
    depends_on:
     - receiver
    environment:
      RECEIVER_ADDRESS: 168.1.0.2
    ports:
      - "5002:5002" #host:container
    networks:
      - backend
  app2: # build flight altitude tracker (app2) at host port 5003
    image: aqualtune/flight_altitude_app:latest
    depends_on:
        - receiver
    environment:
      RECEIVER_ADDRESS: 168.1.0.2
    ports:
      - "5003:5003" #host:container
    networks:
      - backend
 networks:  # generates an internal network
  backend:
    ipam:
      driver: default
      config:
        - subnet: 168.1.0.0/24
```

REFERENCES

ADAMS, L. **Choosing the Right Architecture for Real-Time Signal Processing Designs**. US: Texas Instruments, 2002. Available at: <https://www.ti.com/lit/wp/spra879/spra879.pdf>, acessed in 20 Aug 2020.

AHMED, T.; ALLEG, A.; MARIE-MAGDELAINE, N. **An Architecture Framework for Virtualization of IoT Network**. *In*: IEEE CONFERENCE ON NETWORK SOFTWARIZATION (NETSOFT), 2019., 2019. **Proceedings [...]** [S.l.: s.n.], 2019. p. 183–187.

CARPIO, F.; DELGADO, M.; JUKAN, A. **Engineering and Experimentally Benchmarking a Container-based Edge Computing System**. *In*: ICC 2020 - 2020 IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS (ICC), 2020. **Proceedings [...]** [S.l.: s.n.], 2020. p. 1–6.

CHAE, M.; LEE, H.; LEE, K. **A performance comparison of linux containers and virtual machines using Docker and KVM**. **Cluster Computing**, [S.l.], v. 22, p. 1765 – 1775, Dec 2017.

CISCO. **Cisco Annual Internet Report (2018–2023)**. US: Cisco, 2020. Available at: <https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html>, acessed in 15 July 2020.

COOK, J. **Docker for data science**. Santa Monica CA USA: Appress, 2017. 29-34 p.

CZIVA, R.; PEZAROS, D. P. **Container Network Functions: bringing nfv to the network edge**. **IEEE Communications Magazine**, [S.l.], v. 55, n. 6, p. 24–31, 2017.

DUA, R.; RAJA, A. R.; KAKADIA, D. **Virtualization vs Containerization to Support PaaS**. *In*: IEEE INTERNATIONAL CONFERENCE ON CLOUD ENGINEERING, 2014., 2014. **Proceedings [...]** [S.l.: s.n.], 2014. p. 610–614.

ERICH, F. M. A.; AMRIT, C.; DANEVA, M. **A qualitative study of DevOps usage in practice**. Journal of Software: Evolution and Process, [S.l.], v. 29, n. 6, p. e1885, 2017. e1885 smr.1885.

ETSI. **ETSI guideline for NFV reference architecture framework devices**. [S.l.]: 3GPP, 2013. Available at: <https://www.etsi.org/deliver/etsi_gs/nfv/001_099/001/01.01.01_60/gs_nfv001v010101p.pdf>, acessed in 20 sep 2020.

ETSI. **ETSI Requirements for service interfaces and object model for OS container management and orchestration specification**. [S.l.]: 3GPP, 2020. Available at: <https://www.etsi.org/deliver/etsi_gs/NFV-IFA/001_099/040/04.01.01_60/gs_NFV-IFA040v040101p.pdf>, acessed in 03 apr 2021.

EUROCAE. **ED-102A Minimum Operational Performance Standards for 1090 Mhz Extended Squitter Automatic Dependent Surveillance – Broadcast (ADS-B) and Traffic Information Services – Broadcast (TIS-B)**. France: EUROCAE, 2009. 27-35 p.

EUROCONTROL. **Flight Crew Guidance for Flight Operations in ADS-B Only**. Belgium: EUROCONTROL, 2008.

FAA. **AC20-165B Airworthiness Approval of Automatic Dependent Surveillance-Broadcast OUT Systems**. USA: FAA, 2012. 3-5 p.

FETTE, B. A. (Ed.). **Cognitive Radio Technology**. Second Edition. ed. Oxford: Academic Press, 2009. 73-77 p.

GOPALASINGHAM, A. *et al.* **Virtualization of radio access network by Virtual Machine and Docker: practice and performance analysis**. *In*: IFIP/IEEE SYMPOSIUM ON INTEGRATED NETWORK AND SERVICE MANAGEMENT (IM), 2017., 2017. **Proceedings [...]** [S.l.: s.n.], 2017. p. 680–685.

HOSTETTER, M. **gr-adsb**. Maryland US: [s.n.], 2019. GNURadio out-of-tree module Available at: <https://github.com/mhostetter/gr-adsb>, acessed in 10 may 2020.

IBM. **Virtualization**. US: [s.n.], 2019. Virtualization IBM Cloud Education - Compute, Available at: <https://www.ibm.com/cloud/learn/virtualization>-a-complete-guide>, acessed in 12 Dec 2020.

IMRITH, V. N. *et al.* **Dynamic Orchestration of Security Services at Fog Nodes for 5G IoT**. *In*: ICC 2020 - 2020 IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS (ICC), 2020. **Proceedings [...]** [S.l.: s.n.], 2020. p. 1–6.

JAIN, N.; CHOUDHARY, S. **Overview of virtualization in cloud computing**. *In*: SYMPOSIUM ON COLOSSAL DATA ANALYSIS AND NETWORKING (CDAN), 2016., 2016. **Proceedings [...]** [S.l.: s.n.], 2016. p. 1–4.

JOACHIN, T. **Complete Reverse Engineering of LoRa PHY**. Lausanne, Switzerland.: EPFL, 2020. Available at: <https://www.epfl.ch/labs/tcl/wp-content/uploads/2020/02/Reverse_Eng_Report.pdf>, acessed in 15 Dec 2020.

KERRISK, M. **veth(4) — Linux manual page**. [S.l.]: Linux man-pages project, 2020. Available at: <https://man7.org/linux/man-pages/man4/veth.4.html>, acessed in 01 Sept 2020.

KINNARY, J. **Accelerating Development Velocity Using Docker**. San Francisco CA USA: Appress, 2018. 29-34 p.

KIST, M. *et al.* **HyDRA: a hypervisor for software defined radios to enable radio virtualization in mobile networks**. *In*: IEEE CONFERENCE ON COMPUTER COMMUNICATIONS WORKSHOPS (INFOCOM WKSHPS), 2017., 2017. **Proceedings [...]** [S.l.: s.n.], 2017. p. 960–961.

KIST, M. *et al.* **SDR Virtualization in Future Mobile Networks: enabling multi-programmable air-interfaces**. *In*: IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS (ICC), 2018., 2018. **Proceedings [...]** [S.l.: s.n.], 2018. p. 1–6.

KNIGHT, M. **gr-Lora Bastille Research**. [S.l.: s.n.], 2017. GNURadio out-of-tree module Available at: <https://github.com/BastilleResearch/gr-lora>, acessed in 15 Jan 2021.

KRISHNAN, R. *et al.* **Software defined radio (SDR) foundations, technology tradeoffs: a survey**. *In*: IEEE INTERNATIONAL CONFERENCE ON POWER, CONTROL, SIGNALS AND INSTRUMENTATION ENGINEERING (ICPCSI), 2017., 2017. **Proceedings [...]** [S.l.: s.n.], 2017. p. 2677–2682.

KUMAR, B.; DEREMER, D.; MARSHALL, D. **An Illustrated Dictionary of Aviation**. [S.l.]: McGraw-Hill Education, 2005.

LINGAYAT, A.; BADRE, R. R.; GUPTA, A. K. **Performance Evaluation for Deploying Docker Containers On Baremetal and Virtual Machine**. *In*: INTERNATIONAL CONFERENCE ON COMMUNICATION AND ELECTRONICS SYSTEMS (ICCES), 2018., 2018. **Proceedings [...]** [S.l.: s.n.], 2018. p. 1019–1023.

LIU, W. *et al.* **Enabling Virtual Radio Functions on Software Defined Radio for Future Wireless Networks**. Wireless Personal Communications, [S.l.], p. 1579–1595, Apr 2020. Available at: <https://doi.org/10.1007/s11277-020-07245-x>. Acessed: 13 Jun. 2020.

MACHADO, E. R. **gr-adsbTransmit**. Porto Alegre: [s.n.], 2020. GNURadio out-of-tree module, Available at: <https://github.com/edersonrmachado/gr-adsbTransmit>, acessed in 10 Nov 2020.

MACHADO-FERANDEZ, J. **Software Defined Radio: basic principles and applications**. Revista Facultad de Ingenieria, [S.l.], v. 24, p. 79 – 96, 01 2015.

MALHOTRA, L.; AGARWAL, D.; JAISWAL, A. **Virtualization in Cloud Computing**. Journal of Information Technology & Software Engineering, [S.l.], v. 4, p. 2, 2014.

MARTINS, R. J. *et al.* **Micro-service Based Network Management for Distributed Applications**. *In*: AINA 2020 - 2020 IEEE INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION NETWORKING AND APPLICATIONS (AINA), 2020, Caserna, Italy. **Proceedings [...]** [S.l.: s.n.], 2020. p. 922–933.

MITOLA, J. **Software radio architecture: a mathematical perspective**. IEEE Journal on Selected Areas in Communications, [S.l.], v. 17, n. 4, p. 514–538, 1999.

MOGA, A.; SIVANTHI, T.; FRANKE, C. **OS-level virtualization for industrial automation systems: are we there yet?** Proceedings of the 31st Annual ACM Symposium on Applied Computing, [S.l.], 2016.

MONTGOMERY, D. C. **Design and Analysis of Experiments**. 8ed. ed. [S.l.]: John Wiley & Sons, 2013. 12 p.

NI. **UHD USRP Hardware Driver**. [S.l.]: Ettus Research, 2020. Available at: <https://kb.ettus.com/UHD>. Accessed in: 10 set. 2020.

PALURU, M. **Introduction to Container Networking**. [S.l.]: rancher, 2019. Available at: <https://rancher.com/learning-paths/introduction-to-container-networking/>, acessed in 30 Aug 2020.

SANFILIPPO, S. **Dump1090 a simple Mode S decoder for RTLSDR devices**. Italy: [s.n.], 2013. Open Source Software module Available at: <https://github.com/antirez/dump1090>, acessed in 05 may 2020.

SCHOUHAMER IMMINK, K.; PATROVICS, L. **Performance assessment of DC-free multimode codes**. Communications, IEEE Transactions on, [S.l.], v. 45, p. 293 – 299, 04 1997.

SINHA, D.; VERMA, A. K.; KUMAR, S. **Software defined radio: operation, challenges and possible solutions**. *In*: INTERNATIONAL CONFERENCE ON INTELLIGENT SYSTEMS AND CONTROL (ISCO), 2016., 2016. **Proceedings [. . .]** [S.l.: s.n.], 2016. p. 1–5.

SMITH, D. **A perspective on multi-band, multi-mission radios**. *In*: MILCOM '95, 1995. **Proceedings [. . .]** [S.l.: s.n.], 1995. v. 1, p. 45–49 vol.1.

SOA-RAF. **Reference Architecture Foundation for Service Oriented Architecture Version 1.0**. [S.l.]: OASIS Comittee Specification 01, 2012. 21 p. Available at: <http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html.>.

STRUHÁR, V. *et al.* **Real-Time Containers: a survey**. *In*: WORKSHOP ON FOG COMPUTING AND THE IOT (FOG-IOT 2020), 2., 2020, Dagstuhl, Germany. **Proceedings [. . .]** Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020. p. 7:1–7:9. (OpenAccess Series in Informatics (OASIcs), v. 80).

SU, G. **cVM: Containerized Virtual Machine**. *In*: IEEE 6TH INTERNATIONAL CONFERENCE ON COLLABORATION AND INTERNET COMPUTING (CIC), 2020., 2020. **Proceedings [. . .]** [S.l.: s.n.], 2020. p. 1–7.

TASCI, T.; MELCHER, J.; VERL, A. **A Container-based Architecture for Real-Time Control Applications**. *In*: IEEE INTERNATIONAL CONFERENCE ON ENGINEERING, TECHNOLOGY AND INNOVATION (ICE/ITMC), 2018., 2018. **Proceedings [. . .]** [S.l.: s.n.], 2018. p. 1–9.

ULUSOY, S. **How Docker Container Networking Works - Mimic It Using Linux Network Namespaces**. [S.l.]: Dev Community, 2020. Available at: <https://dev.to/polarbit/how-docker-container-networking-works-mimic-it-using-linux-network-namespaces-9mj>, acessed in 30 Aug 2020.

ZENGLIN, D. *et al.* **Software Defined Systems - Sensing, Communication and Computation**. Switzerland AG 2020: Springer, Cham, 2020. 1,10 p.