# Containers-Based Network Services Deployment: A Practical Approach

Andrés Yazán[1], Christian Tipantuña[1], and Jorge Carvajal-Rodriguez[1]

*Abstract* — **In recent years, virtualizing network services and functions has enabled optimizing hardware resources on resource-constrained devices, such as CPU, memory, and storage. Traditional virtualization is achieved through virtual machines using a layer known as a hypervisor. While this form of virtualization offers advantages such as scalability and portability, it has disadvantages in terms of performance compared to nonvirtualized deployments. In this context, alternative virtualization technologies (like containers) allow virtualization on the same physical infrastructure, improving overall performance, portability, and service scalability. This paper implements the deployment of network services on the Raspberry Pi development platform, which has limited resources. This is achieved through a multi-container virtualization solution using the Docker Compose tool, based on Docker containerization technology. Finally, a performance analysis of the implemented virtualization solution is conducted in terms of resource utilization by each service.**

*Keywords* — **Virtualization; Virtual Machines; Container Raspberry Pi; Docker; Docker Compose; Performance.**

*Resumen* — **En los últimos años, la virtualization de servicios y funciones de red ha permitido optimizar los recursos de hardware, como CPU, memoria y almacenamiento, en equipos con limitaciones de recursos. La virtualization tradicional se lleva a cabo mediante máquinas virtuales, utilizando una capa conocida como hipervisor. A pesar de que esta forma de virtualization ofrece ventajas como escalabilidad y portabilidad, presenta desventajas en términos de rendimiento en comparación con un despliegue no virtualizado. En este contexto, han surgido tecnologías alternativas de virtualization (como los contenedores) que permiten la virtualization en la misma infraestructura física, mejorando el rendimiento general, la portabilidad y la escalabilidad de los servicios. En este artículo se implementa el despliegue de servicios de red en la plataforma de desarrollo Raspberry Pi que cuenta con recursos limitados. Esto se logra mediante una solución de virtualization multicontenedor utilizando la herramienta Docker Compose, basada en la tecnología de contenerización Docker. Finalmente se lleva a cabo un análisis del rendimiento de la solución de virtualization implementada en términos de la utilization de recursos por parte de cada uno de los servicios.**

*Palabras Clave* — **Virtualization; Máquinas Virtuales; Contenedor; Raspberry Pi; Docker; Docker Compose; Rendimiento.**

## I. INTRODUCTION

THE growing demand for network services, applications, and resources from end-users has created limitations in the capacity of service providers to meet these needs due to a shortage of necessary hardware resources to scale proportionally to the demands. Service providers have had to adopt new technologies to meet current demands, maximize resource efficiency, and offer end-users high-quality service (QoS). In this context, virtualization technologies plays a fundamental role in the information technology industry. While virtualization technologies such as Virtual Machines (VMs) provide virtualized services, they present significant performance and resource efficiency problems. For this reason, container-based virtualization technologies have become the preferred choice, as they offer highly efficient virtualized services by operating directly on a device's native software infrastructure, leveraging the features of an operating system kernel to create virtualization. These features include 'namespaces' and 'cgroups', which provide an isolated and independent environment within the native infrastructure in which they run [1].

This paper aims to describe, implement, and analyze a solution for network services based on Docker containers. Is analyzed the performance of containerized services in environments with limited CPU, memory, and storage resources, such as Raspberry Pi development boards. To achieve this, the work is structured as follows:

Section II: A brief description of concepts and related work on virtualization technologies, virtual machines, Docker containers, network services, and microservices is provided.

Section III: Describes the methodology, test environment, software tools, and hardware used for designing and implementing network services using Docker containers.

Section IV: are presented the results obtained in the implementation, and evaluated the performance of the implemented containerization system based on CPU usage, memory usage, and load.

Section V: Provides concluding remarks about the work developed.

Authors acknowledge the support provided by Escuela Politécnica National in the project PIIF-21-04. Corresponding author: christian.tipantuna@epn. edu.ec.

[1] Andrés Yazán is in the Department of Electronics, Telecommunications and Information Networks of the National Polytechnic School, Quito 170517, Ecuador, (e-mail: andres.yazan@epn.edu.ec). ORCID number 0009-0001-7811-1275.

[2] Christian Tipantuña is in the Department of Electronics, Telecommunications and Information Networks of the National Polytechnic School, Quito 170517, Ecuador, (e-mail: christian.tipantuna@epn.edu.ec). ORCID number 0000-0002-8655-325X.

[3] Jorge Carvajal-Rodriguez is in the Department of Electronics, Telecommunications and Information Networks of the National Polytechnic School, Quito 170517, Ecuador, (e-mail: jorge.carvajal@epn.edu.ec). ORCID number 0000-0003-0369-9964.

Manuscript Received: Sep 22, 2023
Revised: Nov 08, 2023
Accepted: Nov 27, 2023
DOI: https://doi.org/10.29019/enfoqueute.1005

## II. BACKGROUND AND RELATED WORK

This section presents the fundamental concepts of virtualization architectures based on Virtual Machines (VMs), followed by container-based virtualization, emphasizing Docker technology. Additionally, we will provide a brief description of microservices infrastructure.

Virtualization is a process or technology that allows the segmentation of software and hardware resources from a physical architecture to deploy multiple dedicated resources in virtual environments for processes or applications. The main virtualization technologies include those based on hypervisors and container-based technologies. Hypervisor-based technology adds a layer of software to the conventional computational architecture, specifically to the underlying operating system, also known as the hypervisor. This layer virtualizes and manages the system's physical resources, such as CPU, RAM, and storage, distributing them optimally for each Virtual Machine (VM). Thus, the hypervisor creates an isolated environment for each VM with its complete operating system, allowing independent execution from the main operating system [2], However, this hypervisor-based approach may decrease overall performance, especially in environments with high resource demand. It is important to note that this hypervisor layer is present in all traditional virtualization technologies, meaning its negative impact on performance is constant in any implementation case, as mentioned [3].

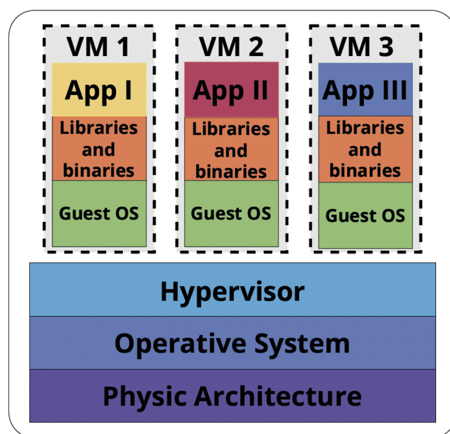
On the other hand, containerization technologies are software that represents a virtualization operating at the kernel level of the operating system. They enable the execution of applications and services in isolated and portable environments within the same physical system, called containers [2], Additionally, using namespaces and cgroups, this technology allows the isolation and resource management for each container [4].

In practice, container processes outperform hypervisorbased solutions by eliminating the virtualization layer and operating directly on the host system's kernel, as shown in Figure 1 [2], In addition to this advantage, containers have inherent characteristics derived from underlying technologies, such as autonomy and independence between containers in resource usage and deployment. Portability is also noteworthy, as containers use lightweight application images, usually in the order of MBs, compared to the GBs images of VMs that include a complete operating system. These features enable services to achieve high scalability and easy migration. Furthermore, they ensure high service availability by allowing the execution of multiple instances of the same application to maintain uninterrupted service, even if one or more containers of the same image stop working [1], [3], [5]. These advantages are enhanced when combined with a microservices-based architecture, which defines a software design model with functions of a service distributed through autonomous and independent modules (Figure 2b), unlike the traditional architecture of monolithic applications, where functions are integrated into a single structure and are interdependent (Figure 2a) [6]. Container characteristics allow leveraging this type of model to optimize service deployment, unlike those based on VMs.

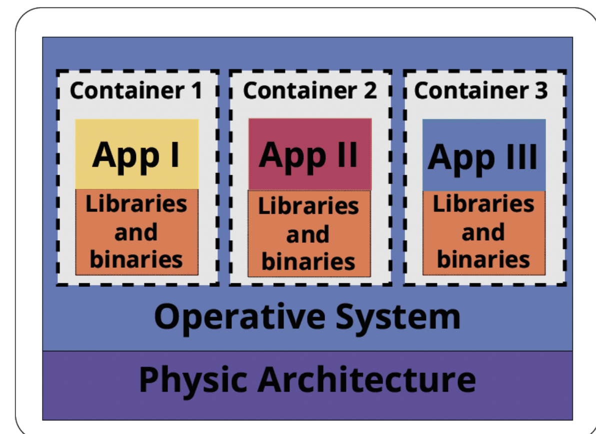
On the other hand, containers also have disadvantages compared to VMs. For example, although containers are independent and isolated entities, they do not provide complete isolation with the operating system where they share kernel resources, as in the case of VMs. This, in turn, poses a security issue, as any impact at the operating system level can affect containers, as indicated in previous lines [7], [8], [9].

In environments with limited hardware resources, such as the one in this study, a container-based approach leverages these characteristics for optimal and scalable service deployment. In this case, security is not a parameter of study, this work focuses exclusively on measuring resource usage.

Docker is an open-source containerization technology that creates, runs, and manages fightweight, portable, and self- sufficient containers. These features have established Docker as a leader in the containerization technology market [10]. Docker implements its architecture on the operating system kernel to achieve container deployment with these characteristics, utilizing namespaces and cgroups to isolate and manage container resources. Docker also uses a file system known as the 'Advanced Union File System' (AUFS) for layer based image construction, contributing to storage resource optimization during Docker image creation [11].

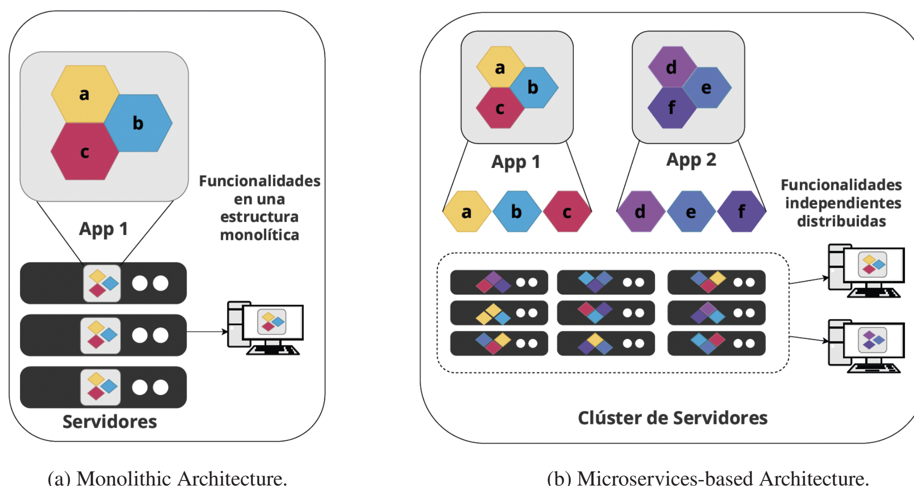



(a) Virtualization Architecture based on Hypervisor




(b) Virtualization Architecture based on Containers

**Fig. 1.** Virtualization Architectures Comparison, based on [2], [4].




(a) Monolithic Architecture.

(b) Microservices-based Architecture.

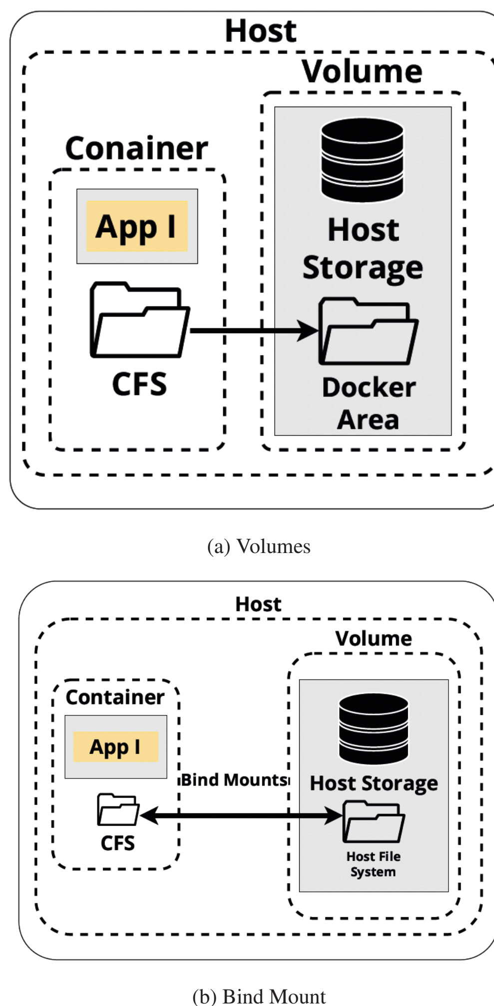
**Fig. 2.** Comparison of service architectures, based on [6].

Above the underlying technologies, there is Docker's architecture known as 'Docker engine', which is the specific name for the containerization technology developed by Docker. This architecture comprises 'Docker objects', representing functionalities within the Docker environment. These functionalities relate to storage, networking, container images, and containers. These elements can have different types depending on their utility and characteristics. Docker objects related to storage functionality are called 'Docker volumes'. These are storage mechanisms created from directories or files stored on the host. Once created, directories are mounted into the container to access a file system outside the isolated environment, as shown in Figure 3a [12].

On the other hand, storage mechanisms are similar to volumes known as 'bind mounts'. Unlike volumes, these are not managed by Docker. They allow data to be mounted to a specific folder on the host into a container, and the stored data persists beyond the container's lifecycle, as illustrated in Figure 3b [13]. 'Volumes' and 'bind mounts' can be used by multiple containers to share the same storage space. Additionally, these mechanisms can be employed for migrating data from containerized services, considering their functionality and ability to maintain data persistence between containers and over time.

Regarding network functionalities, Docker offers what are known as 'Docker networks', entities responsible for providing basic network functionalities through 'network drivers', which share the same name as the network they manage [14]. Docker provides three default networks:

- None: This Docker network has no network interface outside the container. It only has a connection between the container and the loopback interface, and it is commonly used for offline testing, as illustrated in Figure 4a [15].
- Bridge: The 'bridge' network is Docker's default network and uses Linux's bridge functionality to allow communication between containers. Docker creates virtual connections between containers and the virtual network interface called 'dockerO', as shown in Figure 4b. An internal network is created when this connection is established, and IP addresses are automatically assigned to each container, enabling communication within this network. However, initially, they cannot communicate outside of this network. Nevertheless, using iptables and Docker's Network Address Translation (NAT), it is possible to configure the port mapping to allow communication from the container network to the external network [16].




(a) Volumes

(b) Bind Mount

**Fig. 3.** Docker storage mechanisms: Volumes, Bind Mounts, Tmpfs Mounts, based on [12], [13].

- Host: The 'host' network allows the container to share the same network namespace as the host. In other words, the container shares all network interfaces of the host without any level of abstraction between them, as shown in Figure 4c. Due to this configuration, this network offers better performance than other networks, as it does not require addressing, port mapping, or NAT for container connections to an external network since the container network is identical to the host network. However, it is important to mention that if two or more containers attempt to use the same port under a host network, conflicts may arise [17].

It is worth mentioning that, according to [18], [19], [20], 'bridge' type networks tend to have lower performance than other types of networks. On the other hand, 'host' type networks, due to their network characteristics, do not have an abstraction level that limits their performance, as is the case with 'Bridge' type networks.

## III. METHODOLOGY

This section describes the process and methodology for implementing network services in Docker containers. All files used for deploying the services, such as 'dockerfiles' and 'Docker compose', along with the employed procedure, are available in [21]. The images corresponding to the implemented services are also hosted in the Docker Hub repository in [22].

### A. Scenario

The implemented solution is based on Docker Compose and utilizes two Raspberry Pi boards to carry out a multi-container deployment of services. These services constitute a traditional Internet architecture, including DHCP, DNS, FTP, Web, VoIP, and Routing. All of this is achieved through a YAML file. In this approach, one of the Raspberry Pi boards serves as the main host for the containerized services, while the second one is used for remote client connections through the containerized DHCP and Routing services.

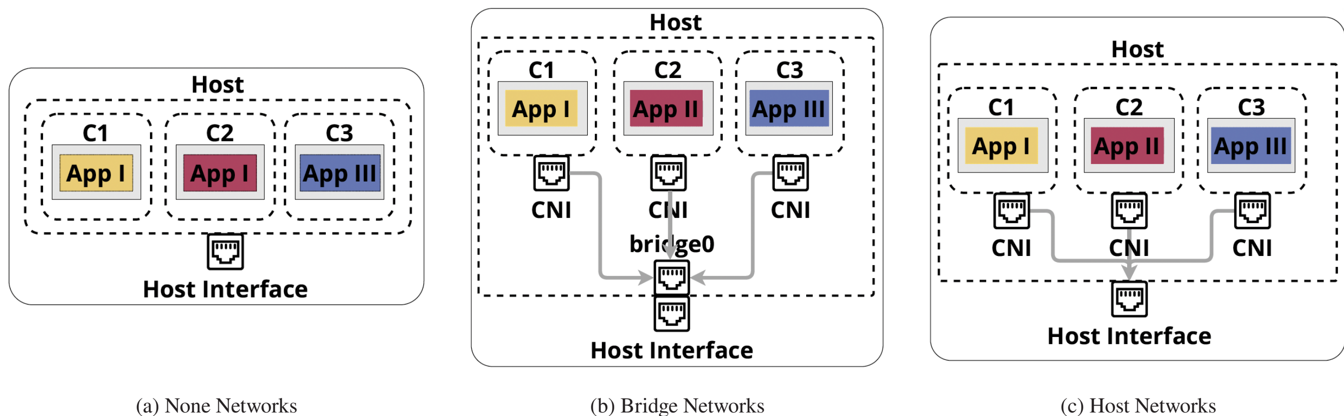
In this work, the performance of containerized services is examined in a wired connection, taking into account the delays that a wireless network may introduce. This approach is essential for delay-sensitive services, such as VoIP services, which require delays below 150 ms. A more detailed exploration of this approach is reserved for future work.

### B. Physical and logical configurations

Each Raspberry Pi board uses USB-Ethernet adapters from the Realtek brand, model RTL8152, with up to l00baseT/Full-Duplex capacity. This is done to expand the number of physical ports available for the routing service. There are 4 Ethernet interfaces, 1 WLAN interface, and five virtual WLAN interfaces. These interfaces are associated with both an IPv4 address of a containerized service and a monitoring application, as shown in Figure 5, representing the logical distribution of the implementation on the Raspberry Pi board. Figure 6, illustrates the topology for the joint implementation through Docker Compose. The Access Point function is also used on one of the Raspberry Pi boards using Hostap software. This is done to establish a wireless connection for network monitoring. The SSH remote access service installs the board's dependencies, access, and configuration. This enables wireless access to the equipment's configuration or through any available Ethernet cable.

### C. Base images for docker containers

The base images for Docker containers are based on Alpine Linux, the recommended choice for conserving storage resources in both the resulting images and container instances, as shown in [23]. As for service images, Nginx, Asterisk, and FRRouting are already available as dedicated images in the ARM32v7 architecture. Therefore, instead of building these services completely, files based on these images are generated to leverage their functionality.




(a) None Networks (b) Bridge Networks (c) Host Networks

**Fig. 4.** Docker networks: None, Bridge, and Host Networks, based on [18].

### D. Storage

Two types of volumes are used for storage: Named Volumes and Bind Mounts. Named Volumes are used to maintain the persistence of logs from containerized services and to share them with other containers. On the other hand, bind mounts are used to configure container files directly from the host.

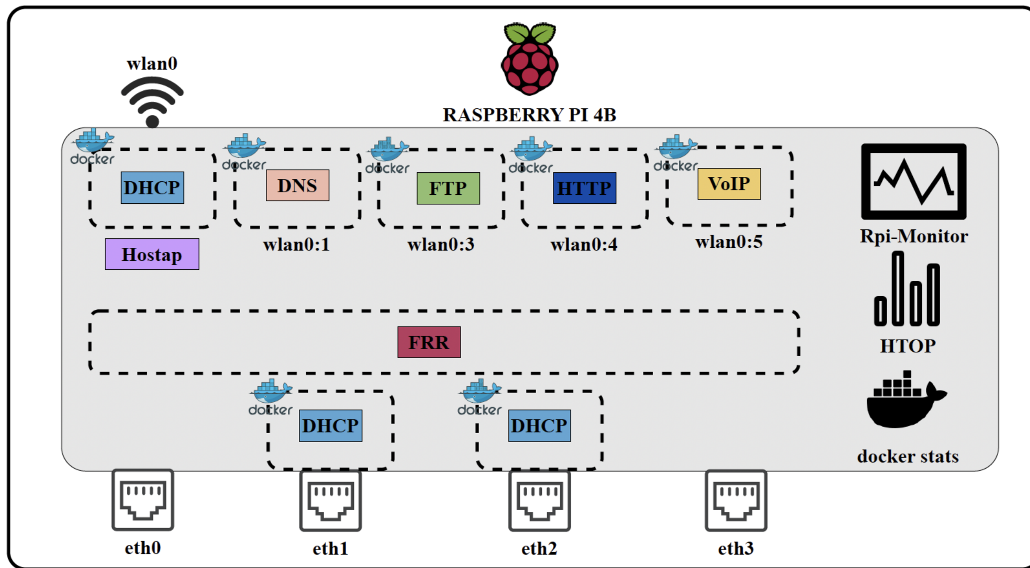



**Fig. 5.** Design: Logical distribution of Docker Containers on the Raspberry Pi board.

### *E. Docker networks*

Regarding Docker networks, the 'host' type network is used exclusively. This choice is based on previous considerations about Docker network performance and aims to optimize the implementation's performance.

### *F. Measurement tools, metrics ant testing setup*

For the performance analysis of containerized services, various general performance parameters are considered, such as CPU usage, memory usage, CPU load average, network traffic in and out, and a physical parameter, CPU temperature, for each containerized service. To perform this task, two computing devices are used to test the services: i) One to evaluate the service's operation and ii) A second to monitor the host's performance. Detailed descriptions of these devices can be found in Table I.

Client 2, acting as a monitoring device in the topology shown in Figure 6, connects wirelessly via SSH to collect these metrics using various software tools. Among them is the 'docker stats' command from Docker, which measures CPU and memory usage performance, and 'Htop' as a tool for visualizing CPU and RAM system resources and processes. Additionally, the RPI-Monitor tool, a monitoring software based on a web interface, is employed for system usage statistics visualization on Raspberry Pi devices. This allows access to performance metric data such as CPU load average, memory usage, and additional parameters like temperature.

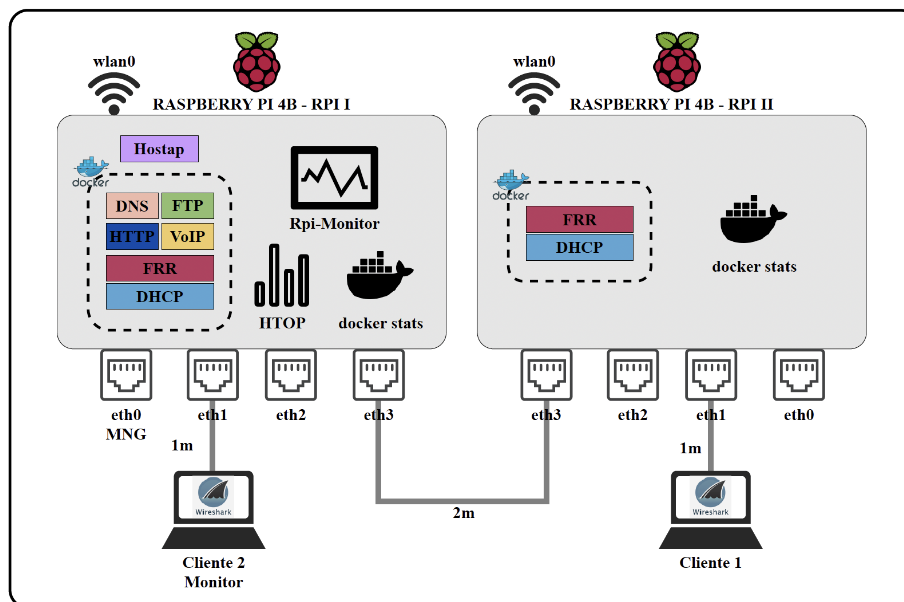



**Fig. 6.** Design: Container-Based Service Deployment Topology using Docker Compose.

TABLE I
HARDWARE USED FOR THE TEST SCENARIO.

| Model | Processor | Ram Memory | Operative System |
|---|---|---|---|
| LENOVO IdeaPad S410p | Intel(R) Core(TM) i5-4200 de 64bits CPU@1.6GHz | 8 GB | Windows 10 Pro |
| HP 15-ef1xxx | AMD Ryzen 5 4500U de 64bits CPU@2.38GHz | 8GB | Windows 11 Home |
| Raspberry Pi 4B | Broadcom BCM2711 Cortex-A72 (ARM v8) de 64bit @1.8GHz | 4GB | Debian 11 (Bullseye) |

Furthermore, it is essential to consider the analysis of services such as VoIP and assess Quality of Service (QoS) parameters like delay and jitter [24], For this purpose, Wireshark software is used on each device in Figure 6, allowing the capture and analysis of data packets based on SIP and RTP protocols. This provides a deeper insight into the quality of real-time communication. For the measurement process, measurements are taken during specific periods for each containerized service, and then the data is tabulated according to the respective measurement period. The data from each service's test period is used and averaged in the case of results obtained through the RPI-Monitor tool. For data obtained through HTOP and Docker stats, the maximum values for each metric are taken, as visualized in the open SSH terminals during the test period.

### G. Service software

The network services implemented in Docker containers are based on Linux servers following a client-server architecture. In this context, each service incorporates a daemon that runs and manages the services according to predefined configuration parameters. Below are the details of each software, its configuration, and the type of implementation in containers:

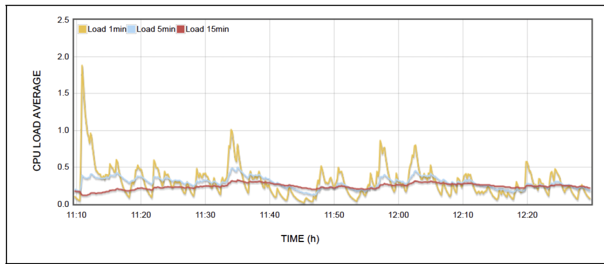
- Domain name resolution service: It uses a server based on the Internet Systems Consortium's Berkeley Internet Name Domain (Bind9), one of the most popular DNS servers in Linux distributions. This server acts as a master server, locally hosting primary DNS zone records and responding to pre-configured name resolution requests for each containerized service and the monitoring service.
- Addressing service: It is based on the Internet Systems Consortium DHCP (ISC DHCP) server, widely used for IP address assignment and network configuration. The implementation follows a basic configuration of IPv4 address assignment to DHCP clients, providing the addresses of the containerized DNS server and the gateway address configured for each of the Raspberry Pi's Ethernet ports.
- File transfer service: It uses the Very Secure File Transfer Protocol (VSFTP) server, which allows secure and efficient file or directory transfers. The implementation uses active mode and supports custom file transfers for configured local users. These users are within a chroot environment containing a 64KB file and 70KB of storage space for file transfers with the FTP client.
- Web service: It uses a server based on Nginx, known for its high performance, scalability, and low resource consumption [25]. The web service implements a default web server along with two virtual web servers based on Nginx virtual blocks, allowing the hosting of multiple web pages. These pages are accessed along with the aforementioned containerized DNS service.
- VoIP service: It is based on the Sangoma Asterisk server, an open-source framework under the GPL license that enables the development of real-time multiprotocol communication applications, such as high-quality voice and video applications. This service implements a PBX server to configure and manage VoIP extensions. Four IP phone station extensions are configured, of which two are used for functional tests between two clients connected to the Raspberry Pi's Ethernet ports to evaluate the containerized VoIP service without the limitations of wireless connections.
- Routing service: It is based on the open-source FRRouting software, providing traditional router functionality. For the current implementation, the OSPF protocol is one of the most widely used routing protocols today. This protocol is the routing system between the Raspberry Pi boards, enabling connection to the services.

## IV. RESULTS

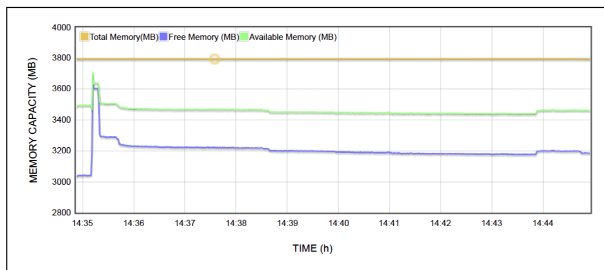
### A. CPU and Memory Performance Analysis

In this section, are presented the results obtained from the implementation using Docker Compose. As initial indicators of the results, data collected through the 'Rpi-Monitor' tool are obtained, which display CPU utilization statistics in Figure 7a, memory usage in Figure 7b, and CPU temperature in Figure 7c, over an entire testing interval for each of the containerized services. For the deployment using Docker Compose, the CPU load average indicator starts with an initial value of 1.88 (47.08 %) at the host's startup. Subsequently, this load decreases to 0.37 (9.25 %). From this point onwards, the network service tests are initiated and divided into sections as detailed below.

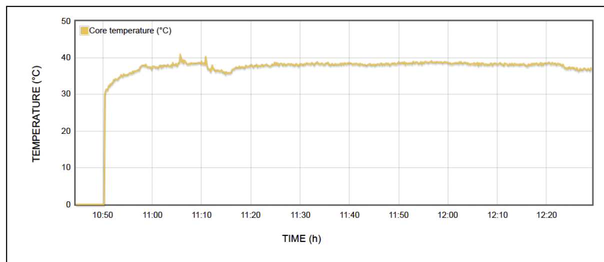
- Section I (11:30 am - 11:40 am) - Routing: During the connection between the RPI devices, a maximum load value of approximately 1.0 (25 %) can be observed. This value corresponds to the initial OSPF routing process between the devices.
- Section II (11:45 am - 11:50 am) - End-to-end Connection: When starting the end-to-end connection tests between the clients. During this period, the maximum load value reached is 0.518 (12.95 %).
- Section III (11:52 am - 11:53 am) - Connection to servers: The maximum load reached during the tests for connections to the containerized servers is 0.5 (12.5 %).
- Section IV (11:53 am - 11:54 am) - DNS: In this section, the load reaches values of 0.44 (11 %).




(a) Resource Usage: CPU Load Average.




(b) Resource Usage: Memory Usage.




(c) Resource Usage: Temperature.

**Fig. 7.** Resource Usage Using RPI-MONTTOR for Docker Compose.

- Section V (11:59 am - 12:00 pm) - HTTP: The load reached during the access to the HTTP pages obtains a maximum value of 0.863 (21.575 %). Additionally, according to the DNS traffic I/O graphs analysis, the traffic increases considerably during the HTTP testing period and remains elevated after this test.
- Section VI (12:00 pm - 12:04 pm) - FTP: During the tests of the 2 FTP connections, maximum load values of 0.801 (20.025 %) are recorded for client 1.
- Section VII (12:04 pm - 12:22 pm) - VoIP: During the VoIP tests, it is observed that during the call from extension 2001 to extension 2002, the CPU load reaches its peak, reaching 0.413 (0.33 %). The load remains low for the rest of the calls at an average of 0.180 (4.5 %).

Figure 7b shows the system memory usage, which remains constant as the tests run. The available free memory starts at 3 280 MB during the host's startup and remains close to 3 176.5 MB. This reflects a constant memory usage of approximately 617.8 MB, concerning a total of 3 794.32 MB of available memory.

In Figure 7c the CPU temperature can be observed, which goes from 35.95°C at startup and increases to maintain an average temperature of 38.6°C, with a maximum of 39.2°C during the HTTP tests. After the VoIP tests, it drops to 36.7°C.

The second performance indicator obtained through the 'docker stats' and 'htop' tools is detailed in Table II. The DNS service stands out with a maximum CPU usage of 15.02 %, followed by the FTP service with usage of up to 11.89 %, and the VoIP service with 5.18 %. The other containerized services show relatively low values, all below 1 %.

Regarding memory usage, the DNS service also has the highest value, with 1.1 %, followed by the VoIP service, with 0.8 %. It is important to note that these memory values are constant for all services during the Docker Compose implementation.

### *B. Quality of Service Analysis*

The analysis of RTP traffic provides information about the QoS of the VoIP service, as detailed in Figure 8. The results for VoIP calls are similar to those implemented previously, including:

- Payload codec: g711U vocoder.
- Packet loss: 0 % packet loss for all calls.
- Delays: Minimal delays are recorded, with slightly higher values for client 1 than for client 2. The averages of these values are 18.39 ms for client 1 and 7.63 ms for client 2. The average delay is 19.99 ms for both clients, with maximum delays of 21.74 ms for client 1 and 32.87 ms for client 2.
- Jitter: Regarding jitter, values vary significantly for client 2, where they can reach up to 7, while client 1 experiences jitters of up to 1.46.

## V. CONCLUTIONS

- Implementing network services through containers allows for effective deployment on systems with limited CPU, memory, and storage resources, such as Raspberry Pi boards. As demonstrated in services like HTTP and VoIP, these instances achieved a final product, a web page, or a voice call without significant degradation in service quality and with optimal resource usage. For example, three web pages were loaded without errors in the implementation using Docker CLI for HTTP, with CPU usage as low as 0.47 % and memory consumption as 0.1 %. Regarding the VoIP service, calls were made without distortions, delays, or audio loss, maintaining CPU usage at 5.18 % and memory consumption at 0.8 %.
- The implementation of network services through containers has shown minimal resource usage in most cases. Services like DHCP, HTTP, and Routing show zero CPU and memory consumption. In contrast, services like DNS, FTP, and VoIP show high consumption. This can be partly explained by factors such as the volume of request traffic, which, in the case of DNS, was considerably higher than other services. It can also be due to the transfer of information, as in the case of FTP. Additionally, their position in the network architecture should be considered. This is because their location may imply implicit involvement in other services, which, in turn, can increase resource consumption. This is evident in the case of the DNS service when used implicitly to support web services when making domain name resolutions to access a web page.

TABLE II
RESOURCE USAGE: CPU USAGE VIA DOCKER STATS AND ETOP

| Server | Docker Stats - Max. CPU Usage (%) | HTOP - Max CPU Usage (%) | HTOP - Max. RAM Memory Usage (%) |
|---|---|---|---|
| DHCP WlanO | 0.11 | 0.0 | 0.2 |
| DHCP Ethl | 0.11 | 0.0 | 0.2 |
| DHCP Eth2 | 0.11 | 0.0 | 0.2 |
| DNS | 15.02 | 7.3 | 1.1 |
| FTP | 11.89 | 8.5 | 0.0 |
| HTTP (www, web 1, web2) | 0.47 | 0.7 | 0.1 |
| VoIP | 5.18 | 5.3 | 0.8 |
| Routing | 0.12 | 0.10 | 0.1 |

| Source Address | Source Port | Destination Address | Destination Port | SSRC | Start Time | Duración | Payload | Paquetes | Lost | Min Delta (ms) | Mean Delta (ms) | Max Delta (ms) | Min Jitter | Mean Jitter | Max Jitter | Estado |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 192.168.5.10 | 50451 | 192.168.0.6 | 10076 | 0x1b6efc54 | 2023-08-16 12:18:25.110 | 19.84 | g711U | 993 | 0 (0.0%) | 18.385000 | 19.999764 | 21.690000 | 0.002938 | 0.385861 | 0.581178 | |
| 192.168.5.10 | 60524 | 192.168.0.6 | 10048 | 0x40b37d24 | 2023-08-16 12:16:25.884 | 20.86 | g711U | 1044 | 0 (0.0%) | 17.743000 | 19.999598 | 22.879000 | 0.000562 | 0.499972 | 0.931110 | |
| 192.168.5.10 | 52930 | 192.168.0.6 | 10050 | 0x9a9f8eac | 2023-08-16 12:15:32.634 | 20.72 | g711U | 1037 | 0 (0.0%) | 17.342000 | 19.999571 | 22.819000 | 0.040146 | 0.805904 | 1.467908 | |
| 192.168.5.10 | 56019 | 192.168.0.6 | 10010 | 0xffb25126 | 2023-08-16 12:14:09.019 | 20.34 | g711U | 1018 | 0 (0.0%) | 18.604000 | 20.000493 | 21.655000 | 0.002625 | 0.415269 | 0.597000 | |
| 192.168.5.10 | 62923 | 192.168.0.6 | 10066 | 0xb3d2565e | 2023-08-16 12:13:09.791 | 21.16 | g711U | 1059 | 0 (0.0%) | 18.578000 | 19.998974 | 21.740000 | 0.002000 | 0.364309 | 0.554573 | |
| 192.168.5.10 | 59881 | 192.168.0.6 | 10048 | 0x8cbcc209 | 2023-08-16 12:12:52.297 | 0.26 | g711U | 14 | 0 (0.0%) | 19.032000 | 19.940923 | 20.249000 | 0.001875 | 0.040590 | 0.115222 | |
| 192.168.5.10 | 50420 | 192.168.0.6 | 10080 | 0xbdac85a1 | 2023-08-16 12:11:37.273 | 20.46 | g711U | 1024 | 0 (0.0%) | 13.803000 | 19.999960 | 26.110000 | 0.029500 | 0.891532 | 1.422795 | |

(a) VoIP Service: RTP Details of Calls for Client 1

| Source Address | Source Port | Destination Address | Destination Port | SSRC | Start Time | Duración | Payload | Paquetes | Lost | Min Delta (ms) | Mean Delta (ms) | Max Delta (ms) | Min Jitter | Mean Jitter | Max Jitter | Estado |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 192.168.2.10 | 60568 | 192.168.0.6 | 10012 | 0x505bdcb0 | 2023-08-16 12:18:27.916 | 20.00 | g711U | 1001 | 0 (0.0%) | 6.581000 | 19.999271 | 33.191000 | 0.698750 | 4.465511 | 7.288791 | |
| 192.168.2.10 | 57875 | 192.168.0.6 | 10054 | 0x37974df9 | 2023-08-16 12:16:29.002 | 20.64 | g711U | 1033 | 0 (0.0%) | 8.617000 | 19.997191 | 29.940000 | 0.164649 | 1.112223 | 3.607699 | |
| 192.168.2.10 | 62222 | 192.168.0.6 | 10004 | 0x4db8d0e0 | 2023-08-16 12:15:35.454 | 20.86 | g711U | 1044 | 0 (0.0%) | 7.820000 | 20.002165 | 33.089000 | 0.088750 | 1.071696 | 2.909141 | |
| 192.168.2.10 | 58833 | 192.168.0.6 | 10042 | 0x77ee67f | 2023-08-16 12:14:12.193 | 20.04 | g711U | 1003 | 0 (0.0%) | 6.958000 | 20.002512 | 32.637000 | 0.026938 | 4.760490 | 7.429664 | |
| 192.168.2.10 | 53212 | 192.168.0.6 | 10052 | 0x1c2d110f | 2023-08-16 12:13:12.568 | 21.30 | g711U | 1066 | 0 (0.0%) | 16.325000 | 20.004633 | 24.366000 | 0.123213 | 0.706344 | 1.245788 | |
| 192.168.2.10 | 58674 | 192.168.0.6 | 10062 | 0xd04f3acd | 2023-08-16 12:11:40.761 | 19.85 | g711U | 994 | 0 (0.0%) | 7.431000 | 19.992930 | 33.104000 | 0.209277 | 1.957921 | 4.742771 | |

(b) VoIP Service: RTP Details of Calls for Client 2

**Fig. 8.** VoIP Service: SIP Call Details for VoIP Calls.

- The architecture of containerized services also significantly influences their performance. This is because the underlying software has a variety of architectures to offer the service. Some services, like DHCP, DNS, and FTP, adopt a simple client-server architecture based on daemons and configuration files. Others, like HTTP, VoIP, and Routing, have more complex architectures with dedicated modules to provide the service. For example, the HTTP service, based on NGINX, which allows for efficient deployment without containerization, highlights the importance of decentralization, even when dealing with a single application. Its structure consists of modules with internal management that favors efficiency. In contrast, services like VoIP and Routing have architectures that can be more challenging regarding management and performance.
- Containerizing services provide a high level of scalability, allowing the deployment of multiple containers to provide versatile, flexible, and efficient services. An example is the DHCP service, where multiple containers based on the same image are deployed. This provides a highly flexible addressing service with multiple configurations available for deployment. This achievement is partly due to using environment variables, which allow modifying a containerized service without directly modifying it. This creates a scalable, dynamic, and adaptable service that can be efficiently offered to users, as determined by CPU usage and memory consumption.
- Docker is a highly versatile tool that brings significant benefits to the deployment of network services. Its diverse ecosystem encompasses essential plugins, such as storage through volumes and bind mounts. In this context, it has been observed that these mechanisms allow interaction with a container's file system, making it easier to modify and configure a service without the need to directly access the containerized environment or altogether remove the isolated environment to make changes. Additionally, it has been found that rebuilding containers using volumes is a valuable mechanism for migrating services. An example of this is the use of containers to store configuration logs, as in the case of routing services with 'daemon' and 'zebra.config' files or in the case of the DHCP service with lease records in 'dhcpd.lease'. These records allow replicating the same configurations in other containers, ensuring the continuity of the service. However, it is important to note that logs stored via volumes can be prone to corruption. This is because, over time, these files

can begin to record incomprehensible strings of characters, affecting the service's operation.

- Another highly versatile tool for service deployment is network controllers through network objects. These networks enable the configuration and modification of service management and communication, providing essential features such as network isolation, traffic control, access, and scalability. This flexibility is evident when implementing services with 'bridge'-type networks. In this configuration, services were mapped to specific ports to allow external access through the container. As observed, using ports different from the default value for Network Address Translation (NAT) between the container and the external network provided greater isolation and security for the service. It also allowed the incorporation of multiple containers offering the same service but mapped to different ports to receive the service. This is especially useful in services like HTTP, where high scalability is achieved using a different port for each container. However, in services like passive FTP and VoIP, port mapping involves maintaining constant NAT over a range of ports (FTP: for transfer, random port >1 048, and VoIP: audio transmission via RTP, random port between 10 000 and 20 000). In these cases, considering a NAT that covers the entire port range to obtain service connection on a non-specific port was impractical as it affects performance, as mentioned in [19], [20]. For this reason, using 'host'-type networks, which remove the isolation level between containers and share the network with the host, allowed for services with low resource usage but limited scalability.
- Regarding the containerized VoIP service, significantly high performance and QoS are observed when using Ethernet cable transmission. This is confirmed through RTP parameters captured with Wireshark, where the average delay of calls is 20 ms, compared to the maximum allowable delay of 150 ms, and maximum Jitter values of 7 ms between both implementations, staying below the maximum allowable of 50 ms. In summary, containerizing VoIP services for end-to-end calls through wired connections does not affect this QoS.

## REFERENCES


[1] A. Khan. Key Characteristics of a Container Orchestration Platform to Enable a Modem Application. Vol. 4. 2017, pp. 42-48. Available: doi: 10.1109/MCC.2017. 4250933.

[2] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi. Performance Comparison Between Container-Based and Vm-Based Services. Institute of Electrical and Electronics Engineers Inc., 2017, pp. 185-190. isbn: 9781509036721. Available: doi: 10.1109/ICIN.2017.7899408.

[3] Z. Kozhirbayev and R. O. Sinnott. A Performance Comparison of Container-Based Technologies for the Cloud. Vol. 68. North-Holland, 2017, pp. 175-182. Available: doi: 10.1016/J.FUTURE.2016.08.025.

[4] A. Bhardwaj and C. R. Krishna. Virtualization in Cloud Computing: Moving from Hypervisor to Containerization — A Survey. Vol. 46. Springer Science and Business Media Deutschland GmbH, 2021, pp. 8585-8601. Available: doi: 10.1007/s13369-021-05553-3.

[5] V. G. da Silva, M. Kirikova, and G. Alksnis. Containers for Virtualization: An Overview. Vol. 23. Walter de Gruyter GmbH, 2018, pp. 21-27. Available: doi: 10. 2478/acss-2018-0003.

[6] V. Singh and S. K. Peddoju. Container-based Microservice Architecture for Cloud Applications. 2017. ISBN: 9781509064717. Available:

[7] S. Sultan, I. Ahmad, and T. Dimitriou. Container Security: Issues, Challenges, and the Road Ahead. Vol. 7. Institute of Electrical and Electronics Engineers Inc., 2019, pp. 52976-52996. Available: doi: 10.1109/ ACCESS.2019.2911732.

[8] E. Casalicchio and S. lannucci. The State-of-the-Art in Container Technologies: Application, Orchestration and Security. Vol. 32. John Wiley and Sons Ltd, 2020. Available: doi: 10.1002/cpe.5668.

[9] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar. Securing Docker Containers from Denial of Service (DoS) Attacks. Institute of Electrical and Electronics Engineers Inc., 2016, pp. 856-859. isbn: 9781509026289. Available: doi: 10.1109/SCC.2016.123.

[10] C. C. Chen, M. H. Hung, K. C. Lai, and Y. C. Lin. Docker and Kubemetes. In *Industry 4.1: Intelligent Manufacturing with Zero Defects*. 2022. Vol. 1, pp. 169- 213. Available: doi: 10.1002/9781119739920.ch5.

[11] K. Kumar and M. Kurhekar. Economically Efficient Virtualization Over Cloud Using Docker Containers. Institute of Electrical and Electronics Engineers Inc., 2016, pp. 95-100. isbn: 9781509045730. Available: doi: 10.1109/CCEM.2016.24.

[12] S. Bhat. Understanding Docker Volumes. In *Practical Docker with Python: Build, Release, and Distribute Your Python App with Docker*. Berkeley, CA: Apress, 2022, pp. 105-132. isbn: 978-1-4842-7815-4. Available: doi: 10.1007/978-1-4842-7815-4_5.

[13] N. G. Bachiega, P. D. Souza, S. M. Bruschi, and S. D. Souza. Performance Evaluation of Container's Shared Volumes. Institute of Electrical and Electronics Engineers Inc., 2020, pp. 114-123. isbn: 9781728110752. Available: doi: 10.1109/ICSTW50294.2020.00031.

[14] Dockerinc. Networks Overview - Docker Documentation. Available: [Online]. Available: https://docs.docker. com/network/.

[15] Dockerinc. Disable Networking for a Container - Docker Documentation. Available: [Online]. Available: https://docs.docker.com/network/none/.

[16] Dockerinc. Use Bridge Networks - Docker Documentation. Available: [Online]. Available: https://docs.docker. com/network/bridge/.

[17] Dockerinc. Use Host Networking - Docker Documentation. Available: [Online]. Available: https://docs.docker. com/network/host/.

[18] R. Dua, S. K. Konduri, and V. Kohli. *Learning Docker Networking: Become a Proficient Linux Administrator by Learning the Art of Container Networking with Elevated Efficiency Using Docker*. 1st ed. Packt Publishing Ltd., 2016. Vol. 1, pp. 2-11. isbn: 9781785280955. Available:

[19] S. Kun, Z. Yong, C. Wei, and R. Jia. An Analysis and Empirical Study of Container Networks. Institute of Electrical and Electronics Engineers Inc., 2018, pp. 189-197. isbn: 9781538641286. Available: doi: 10.1109/INFOCOM.2018.8485865.

[20] L. L. Mentz, W. J. Loch, and G. P. Koslovski. Comparative Experimental Analysis of Docker Container Networking Drivers. Institute of Electrical and Electronics Engineers Inc., 2020, pp. 1-7. ISBN: 9781728194868. Available: doi: 10. 1109 / CloudNet51028 . 2020 . 9335811.

[21] A. Endara. Network Service on Containers. 2023. Available: [Online]. Available: https://github.com/AndresYE/Network_Service_on_Containers_a_Practical-Approach.

[22] A. Endara. Network Service on Containers - Docker Hub. 2023. Available: [Online]. Available: https://hub.docker.eom/u/andresye.

[23] J. Islam, E. Harjula, T. Kumar, P. Karhula, and M. Ylianttila. Docker Enabled Virtualized Nanoservices for LocalloT Edge Networks. 2019. ISBN: 9781728108643. Available:

[24] M. Mejia, C. Ortiz, W. Ramos, and L. Moscoso. Network Traffic Management in the Quality of Service 'QoS' WAN in Tambopata-Peru 2021. Vol. 28. 2022, pp. 300-318. Available:

[25] W. Kithulwatta, K. Jayasena, B. Kumara, and R. Rathnayaka. Performance Evaluation of Docker-based Apache and Nginx Web Server. In *2022 3rd International Conference for Emerging Technology (INCET)*. 2022, pp. 1-6. Available: doi: 10.1109/INCET54531. 2022.9824303.