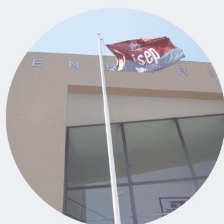
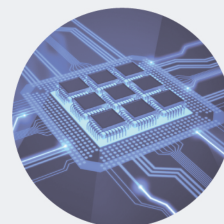
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA DE SISTEMAS COMPUTACIONAIS CRÍTICOS

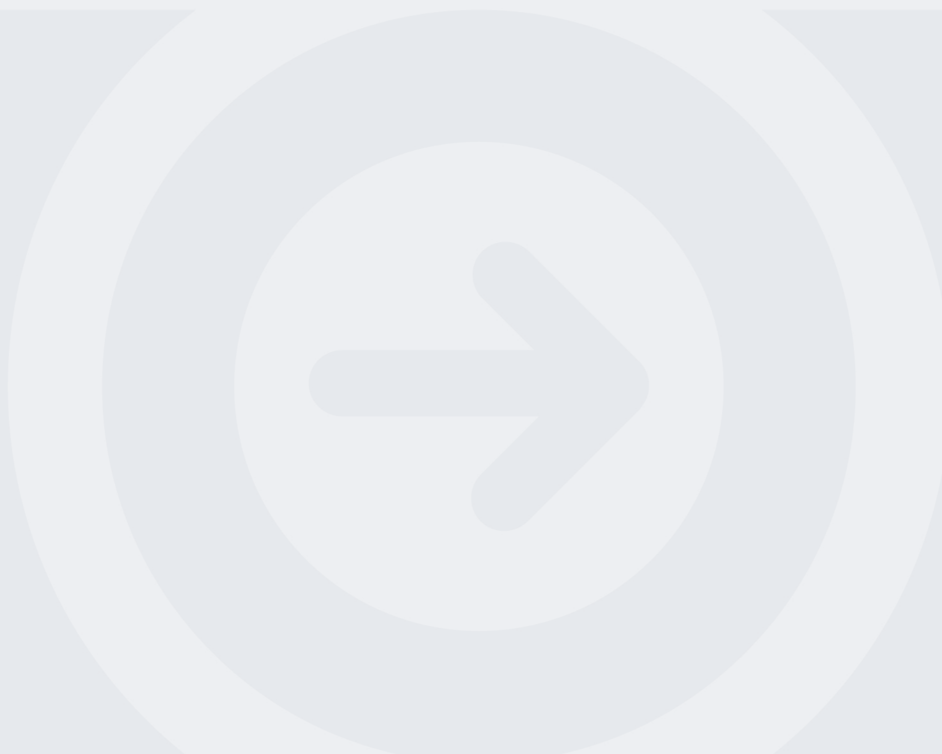




Intelligent Edge Orchestration of Real-Time Applications

RAFAEL CALAI

outubro de 2024



P.PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Intelligent Edge Orchestration of Real-Time Applications

Rafael Calai

Dissertation submitted in partial fulfilment of the requirements for the Master's degree in Critical Computing Systems Engineering

Supervisor: António Manuel de Sousa Barros
Co-Supervisor: Luis Miguel Rosário da Silva Pinho

Evaluation Committee:
President:
Luís Miguel Moreira Lino Ferreira, Professor Coordenador, Instituto Superior de Engenharia do Porto
Members:
Ricardo Augusto Rodrigues da Silva Severino, Professor Adjunto, Instituto Superior de Engenharia do Porto
António Manuel de Sousa Barros, Professor Adjunto, Instituto Superior de Engenharia do Porto

Porto, October 17, 2024

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

I declare that the work presented in this document is original and my own, and has not previously been used for any other purpose.

I further declare that I have fully followed the Code of Good Practices and Conduct of the Polytechnic Institute of Porto.

ISEP, Porto, October 17, 2024

Abstract

With the rapid development of embedded systems, new communication technologies, and distributed systems, the complexity of critical system development has increased. In the past, these systems typically only performed the function of controlling a specific process, without the need for communication or being components of a complex system centralizing data processing for decision-making. Nowadays, these critical systems have more tasks and are much more integrated, aiming to provide new functionalities to users, reduce costs, and enhance the security and availability of essential services. This scenario has brought new challenges to the development of such systems, including increased computational needs, new communication methods, ensuring cybersecurity, and meeting real-time requirements in distributed systems.

The central theme of the thesis will focus on the challenge associated with ensuring real-time requirements in distributed systems.To address this issue, we propose a scheduling algorithm for Docker Swarm, built on top of RT-Linux. This algorithm is designed to ensure that each process meets its temporal constraints by prioritising allocation according to their respective priorities and the resources available.

Keywords: Critical systems, distributed systems, real time, embedded systems.

Resumo

Com o rápido desenvolvimento dos sistemas embebidos, novos meios de comunicação e dos sistemas distribuídos, elevou a complexidade do processo de desenvolvimento de sistemas críticos. No passado, estes sistemas geralmente apenas desempenhavam a função de controlar um processo específico, sem a necessidade de comunicação ou de serem componentes de um sistema complexo que centralizasse o processamento de dados para tomada de decisão. Nos dias de hoje, esses sistemas críticos possuem mais tarefas e são muito mais integrados, visando proporcionar novas funcionalidades aos utilizadores, reduzir custos e aumentar a segurança e disponibilidade de serviços essenciais. Este cenário trouxe consigo novos desafios para o desenvolvimento deste tipo de sistemas, tais como: maior necessidade computacional, novos meios de comunicação, garantia de segurança cibernética e assegurar requisitos de tempo real em sistemas distribuídos.

O tema central da tese irá focar no desafio associado à garantia de requisitos em sistemas distribuídos. Para resolver esse problema, propomos um algoritmo de escalonamento para Docker Swarm, construído sobre RT-Linux. Este algoritmo foi projetado para garantir que cada o processo atende às suas restrições temporais, priorizando a alocação de acordo com seus respectivos prioridades e os recursos disponíveis.

Contents

List of Figures xiii

List of Tables xv

List of Source Code xvii

List of Acronyms xix

1 Introduction 1
1.1 Problem Statement . . . 1
1.2 Research Objectives . . . 2
1.3 Contributions . . . 2
1.4 Document Structure . . . 3

2 State of the Art 5
2.1 Safety Critical Systems . . . 5
2.2 Distributed Systems . . . 5
2.2.1 Cloud Computing . . . 6
2.2.2 Fog Computing . . . 7
2.2.3 Edge Computing . . . 8
2.3 Related Works . . . 9
2.3.1 Enabling Real-Time Container Orchestration . . . 9
2.3.2 Hierarchical Resource Orchestration Framework . . . 10
2.3.3 Elastic Software Architecture for Extreme-Scale Big Data Analytics 11

3 Technologies 13
3.1 Containerization . . . 13
3.2 Docker . . . 13
3.2.1 Docker vs VM . . . 14
3.2.2 Architecture . . . 14
The Docker Daemon . . . 15
The Docker Client . . . 15
Docker Desktop . . . 15
Docker Registries . . . 15
Docker Objects . . . 16
Images . . . 16
3.3 Orchestration . . . 16
3.4 Docker Swarm . . . 17
3.5 Docker Engine API . . . 18
3.5.1 Python SDK . . . 18
3.5.2 Docker Client Class . . . 18

3.5.3 Container Class . . . 19
3.5.4 Service Class . . . 19
3.6 Scheduling Algorithms . . . 20
3.6.1 Fixed Priority Scheduling Algorithms . . . 20
3.6.2 Dynamic Priority Scheduling Algorithms . . . 21

4 Requirements and System Architecture 23
4.1 Requirements . . . 23
4.1.1 General Scheduling Requirements . . . 23
4.1.2 Service Management . . . 24
4.1.3 Network Communication . . . 24
4.1.4 Configurability and Node Management . . . 24
4.1.5 Event Handling . . . 25
4.2 System Architecture . . . 25
4.2.1 Manager Node . . . 25
4.2.2 Worker Node . . . 26

5 Development 29
5.1 Hardware Environment . . . 29
5.1.1 Raspberry Pi . . . 29
5.1.2 Network Configuration . . . 31
5.2 Docker Swarm Cluster . . . 32
5.3 Pause/Unpause Service . . . 32
5.4 Income Manager Service . . . 33
5.5 Swarm RT-Sched . . . 34
5.5.1 Service Request Thread . . . 35
5.5.2 Cluster Scheduler Thread . . . 36
Trigger Mechanism . . . 36
Handling the Scheduling Queue . . . 37
Load Balancing . . . 38
Service Creation . . . 39
Preemption Logic . . . 40
Service Management . . . 40
Concurrency Control . . . 40
5.5.3 Service Monitor Thread . . . 40
5.5.4 Remove Service Thread . . . 41
5.5.5 Remove Service Server Thread . . . 41
5.6 Client Simulator Script . . . 41

6 Experimental Results 43
6.1 Schedulable Task-set . . . 43
6.2 Non-Schedulable Task-set . . . 46
6.3 Worker Node Hardware Fault . . . 48
6.4 Soak Test . . . 50
6.5 Service Resources Utilization . . . 50
6.6 Manager Node Resource Utilization . . . 52

7 Conclusions 55
7.1 Practical and Theoretical Implications . . . 55
7.2 Limitations . . . 56

7.3 Future Work 57

Bibliography **59**

List of Figures

2.1 Distributed systems . . . 6
2.2 Non-functional requirements . . . 8
2.3 REACT architecture [8]. . . . 9
2.4 Hierarchical Resource Orchestration Framework architecture [9]. . . . 11

3.1 Docker vs Virtual Machine (VM) [14]. . . . 14
3.2 Docker architecture. [15]. . . . 15

4.1 Docker Swarm cluster. . . . 25
4.2 Manager node. . . . 26
4.3 Worker node. . . . 27

5.1 Raspberry Pi 5 [26]. . . . 30
5.2 Ethernet connection diagram. . . . 31
5.3 Docker Swarm nodes. . . . 32
5.4 Pause/unpause service list. . . . 33
5.5 Miss deadline task request. . . . 35

6.1 Schedulable task-set. . . . 45
6.2 Start schedulable task-set log. . . . 46
6.3 Resource utilization task-set. . . . 51
6.4 Resource utilization service without limitation. . . . 52
6.5 Resource utilization Manager node in idle. . . . 53
6.6 Run-time resource utilization Manager node. . . . 53

List of Tables

4.1 General Scheduling Requirements 23
4.2 Service Management Requirements 24
4.3 Network Communication Requirements 24
4.4 Configurability and Node Management Requirements 24
4.5 Event Handling Requirements 25

6.1 Schedulable task-set 43
6.2 Non-Schedulable task-set 47
6.3 Request Without Response. 49
6.4 Denied task requests. 49
6.5 Stress test task-set 51

List of Source Code

5.1 Docker Swarm initialization command. 32
5.2 Docker Swarm join command. 32
5.3 Create service pause_unpause. 32
5.4 Code of pause_unpause script. 33
5.5 Function service response server. 34
5.6 Function cluster scheduler thread. 37
5.7 Function check schedulability. 38
5.8 Function pause lower priority service. 38
5.9 Function create service. 39
5.10 task-set JSON structure. 42

List of Acronyms

API	Application Programming Interface.
BE	Best Effort.
CA	Certificate Authority.
COMPS	COMP Superscalar.
CPU	Central Processing Unit.
DNS	Domain Name System.
EARS	Easy Approach to Requirements Syntax.
EDF	Earliest Deadline First.
HCBS	Hierarchical Constant Bandwidth Server.
HTTP	Hypertext Transfer Protocol.
IoMT	Internet of Medical Things.
IoT	Internet of Things.
IoV	Internet of Vehicles.
JSON	JavaScript Object Notation.
LAN	Local Areea Network.
LLF	Least Laxity First.
MCU	Microcontroller Unit.
MFC	Mobile Fog Computing.
MPU	Microprocessing Unit.
OS	Operating System.
QoS	Quality of Service.
RMS	Rate-Monotonic Scheduling.
RTOS	Real Time Operating System.
SDK	Software Development Kit.
SSH	Security Shell.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.

UI	User Interface.
VCPU	Virtual Central Processing Unit.
VM	Virtual Machine.

Chapter 1

Introduction

The rapid advancement of hardware capabilities and the increasing demand for dynamic and flexible applications have significantly heightened the complexity of critical distributed systems. Traditional centralized, self-contained applications are being replaced by distributed workflows that integrate components across different network nodes, necessitating innovative middleware, enhanced runtime environments, and sophisticated development methodologies. The rise of fog and edge computing has created a computing continuum that ranges from small Internet of Things (IoT) devices to large-scale cloud infrastructure. Despite these advancements, current commercial orchestration platforms, primarily designed for cloud computing, fall short of meeting the real-time requirements of critical systems that operate under strict time constraints.

This research focuses on enhancing the orchestration capabilities of Docker Swarm to ensure it meets the strict temporal requirements of distributed critical systems. By implementing a real-time scheduler that assigns priorities based on time constraints, this study aims to provide a robust solution that guarantees timely execution while optimizing resource allocation. The proposed solution addresses the limitations of existing orchestration platforms by incorporating real-time capabilities into Docker Swarm, ensuring reliable operation and compliance with critical deadlines.

1.1 Problem Statement

Despite the benefits of distributed computing, existing container orchestration platforms like Docker Swarm are primarily optimized for balancing computational load and maintaining system availability in cloud environments. These platforms are not designed to meet the strict timing requirements of critical systems that demand guaranteed real-time performance. As a result, current orchestration tools are inadequate for applications such as autonomous vehicles, healthcare systems, and industrial automation, where even minor delays can lead to severe consequences, including loss of life, financial losses, or significant operational failures.

Moreover, the challenges of integrating edge and fog computing into critical systems require new scheduling methods that can handle variable network latencies, dynamically changing workloads, and resource constraints while ensuring real-time performance. Without addressing these limitations, the potential of emerging technologies and distributed critical systems cannot be fully realized. This research addresses this gap by proposing a novel scheduling algorithm for Docker Swarm, enhancing its capabilities to support real-time applications effectively.

1.2 Research Objectives

The primary aim of this research was to develop and validate a real-time scheduling algorithm for Docker Swarm that enhances its ability to manage distributed critical systems effectively. The specific objectives of this research were to:

1. **Analyze the Characteristics of Critical Distributed Systems**: Conducted a comprehensive study of the unique requirements and challenges of critical distributed systems, including their real-time constraints, network latencies, and resource limitations.
2. **Evaluate Existing Technologies and Frameworks**: Surveyed and critically evaluated existing technologies and orchestration frameworks for distributed systems, focusing on their suitability for real-time applications and identifying their limitations.
3. **Investigate Related Research**: Reviewed the state-of-the-art in real-time container orchestration to identify gaps, derive insights, and build a foundation for designing a new solution.
4. **Design a Real-Time Scheduling Algorithm for Docker Swarm**: Developed a novel scheduling algorithm customized to Docker Swarm that incorporated real-time capabilities, ensuring that tasks were executed within their specified time constraints while optimizing resource allocation.
5. **Implement and Integrate the Proposed Algorithm**: Integrated the developed scheduling algorithm into Docker Swarm, creating a prototype system that could be deployed in real-world distributed environments.
6. **Perform Experimental Validation and Testing**: Conducted extensive testing and experimentation across multiple scenarios to validate the effectiveness, efficiency, and reliability of the proposed algorithm in meeting real-time requirements.
7. **Analyze Results and Identify Areas for Improvement**: Performed a thorough analysis of the experimental data to evaluate the algorithm's performance, identified its strengths and weaknesses, and suggested areas for enhancement.
8. **Propose Future Research Directions**: Based on the findings, recommended future research paths to further advance real-time capabilities in container orchestration and address unresolved challenges.

1.3 Contributions

This dissertation makes the following key contributions to society:

1. **Support for Emerging Technologies**: By enabling real-time capabilities in distributed systems at the edge, the study supports the deployment of emerging technologies like autonomous vehicles, smart cities, and industrial automation. These advancements can improve urban planning, traffic management, and public infrastructure, leading to more efficient, safer, and smarter environments.
2. **Improved Reliability of Critical Services**: The research enhances the reliability of critical real-time systems, such as those used in healthcare, transportation, and emergency response. By ensuring these systems meet strict deadlines, the risk of failures that could lead to loss of life, injury, or financial loss is significantly reduced, thereby contributing to public safety and well-being.

3. **Enhanced Cybersecurity Measures**: The proposed improvements to Docker Swarm's real-time orchestration provide better control over the allocation and execution of processes, reducing vulnerabilities and the potential for cyber-attacks. This contributes to a safer digital landscape, protecting sensitive data and critical infrastructure from malicious activities.
4. **Economic Impact**: By optimizing the use of resources in distributed systems, the research can help reduce operational costs for industries relying on critical systems, such as manufacturing, logistics, and telecommunications. Cost savings can be passed on to consumers, making advanced technologies more accessible to a broader audience.
5. **Environmental Benefits**: The improved efficiency in resource allocation and task execution reduces the energy consumption of distributed systems, particularly in edge and fog computing environments. This contributes to lower carbon emissions and supports global sustainability efforts by minimizing the environmental impact of data centers and other computational infrastructures.
6. **Empowerment through Innovation**: By providing a framework for enhancing real-time capabilities in edge computing, the research encourages innovation and empowers developers and engineers to create new solutions that address pressing societal challenges, from disaster management to precision agriculture.

1.4 Document Structure

This thesis is structured to provide a comprehensive exploration and validation of real-time container orchestration. Chapter 2 examines the state of the art, covering key concepts related to the thesis and reviewing relevant works and simulation frameworks that serve as a foundation for this study. Chapter 3 introduces the core technologies employed in developing the proof of concept algorithm, while Chapter 4 outlines the overall framework architecture, detailing how these technologies work together to enable real-time orchestration.

In Chapter 5, the focus shifts to the development of services and the setup of the hardware and software systems essential for the implementation. Chapter 6 then presents an in-depth analysis of the experimental results, demonstrating the effectiveness of the proposed approach. Chapter 7 concludes this dissertation with a summary of the key findings, a discussion of the limitations, and suggestions for future research directions.

Chapter 2

State of the Art

This chapter outlines key concepts and essential technologies needed for proposing an intelligent edge orchestration algorithm for real-time applications. Section 2.1 briefly introduces Safety Critical Systems, providing foundational insights. Section 2.2 explores distributed systems, cloud, fog, and edge computing. Lastly, Section 2.3 presents available papers related to real-time container orchestration. This structured approach sets the stage for the subsequent development and discussion of the intelligent edge orchestration algorithm in the following chapters.

2.1 Safety Critical Systems

Safety critical systems are systems that a failure could result in loss of life, financial losses or damage to the environment. There are many well-known examples in application areas such as aviation, medical devices, aircraft flight control, railway, air defence systems and nuclear systems. Nowadays many modern systems are becoming safety critical in a general sense because financial loss and even loss of life can result from their failure. Future safety critical systems will be more common, much more integrated, and more powerful. From a software perspective, developing safety critical systems in the numbers required and with adequate dependability is going to require significant advances in areas such as specification, architecture, verification and the software process. The very visible problems that have arisen in the area of information system security suggests that security is a major challenge as they have to communicate with other system and have to be integrated with cloud computing [1].

During the process of development a critical safety system a standard has to be followed. The IEC 61508 standard categorizes systems into four distinct groups based on the associated risks. Functional safety engineering is in charge of identifying specific hazardous failures which lead to serious consequences (e.g., death) and then establishing maximum tolerable frequency targets for each mode of failure. Equipment whose failure contributes to each of these hazards is identified and usually referred to as “safety related”. A safety function is thus defined as a function, of a piece of equipment, which maintains it in a safe state, or brings it to a safe state, in respect of some particular hazard [2].

2.2 Distributed Systems

A distributed system is a collection of computer programs that utilize computational resources across multiple, separate computation nodes to achieve a common, shared goal. Distributed systems aim to eliminate bottlenecks or central points of failure from a system. In typical distributed systems, latency is not a concern, making cloud computing the ideal

solution. However, in critical systems, distributed architecture must adhere to requirements of stability, security, availability, and predictability, as a failure in such systems can have severe consequences. In this context, fog computing emerges as an excellent choice, as it can ensure stringent timing constraints.

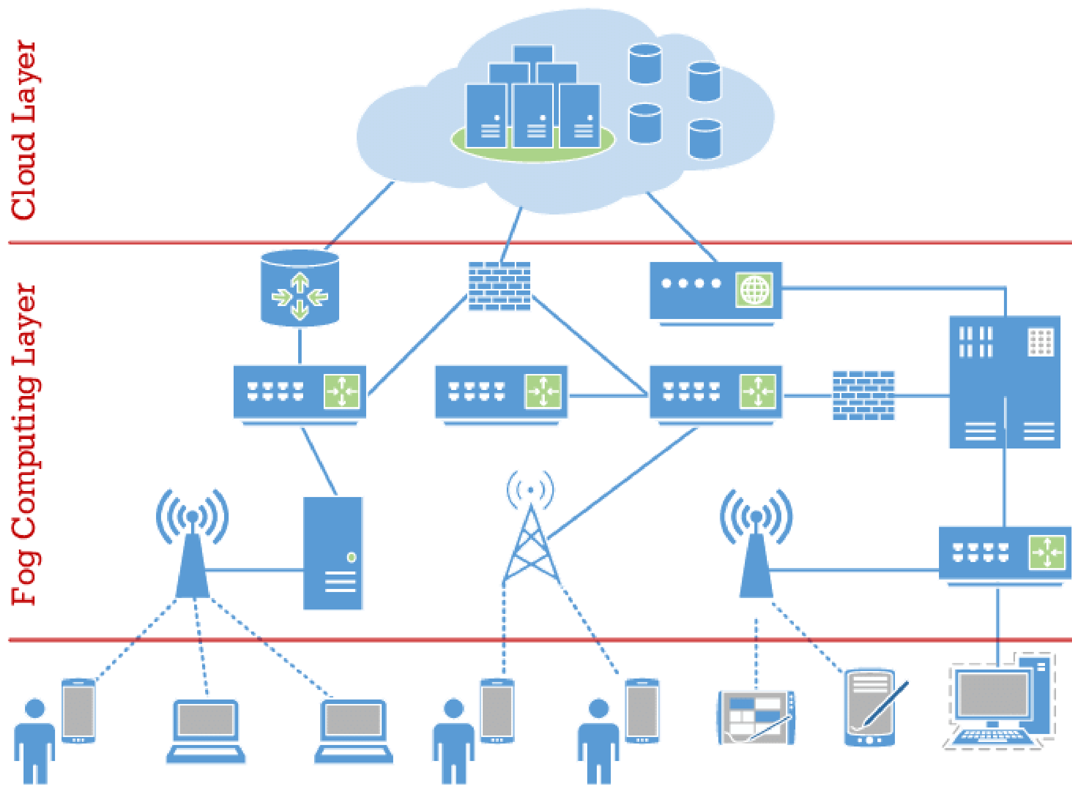


Figure 2.1: Distributed Systems [3].

According to [4] the architecture of a distributed system with cloud, fog and edge is represented in figure 2.1. Edge computing with IoT devices that have less computational power, Fog computing that processes, stores data, and communicates with Edge, and finally Cloud computing that has more computational capabilities but also introduces more latency and cost. This kind of architecture can provide low cost as edge devices don't need costly microprocessors and memory to store and process high amounts of data. Fog computing can provide processing power with controlled latency, and Cloud computing is able to handle the part of processing that is not time-critical and provide scalability to the system.

2.2.1 Cloud Computing

Cloud computing essentially involves the provision of computing services such as servers, storage, databases, networking, software, analytics, and intelligence over the internet (referred to as "*the cloud*"). This model enables faster innovation, flexible resource allocation, and economies of scale. Companies typically pay only for the specific cloud services they utilize, thereby helping to lower operating costs, optimize infrastructure efficiency, and scale in line with changing business needs [5].

In recent years, companies have increasingly chosen to migrate from traditional computing to cloud-based systems due to seven primary factors [5].

- Cost savings are achieved by eliminating initial investments in infrastructure;
- The speed at which new capabilities can be deployed and scaled up or down as needed is significantly enhanced;
- The global scale of cloud services enables efficient scaling of computational power tailored to each location's requirements;
- Productivity is increased as companies are freed from managing physical infrastructure, allowing them to focus more on revenue-generating activities;
- Cloud providers often leverage the latest technologies, leading to improved performance;
- Cloud-based solutions offer enhanced reliability through features such as data backup, disaster recovery, and business continuity, which are facilitated and made more cost-effective by mirroring data across multiple redundant sites on the provider's network;
- Security is bolstered by many cloud providers, who offer a wide range of policies, technologies, and controls to enhance overall security posture and safeguard data, applications, and infrastructure from potential threats;

2.2.2 Fog Computing

Fog computing was first introduced by Cisco with the purpose of extending the cloud capabilities closer to the edge of the network. This architecture distributes the tasks from the distant central management system in the cloud to the intermediate nodes (e.g. routers, switches, hubs, etc.), which contain some computational resources, the main goals are reduce the latency caused by transmitting messages between IoT devices and cloud, cost with cloud and reduce network utilization. Fog computing provides five basic mechanisms: storage, processing power, acceleration, networking, and control toward enhancing IoT systems in five subjects: security, cognition, agility, low latency, and efficiency. For example, in Internet of Vehicles (IoV) application, the central server can migrate the best route determination function from the cloud to the roadside fog nodes to assist the travel of the connected vehicles. As another example, in an outdoor-based Internet of Medical Things (IoMT) application, the hospital system can distribute the health measurement function and the alarm function to the user equipment in order to perform timely determination of the patient's health condition and to perform an alarm to catch the proximal passengers attention when the patient is having an incident [6].

In Fog computing there are two main topologies: Mobile Fog Computing (MFC). MFC brings numerous advantages to mobile IoT in terms of rapidness, ultra-low latency, substitutability and sustainability, efficiency, and self-awareness. However, the dynamic nature of MFC environment raises many challenges in terms of resource and network heterogeneity, the mobility of the participate entities, the cost of operation, handover and so forth. The second topology, the static fog computing frameworks designed for applications, such as the smart home or smart factory would not fully address the MFC-specific challenges because they have different perspectives from the involved entities and the topology. For example, a classic fog computing framework, which may involve a thin mobile client-side application for smartphone users, would not consider how to provide a reliable fog service to the high-speed moving vehicles. Moreover, the classic fog computing framework also would not consider how to provide a reliable fog service to vessels at sea where the telecommunication base stations are not available, and the satellite Internet is too expensive. [6]

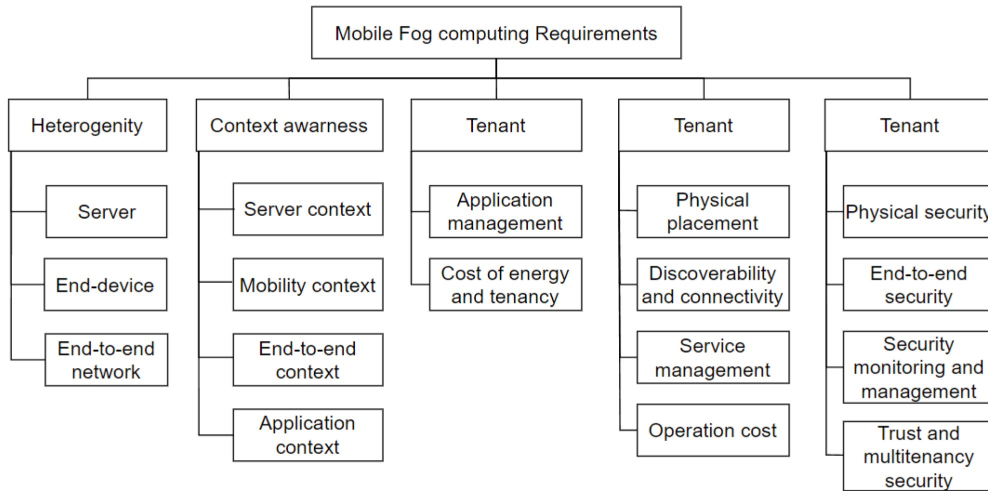


Figure 2.2: Non-functional requirements [6].

Figure 2.2 summarizes the non-functional requirements for MFC. In general, the top requirements are co-related one to another. For example, heterogeneity is a common factor that needs to be considered when the fog tenant plans to choose which fog server to use to deploy their applications. Similarly, context awareness, which represents the runtime factors, is also influencing the decision of when and where to deploy and execute the tasks. Furthermore, the tenant-side end-device or end-user (i.e. tenant-side clients) is influencing the decision of how the provider of fog servers manage the fog nodes. On the other hand, the Quality of Service (QoS) of the provider's fog nodes also influences the decision for the tenant to choose the right fog node for tenant-side clients. In addition, although all the four MFC domains involve the five aspects, the complexity level of each aspect in a different domain can be quite different. For example, an application scheduling scheme designed for a land vehicular fog computing may require significant adjustment when the developers intend to apply the scheme to , unmanned aerial vehicular fog computing because the heterogeneity level and the context factors are very different between the two domains.

2.2.3 Edge Computing

Edge devices are those low-level electronic devices, which operate in the physical world to complete tasks such as sensing, actuating, and controlling. Each edge device is logically controlled by one or more Microcontroller Unit (MCU), with each being a small computer running on a single integrated circuit. The low-level software interface programmed in the MCUs that provide controls to the device's hardware is known as firmware. All the functions, including sensing, controlling, and computing, are coded in the firmware and, therefore, handled by the MCUs. Edge devices can be further categorized as IoT devices and mobile devices. IoT devices are lightweight electronic devices that are interconnected or connected to the edge servers through wireless protocols such as 4G/5G, Wi-Fi, and even Bluetooth. They usually run on lightweight preemptive/cooperative Real Time Operating System (RTOS), e.g., FreeRTOS and Azure RTOS. Some examples of IoT devices include smart home devices, health monitoring devices, and smart warehouse carts in industrialized IoT.

Most of the manufacturers of tiny IoT devices adopt ARM architectures like Cortex-M series MCUs produced by ST Microelectronics, NXP, Nordic, Renesas, so on. Different from tiny IoT devices, mobile devices usually have more advanced and costly preemptive operating systems, e.g., Linux, Android and iOS, providing programmable interfaces for developers to code their own applications on the top of the operating system. Some examples of mobile devices include smartphones, tablets, and central controllers of smart vehicles. Most of the manufacturers of mobile devices adopt Cortex-A series Microprocessing Unit (MPU)s produced by high-performance chip manufacturers such as Qualcomm and NXP [7].

2.3 Related Works

In this section, three selected papers serve as the foundation for the thesis work, focusing on real-time containers. These chosen works have been carefully curated to provide a robust basis for the exploration and development undertaken in the thesis work.

2.3.1 Enabling Real-Time Container Orchestration

Struhár et al. introduce an orchestration system based on the master-minion architecture [8]. This framework comprises a master node and a cluster of minion compute nodes. The master node, serving as the system's core, is responsible for making global decisions about the cluster. It receives user requests for container deployments, monitors the states of compute nodes in the cluster continuously, and schedules containers on computing nodes. To prevent a single point of failure, the master node's functionality can be distributed across multiple physical machines.

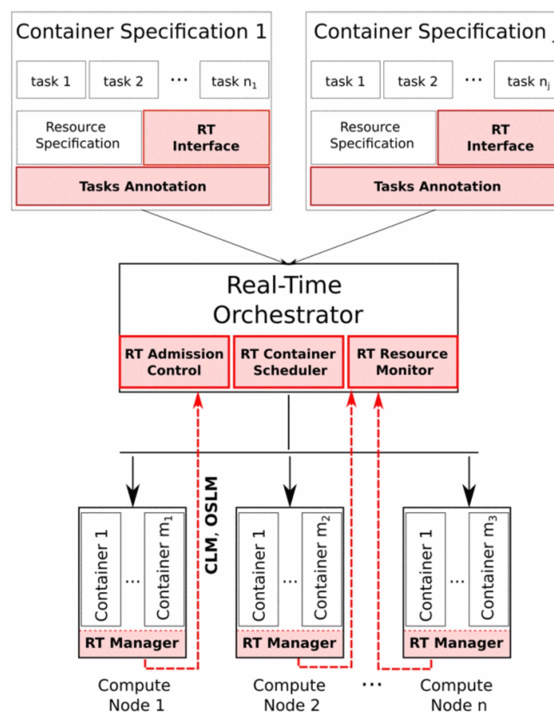


Figure 2.3: REACT architecture [8].

Figure 2.3 presents the architecture of the proposed orchestration system, where Container Specifications is the edge system with several tasks, RT interface and Task Annotation that

attributes tasks and their respective allocation time to the module RT Admission Control in Real-Time Orchestrator.

Real-Time Orchestrator has three modules.

- **RT Admission Control** This module receives tasks and their time requirements for assignment to compute nodes;
- **RT Container Scheduler** Responsible for allocating tasks received by the RT Admission Control to available compute nodes;
- **RT Resource Monitor** Monitors all nodes, compiling information and sending it to the RT Container Scheduler for decision-making.

Compute Nodes host containers and the RT Manager, which measures computational usage and detects when a task is complete, making the container available for the next task.

The system runs on top of a Debian Linux, Kernel 5.2. patched with Hierarchical Scheduling Patch the kernel is extended with a reservation-based scheduling policy in which each Virtual Central Processing Unit (VCPU) is assigned a quota and a period, bounding the execution of the VCPU to the respective quota in each time interval of length equal to the period. On the container level, the global SCHED_DEADLINE policy selects at each time instant the container with the earliest deadline, while at the task level, the SCHED_FIFO/SCHED_RR policy is used to schedule tasks within containers. Once there is no task to be scheduled by the SCHED_FIFO, other BE tasks, are executed via the default scheduling policy. This enables the co-existence of RT and Best Effort (BE) containers.

2.3.2 Hierarchical Resource Orchestration Framework

In the second selected paper, Struhár et al. propose a joint hierarchical container virtualization and orchestration framework to enhance the real-time behavior of containers in a multi-container environment [9]. The framework comprises two phases: an offline phase and an online phase. The offline phase addresses the deployment problem by determining the placement of containers and the configuration of their real-time interfaces based on design-time models. In contrast, the online phase aims to bridge the gap between deployment assumptions, relying on idealized models, and the dynamic behavior of containers that may impact runtime performance. This phase adjusts and allocates Central Processing Unit (CPU) resources among real-time and non-real-time best-effort containers, guided by continuous runtime evaluations of the real-time performance.

To ensure real-time capabilities for containers, the paper implemented the solution using the Hierarchical Constant Bandwidth Server (HCBS) patch. This implementation allows for the adjustment of resources at runtime, facilitating the online adaptation of real-time containers in Linux. The paper provides an evaluation demonstrating the feasibility of this solution.

Figure 2.4 depicts the framework, which comprises the offline phase and a three-level control architecture for the online phase, container-level, node-level, and cluster-level controllers. The offline phase is responsible for (a) computing the ideal real-time interfaces (an initial value of the resource reservation) for a given set of containers and their real-time requirements and (b) selecting a node for an incoming container. The three-level controller hierarchy mitigates temporal disturbances through continuous redistribution of system resources and migration of containers in the event of a shortage of resources on a computing node.

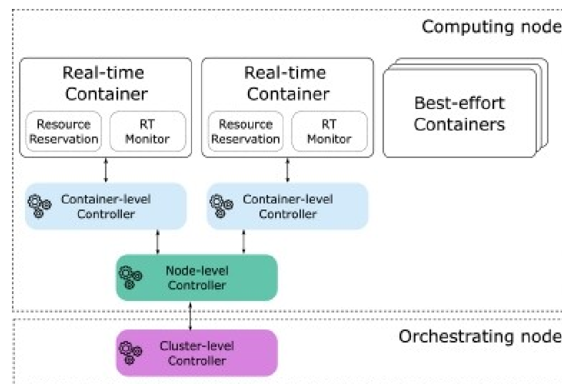


Figure 2.4: Hierarchical Resource Orchestration Framework architecture [9].

2.3.3 Elastic Software Architecture for Extreme-Scale Big Data Analytics

The third work chosen as the foundation is an Elastic Software Architecture for Extreme-Scale Big Data Analytics [10]. While this work has a broader scope, only the proposed orchestration will be utilized as the basis for the thesis work.

The work proposed the use of COMP Superscalar (COMPSs) framework, developed by Barcelona Supercomputing center. This framework is primarily composed of a task-based programming model designed to simplify the development of parallel applications for distributed infrastructures, including clusters, clouds, and containerized platforms. It also includes a runtime system that leverages the inherent parallelism of applications during execution. The framework is complemented by a set of tools that facilitate development, execution monitoring, and post-mortem performance analysis [11].

To enhance programming productivity, the COMP Superscalar (COMPS)s programming model possesses the following characteristics:

- **Agnostic of the actual computing infrastructure:** COMPSs provides a model that abstracts the application from the underlying distributed infrastructure. Therefore, COMPSs programs do not include any details that could tie them to a particular platform, such as deployment or resource management. This characteristic ensures portability between infrastructures with diverse characteristics;
- **Single memory and storage space:** Memory and file system space are abstracted in COMPSs, creating the illusion of a single memory space and file system. The runtime takes care of all necessary data transfers;
- **Standard programming languages:** COMPSs is based on Java but also offers language bindings for Python (PyCOMPSs) and C/C++ applications. This design choice simplifies the learning process, as programmers can leverage their existing knowledge;
- **No Application Programming Interface (API)s:** In the case of COMPSs applications in Java, the model does not require the use of any special API call, pragma, or construct in the application; everything adheres to standard Java syntax and libraries. Regarding Python and C/C++ bindings, a small set of API calls should be used in COMPSs applications.

Chapter 3

Technologies

This chapter provides a foundational overview of the key technologies that support this thesis. Section 3.1 introduces the fundamental concepts of containerization, establishing a basis for understanding the broader technological landscape. Following this, Section 3.2 offers a detailed exploration of Docker, the leading container technology. The chapter then progresses to the concept of container orchestration in Section 3.3, which is essential for managing containers at scale. Section 3.4 focuses on Docker Swarm, widely used platform for container orchestration, Section 3.5 describes the docker API and the last one, Section 3.6 describes the main Scheduling algorithms. Together, these sections provide the essential technological context for the research and developments presented in this thesis.

3.1 Containerization

Container technology is increasingly becoming popular as an alternative to traditional virtual machines as it provides a faster, lighter, and more portable run-time environment for applications. Since containers are portable and have a small footprint, they are suitable to deploy applications on heterogeneous systems, resource limited and power limited computational units making it perfect for fog computing. Deploying containers on edge or fog computing nodes, or deploying containers on edge devices, is extremely advantageous, although it rises challenges in migration, service and resource provisioning and security. A container bundles the application and its binary code, libraries, and configuration files together while sharing the host operating system image. Accordingly, containers efficiently share resources and operate small micro-services, software programs, and even more extensive applications with less overhead than virtual machines. There are many container technologies available as: Docker; CRI-O; rktlet; Containerd and runC. Docker is currently the leading container technology, offering extensive support for various technologies and compatibility with multiple architectures, including ARM, x86, and amd64 [12].

3.2 Docker

Docker offers a powerful mechanism for automating the deployment of applications within containers. In a containerized environment, where applications are virtualized and executed, Docker adds an additional layer by providing a streamlined deployment engine. Designed to deliver a fast and lightweight environment, Docker enables efficient execution of code while also supporting a seamless workflow. This workflow allows code to be easily transitioned from development to testing and finally to production, ensuring that applications are thoroughly tested before they are deployed [13].

By design, containers can be multiplied quickly, where lots of different services are running or even if many instances of a few services are running. When developers decide to run services in containers, they probably need software designed to host and manage those containers. This is broadly known as container orchestration. While Docker and other container engines like Podman and CRI-O are good utilities for container definitions and images, it is their job to create and run containers, not help developers to organize and manage them. Projects like Kubernetes, Docker Swarm, and OKD provide container orchestration for Docker, Podman, CRI-O, and more.

3.2.1 Docker vs VM

Docker works by providing a standard way to run applications. Like virtual machines, containers provide a form of virtualization; however, instead of virtualizing the underlying hardware, containers virtualize the server's operating system. Docker is installed on each server and provides simple commands that can be used to build, start, or stop containers. Figure 3.1 depicts the comparison of Docker vs Virtual Machine (VM)s, Containers run on top the kernel of the host Operating System (OS) differently from VM that need the entirely OS what make VMs much more heavy weigh [14].

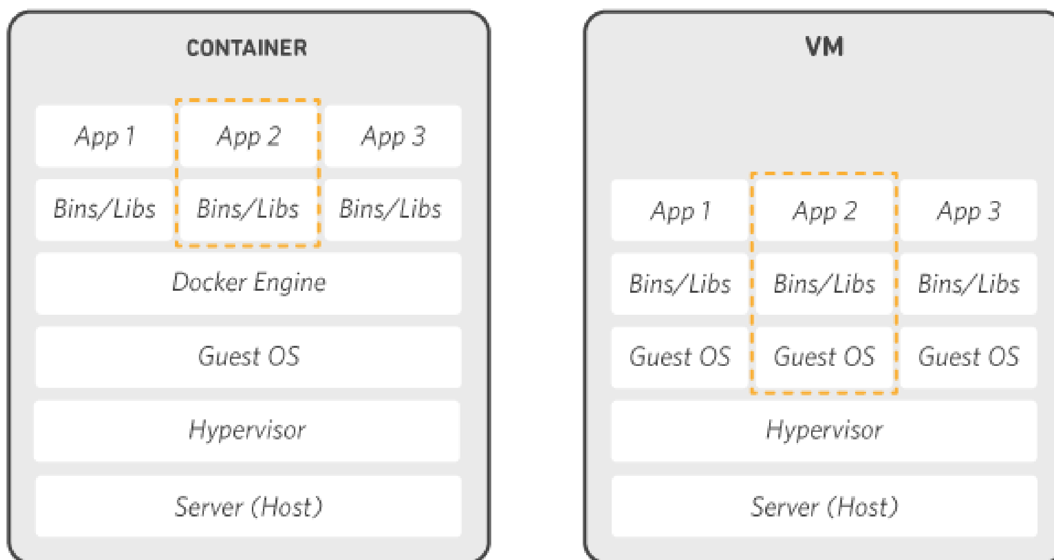


Figure 3.1: Docker vs VM [14].

3.2.2 Architecture

Docker operates on a client-server architecture, where the Docker client interacts with the Docker daemon to handle the core tasks of building, running, and distributing Docker containers. The Docker client and daemon can reside on the same system, or the client can connect to a remote Docker daemon. Communication between the Docker client and daemon is facilitated through a REST API, which can be accessed via UNIX sockets or a network interface. In addition to the primary Docker client, Docker Compose is another tool within the Docker ecosystem that allows management of multi-container applications, enabling the coordination and orchestration of complex services comprised of multiple interconnected containers [15]. Figure 3.2 illustrates the architecture just described.

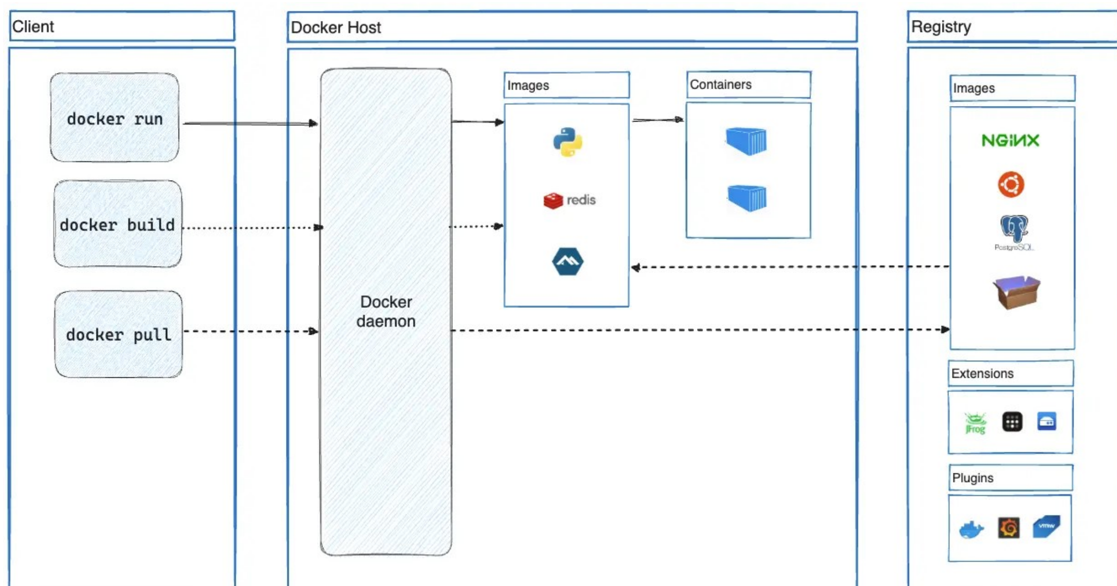


Figure 3.2: Docker architecture. [15].

The Docker Daemon

The Docker daemon (`dockerd`) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. The daemon can also communicate with other daemons to manage Docker services.

The Docker Client

The Docker client (`docker`) serves as the primary interface for interacting with Docker. When commands such as `docker run` are executed, the client sends these commands to `dockerd`, which carries them out. The `docker` command utilizes the Docker API and can communicate with multiple daemons simultaneously.

Docker Desktop

Docker Desktop is an easy-to-install application available for Mac, Windows, or Linux environments that enables the building and sharing of containerized applications and microservices. Docker Desktop includes the Docker daemon (`dockerd`), the Docker client (`docker`), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

Docker Registries

A Docker registry stores Docker images. Docker Hub is a public registry that is available for general use, and Docker looks for images on Docker Hub by default. It is also possible to run a private registry. When the `docker pull` or `docker run` commands are used, Docker retrieves the required images from the configured registry. Similarly, the `docker push` command uploads an image to the configured registry.

Docker Objects

In Docker, various objects such as images, containers, networks, volumes, plugins, and more are created and managed. This section provides a brief overview of some of these objects [15].

Images

An image is a read-only template containing instructions for creating a Docker container. Images are often based on other images with additional customization. For example, an image might be built based on the `ubuntu` image, with the Apache web server, an application, and the necessary configuration details added to make it operational [15].

Images can be custom-built or sourced from those published in a registry. The process of building an image involves creating a `Dockerfile` with a simple syntax that defines the steps needed to create and run the image. Each instruction in a `Dockerfile` creates a layer in the image. When the `Dockerfile` is modified and the image is rebuilt, only the layers that have changed are rebuilt. This approach contributes to making Docker images lightweight, small, and fast compared to other virtualization technologies [15].

3.3 Orchestration

Container orchestration unlocks a wide variety of benefits associated with containers. The primary benefit of container orchestration is streamlined operations. Apart from this, these solutions boost the resilience of the container infrastructure of an enterprise. Finally, they enhance organizational cybersecurity through an automated approach that minimizes human intervention and, thus, error.

Container orchestration solutions primarily serve as a layer between containers and resource pools, using configuration files for controlling enterprise containers. These files are usually written in JavaScript Object Notation (JSON) or YAML format.They rely on these configuration files to locate container images and access container logs. The rules for mounting containers in storage volumes are also stored in these files. Lastly, configuration files are responsible for establishing network connections among containers [16].

In distributed systems container orchestrator is responsible for scheduling container deployment, as well as replicating container groups, to hosts or host clusters. Scheduling and replication are based on various factors, including the availability of memory and processing power. The other factors considered during container deployment include metadata, labels, and location in relation to other hosts. After container deployment at the host level is completed, the container orchestration solution works by taking over the management of containers. The administrator must create a definitions file for the container to ensure the effective management of container automation.

Container orchestration solutions can also scale containers up or down based on workload demands, migrate containers to another host if the current one lacks sufficient resources, manage resource allocation among containers, and handle load balancing to ensure efficient distribution of workloads. Additionally, these solutions simplify service discovery by exposing running container services to other applications on the designated network [16].

There are many well known container orchestration solutions available, the ones that are more used are: Kubernetes [17]; Docker Swarm [18]; Nomad [19]; and Marathon on Mesos [20].

3.4 Docker Swarm

Docker Swarm is a powerful container orchestration tool designed to cluster and schedule Docker containers. It allows IT administrators and developers to manage a cluster of Docker nodes as a unified virtual system. This tool simplifies the deployment, management, and scaling of applications across multiple Docker nodes, providing a streamlined and efficient experience. Docker Swarm includes several key features [21] such as:

- **Cluster Management Integrated with Docker Engine:** Docker Swarm integrates directly with the Docker Engine CLI, enabling the creation of a swarm of Docker Engines for deploying application services. This eliminates the need for additional orchestration software to manage a swarm.
- **Decentralized Design:** In Docker Swarm, node roles are not predetermined at deployment time. Instead, the Docker Engine handles specialization dynamically at runtime. Both manager and worker nodes can be deployed using the Docker Engine, allowing the entire swarm to be built from a single disk image.
- **Declarative Service Model:** Docker Engine employs a declarative model, allowing the possibility to define the desired state of the services the application stack. For instance, is possible to describe an application consisting of a web front-end service, message queuing services, and a database backend.
- **Scaling:** For each service, the desired number of tasks to run can be specified. As scaling up or down occurs, the swarm manager automatically adjusts by adding or removing tasks to maintain the desired state.
- **Desired State Reconciliation:** The swarm manager continuously monitors the cluster's state to ensure it matches the desired configuration. For example, if a service is set to run ten replicas and a worker node hosting two of those replicas fails, the manager will create two new replicas to replace the lost ones and assign them to available workers.
- **Multi-Host Networking:** Docker Swarm allows the definition of an overlay network for services. The swarm manager automatically assigns network addresses to containers on this overlay network when the application is initialized or updated.
- **Service Discovery:** Each service within the swarm is assigned a unique Domain Name System (DNS) name by the swarm manager, which also handles load balancing for the running containers. Any container running in the swarm can be queried via the DNS server embedded in the swarm.
- **Load Balancing:** Docker Swarm allows service ports to be exposed to an external load balancer. Internally, it enables specifying how to distribute service containers across nodes.
- **Security:** Security is a fundamental aspect of Docker Swarm, with each node enforcing Transport Layer Security (TLS) mutual authentication and encryption to secure

communication between nodes. Either self-signed root certificates or certificates from a custom root Certificate Authority (CA) can be used.

- **Rolling Updates:** Docker Swarm supports rolling updates, enabling service updates to be applied incrementally across nodes. The swarm manager allows control over the delay between deployments to different node sets. If any issues arise, rolling back to a previous version of the service is straightforward.

3.5 Docker Engine API

Docker provides the Docker Engine API, a RESTful interface for communicating with the Docker daemon, which allows external control of the Docker Swarm cluster and implementation of the scheduling rules needed for this thesis. This API can be accessed using Hypertext Transfer Protocol (HTTP) clients like `wget` or `curl`, or through HTTP libraries available in most modern programming languages. Additionally, Docker provides an official Software Development Kit (SDK) for Go and Python, which facilitates the development and scaling of Docker applications and solutions.

3.5.1 Python SDK

This includes running and managing containers as well as handling Docker Swarms. To install this library, Docker, Python 3, and pip must already be installed on the target machine [22].

3.5.2 Docker Client Class

There are two ways to establish communication with the Docker Daemon: by using the from_env() function to instantiate the client class, or by manually configuring it through the creation of a DockerClient instance. from_env() returns a client configured from environment variables. The environment variables used are the same as those used by the Docker command-line client. They are:

- **DOCKER_HOST:** The URL to the Docker host.
- **DOCKER_TLS_VERIFY:** Verify the host against a CA certificate.
- **DOCKER_CERT_PATH:** A path to a directory containing TLS certificates to use when connecting to the Docker host.

Parameters:

- **version (str):** The version of the API to use. Set to auto to automatically detect the server's version. Default: auto
- **timeout (int):** Default timeout for API calls, in seconds.
- **max_pool_size (int):** The maximum number of connections to save in the pool.
- **environment (dict):** The environment to read environment variables from. Default: the value of os.environ
- **credstore_env (dict):** Override environment variables when calling the credential store process.

- **use_ssh_client (bool):** If set to True, an ssh connection is made via shelling out to the ssh client. Ensure the ssh client is installed and configured on the host.

3.5.3 Container Class

Local representation of a container object. Detailed configuration may be accessed through the attrs attribute. Local attributes are cached; users may call reload() to query the Docker daemon for the current properties, causing attrs to be refreshed. some Container objects are:

- **attrs:** id The ID of the object.
- **image:** The image of the container.
- **labels:** The labels of a container as dictionary.
- **name:** The name of the container.
- **short_id:** The ID of the object, truncated to 12 characters.
- **status:** The status of the container. For example, running, or exited. The raw representation of this object from the server.

3.5.4 Service Class

Swarm services use a declarative model, which means that define the desired state of the service, and rely upon Docker to maintain this state. Service objects are:

- **id:** The ID of the object.
- **short_id:** The ID of the object, truncated to 12 characters.
- **name:** The service's name.
- **version:** The version number of the service. If this is not the same as the server, the update() function will not work and is not necessary to call reload() before calling it again.
- **attrs:** The raw representation of this object from the server.
- **force_update():** Force update the service even if no changes require it.

  Returns: True if successful.

  Return type: bool
- **logs(**kwargs):** Get log stream for the service. Note: This method works only for services with the json-file or journald logging drivers.

  Parameters:

  - **details (bool):** Show extra details provided to logs. Default: False
  - **follow (bool):** Keep connection open to read logs as they are sent by the Engine. Default: False
  - **stdout (bool):** Return logs from stdout. Default: False
  - **stderr (bool):** Return logs from stderr. Default: False

  - **since (int):** UNIX timestamp for the logs staring point. Default: 0
  - **timestamps (bool):** Add timestamps to every log line.
  - **tail (string or int):** Number of log lines to be returned, counting from the current end of the logs. Specify an integer or 'all' to output all log lines. Default: all

  Returns: Logs for the service.

  Return type: generator

- **reload():** Load this object from the server again and update attrs with the new data.
- **remove():** Stop and remove the service.
- **Raises:** docker.errors.APIError – If the server returns an error.
- **scale(replicas):** Scale service container.

  Parameters: replicas (int) – The number of containers that should be running.

  Returns: True if successful.

  Return type: bool

- **tasks(filters=None):** List the tasks in this service.

  Parameters: filters (dict) – A map of filters to process on the tasks list. Valid filters: id, name, node, label, and desired-state.

  Returns: List of task dictionaries.

  Return type: list

  Raises: docker.errors.APIError – If the server returns an error.

- **update(**kwargs):** Update a service's configuration. Similar to the docker service update command.

  Takes the same parameters as create().

  Raises: docker.errors.APIError – If the server returns an error.

3.6 Scheduling Algorithms

To guarantee no missed deadlines in a hard real-time system, a suitable scheduling algorithm must be chosen. There are two main types of real-time scheduling algorithms: Fixed Priority and Dynamic Priority.

3.6.1 Fixed Priority Scheduling Algorithms

Fixed Priority Scheduling is a real-time scheduling algorithm where each task is assigned a fixed priority, and the scheduler always selects the highest-priority task that is ready to execute. Lower-priority tasks can be preempted if a higher-priority task becomes ready, ensuring that the most critical tasks receive timely processing. This method is predictable and straightforward, making it suitable for systems with static priority requirements, although it may lead to potential issues like starvation for lower-priority tasks and is less effective in handling dynamic priority changes or overload situations.

- **Rate Monotonic Scheduling:** Is based on the principle of preemption. Preemption occurs on a processor when a higher-priority task interrupts and halts the execution of a lower-priority task. This interruption is determined by the priority levels assigned to different tasks within a given task-set. As a preemptive algorithm, Rate-Monotonic Scheduling (RMS) assigns higher priority to tasks with shorter periods. This means that if a task with a shorter period arrives while another task is executing, it preempts the currently running task due to its higher priority. In RM, task priorities are inversely proportional to their periods: the task with the shortest period has the highest priority, while the task with the longest period has the lowest priority. A task-set is considered schedulable under the RM algorithm if it satisfies

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1),$$

where C_i represents the computation time of the i-th task, T_i is the period of the i-th task, and n is the total number of tasks in the system. This equation expresses that a set of processes can be scheduled using RM if the total utilization of the task-set, which is the sum of the ratios of each task's computation time to its period, does not exceed the utilization bound $n(2^{\frac{1}{n}} - 1)$. If this condition is met, the task-set is guaranteed to be schedulable under RM [23] [24].

- **Deadline Monotonic Scheduling:** Proposed by Leung and Whitehead in 1982, applies to processes where $C_i \leq D_i \leq T_i$ and $O_i = 0$. Under this policy, priorities are assigned similarly to the rate-monotonic approach: the process with the shortest deadline is given the highest priority, while processes with progressively longer deadlines receive progressively lower priorities. It is important to note that deadline-monotonic priority assignment is equivalent to rate-monotonic priority assignment when $D_i = T_i$ for all processes. Like the rate-monotonic policy, deadline-monotonic priority assignment is optimal in the sense that if there exists a feasible priority ordering for a set of processes, a deadline-monotonic priority ordering will also be feasible.

3.6.2 Dynamic Priority Scheduling Algorithms

Dynamic priority scheduling algorithms are critical for efficiently managing tasks in computing environments where priorities change over time. These algorithms dynamically adjust task priorities based on factors like deadlines, execution time, and system conditions, ensuring that high-priority tasks are executed promptly. By continuously evaluating and updating priorities, these algorithms help meet real-time constraints, prevent priority inversion, and adapt to varying workloads, thus optimizing system performance and resource utilization. The selection of an appropriate dynamic priority scheduling algorithm depends on the specific needs and constraints of the system, balancing factors such as overhead, complexity, and the nature of task requirements [24].

- **Earliest Deadline First:** The Earliest Deadline First (EDF) scheduling algorithm is a priority-driven algorithm where higher priority is assigned to the request with the earliest deadline. A higher-priority request always preempts a lower-priority one. This scheduling algorithm is an example of a priority-driven algorithm with dynamic priority assignment, meaning that the priority of a request is determined as it arrives. EDF is also known as the deadline-monotonic scheduling algorithm. When a new ready task arrives, it is inserted into a queue of ready tasks, sorted by their deadlines. If sorted lists are used, the EDF algorithm takes $O((N + \alpha)^2)$ time in the worst case, where N represents the total number of requests in each hyper-period of n periodic tasks in

the system, and α represents the number of aperiodic tasks. For the EDF algorithm, all the assumptions made for the Rate-Monotonic (RM) algorithm are valid to EDF, except that the tasks are not required to be periodic. EDF is an optimal uniprocessor scheduling algorithm, meaning that if EDF cannot feasibly schedule a task-set on a uniprocessor, no other scheduling algorithm can. This can be proven using a time slice swapping technique, which demonstrates that any valid schedule for any task-set can be transformed into a valid EDF schedule. reassignment [23].

- **Least Laxity First:** is a dynamic priority scheduling method that operates at the job level. In Least Laxity First (LLF), every moment in time is treated as a scheduling event because the laxity of each task continuously changes. Laxity is defined as the time remaining until a task's deadline minus its remaining execution time. Tasks with the least laxity at any given moment are assigned higher priority, meaning they are scheduled to run before others. LLF is considered an optimal algorithm because if a task-set passes the utilization test, it is guaranteed to be schedulable under LLF. An additional benefit of LLF is its ability to provide advance warning if a task is likely to miss its deadline, allowing for potential corrective actions.However, LLF also has notable disadvantages. One major drawback is its high computational demand, as every time instant triggers a scheduling event. This can lead to significant overhead, particularly when multiple tasks share the same minimum laxity, resulting in frequent context switches. Despite its ability to handle tasks with varying execution times effectively, the overhead associated with LLF can negatively impact overall system performance[25].

Chapter 4

Requirements and System Architecture

This chapter outlines the system requirements and describes the architecture of the proof-of-concept, which is built on a Docker Swarm cluster to ensure network security and scalability. The cluster includes a Manager node and at least one Worker node.The Manager node handles task scheduling using the Python-based Swarm RT-Sched and RT-Income Manager services. Worker nodes execute the assigned services and include a pause/unpause service for system preemption.

4.1 Requirements

The following tables 4.1, 4.2, 4.3, 4.4, and 4.5 represent the structured requirements in Easy Approach to Requirements Syntax (EARS) of the system. The system is designed to handle task scheduling using the EDF algorithm while managing multiple services and nodes. The configuration, event handling, and network communication are key components that allow the system to function efficiently and in real-time.

4.1.1 General Scheduling Requirements

Table 4.1: General Scheduling Requirements

ID	Requirement Description
1.1	The system shall schedule services using the Earliest Deadline First algorithm.
1.2	The system shall evaluate schedulability upon receiving a new service request.
1.3	The system shall be able to handle multiple request in parallel.

4.1.2 Service Management

Table 4.2: Service Management Requirements

ID	Requirement Description
2.1	The system shall contain three primary services: Real-Time Scheduler (RT-sched), Income Manager, and Pause/Unpause Service
2.2	The system shall preempt low-priority services when a higher-priority service needs to be scheduled.
2.3	If a new service request causes a deadline miss, the system shall deny the request.
2.4	If a new service request does not cause a deadline miss, the system shall add it to the scheduling queue according to its priority.
2.5	The system shall remove a service from the cluster when it is completed.

4.1.3 Network Communication

Table 4.3: Network Communication Requirements

ID	Requirement Description
3.1	The system shall listen for new task requests on port 8765.
3.2	The system shall communicate using the TCP protocol.
3.3	The system shall be capable of managing communication across multiple nodes.

4.1.4 Configurability and Node Management

Table 4.4: Configurability and Node Management Requirements

ID	Requirement Description
4.1	The system shall be configurable via a configuration file.
4.2	The configuration file shall include a list of available worker nodes.
4.3	The configuration file shall include the IP addresses of each worker node.
4.4	The configuration file shall include the services allowed by each worker nodes.

4.1.5 Event Handling

Table 4.5: Event Handling Requirements

ID	Requirement Description
5.1	When a service is added to the scheduling queue, the system shall trigger an event.
5.2	When a service transitions from a pending state to a running state, the system shall trigger an event.
5.3	When a service is removed from the cluster, the system shall trigger an event.
5.4	When an event is triggered, the system shall run the algorithm to execute the highest priority services.

4.2 System Architecture

The proof-of-concept system is built on a Docker Swarm cluster, ensuring network security and offering the flexibility to scale as needed. The cluster comprises a Manager node and at least one Worker node, as illustrated in Figure 4.1. The Manager node is responsible for scheduling all task requests based on their deadlines, utilizing the Swarm RT-Sched and RT-Income Manager services, which are developed in Python. Worker nodes run the services requested by the Manager and include an additional pause/unpause service that allows full control over task progression.

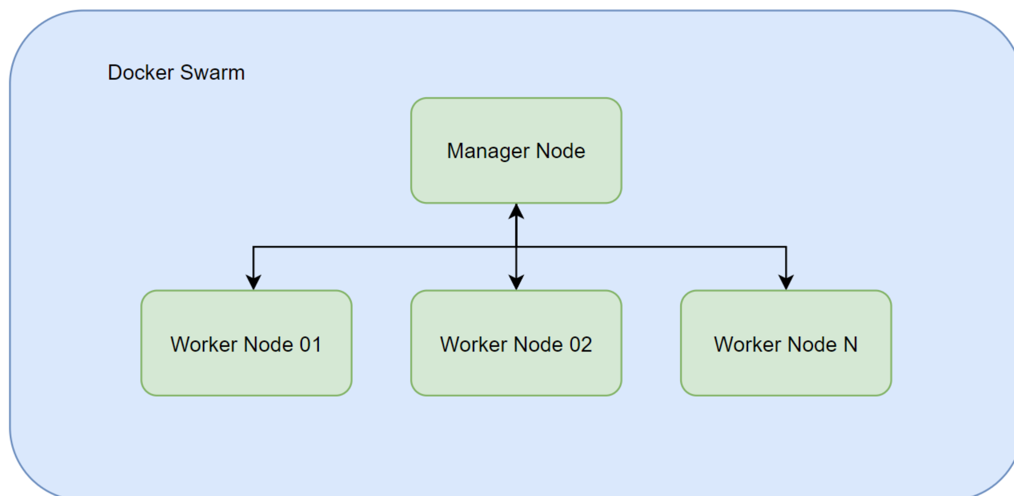


Figure 4.1: Docker Swarm cluster.

4.2.1 Manager Node

The Manager node hosts three key services: (1) the RT-Income Manager, (2) the Swarm RT-Sched and (3) the Swarm Engine API, as shown in Figure 4.2. The RT-Income Manager is responsible for receiving new task requests from client devices on port 8765, obtaining responses from services on port 8768, and redirecting the responses back to the client that requested the task. The Swarm RT-Sched processes incoming requests from the RT-Income

Manager and schedules all services in the cluster according to their deadlines. Lastly, the Swarm Engine API is responsible for managing the Docker Swarm cluster.

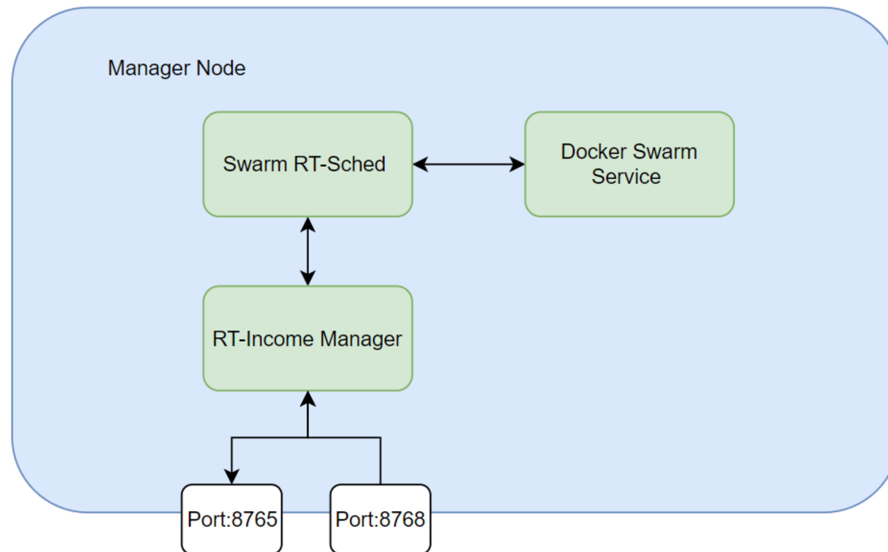


Figure 4.2: Manager node.

4.2.2 Worker Node

Figure 4.3 illustrates the communication between the services and the Manager node, along with the creation of these services by the Docker Engine. For a Worker node to receive services, it must be assigned to the Swarm cluster and remain active. The number of services on each Worker node is configurable via a JSON file that the Swarm RT-Sched reads. The Worker node exclusively runs the services deployed by the Manager node, along with a pause/unpause service. This additional service is necessary because Docker Swarm does not natively support pausing a service. As a workaround, the pause/unpause service can pause and resume the container associated with a service, enabling preemption within the system.

When a service completes its processing, the response must be send to the RT-Income Manager service via port 8768, ensuring isolation between Worker nodes and client devices. The pause/unpause service operates as an HTTP server on port 8770, executing commands to pause and unpause containers on the Worker node to which it is assigned.

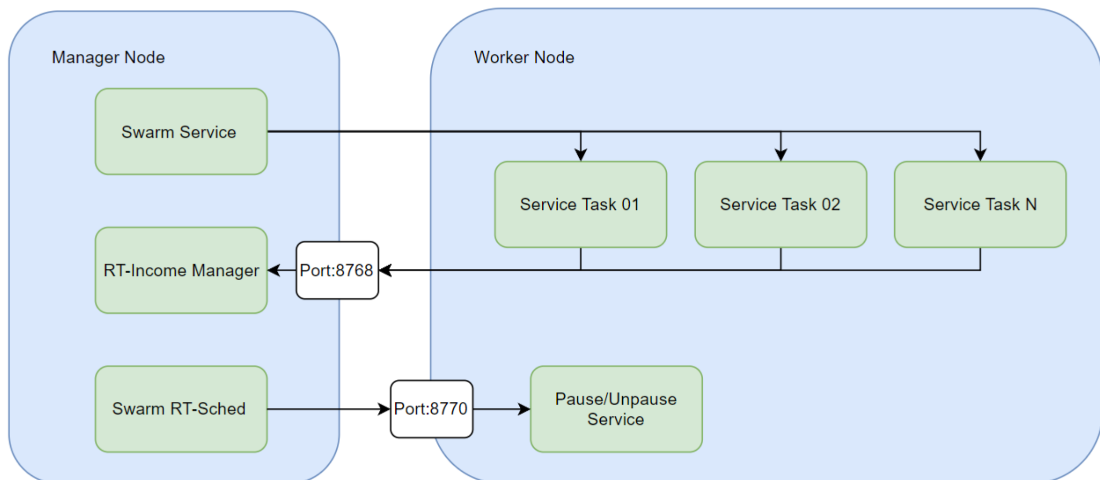


Figure 4.3: Worker node.

Chapter 5

Development

This chapter provides a comprehensive overview of the development process for the system's core components, focusing on achieving real-time container orchestration within a Docker Swarm cluster. It begins by detailing the hardware setup, which forms the foundation of the system, outlining the selection and configuration of hardware components to ensure optimal performance and reliability.

Next, the chapter delves into the development of key software components. The Pause-Unpause Service, running on worker nodes, enables task preemption. The Income Manager Service manages incoming task requests from client devices, coordinates with services to obtain responses, and redirects these responses back to the requesting clients. The Swarm RT-Sched is highlighted as a critical component responsible for task scheduling, load balancing, and managing service life-cycles across the cluster, ensuring timely task execution while optimizing resource utilization.

The chapter also discusses various supporting threads that interact with the Cluster Scheduler to maintain system stability and responsiveness. These include the Service Monitor Thread, which monitors the state of services, and the Remove Service Thread, which handles resource cleanup to ensure efficient operations.

Additionally, the development of the Client Simulator Script is covered. This tool is essential for simulating client requests and assessing the system's real-time performance. Through these discussions, the chapter demonstrates how the integration of hardware and software components creates a robust, high-performance environment for real-time container orchestration, effectively managing dynamic workloads in a distributed system.

5.1 Hardware Environment

To simulate the fog Docker Swarm cluster, three Raspberry Pi 5 were used: one served as the Manager node, and the other two as worker nodes. They were connected via Ethernet cables through a switch.

5.1.1 Raspberry Pi

The Raspberry Pi is a compact yet versatile prototyping device that has significantly impacted scientific research and education. Its affordability, small footprint, and user-friendly design make it an ideal platform for a wide range of scientific applications, including data acquisition, control systems, and modeling. Moreover, the Raspberry Pi's powerful processor and substantial memory make it well-suited for scientific modeling and simulation, capable of

handling complex software with efficiency. For these reasons the Raspberry Pi 5 was chosen for this thesis.

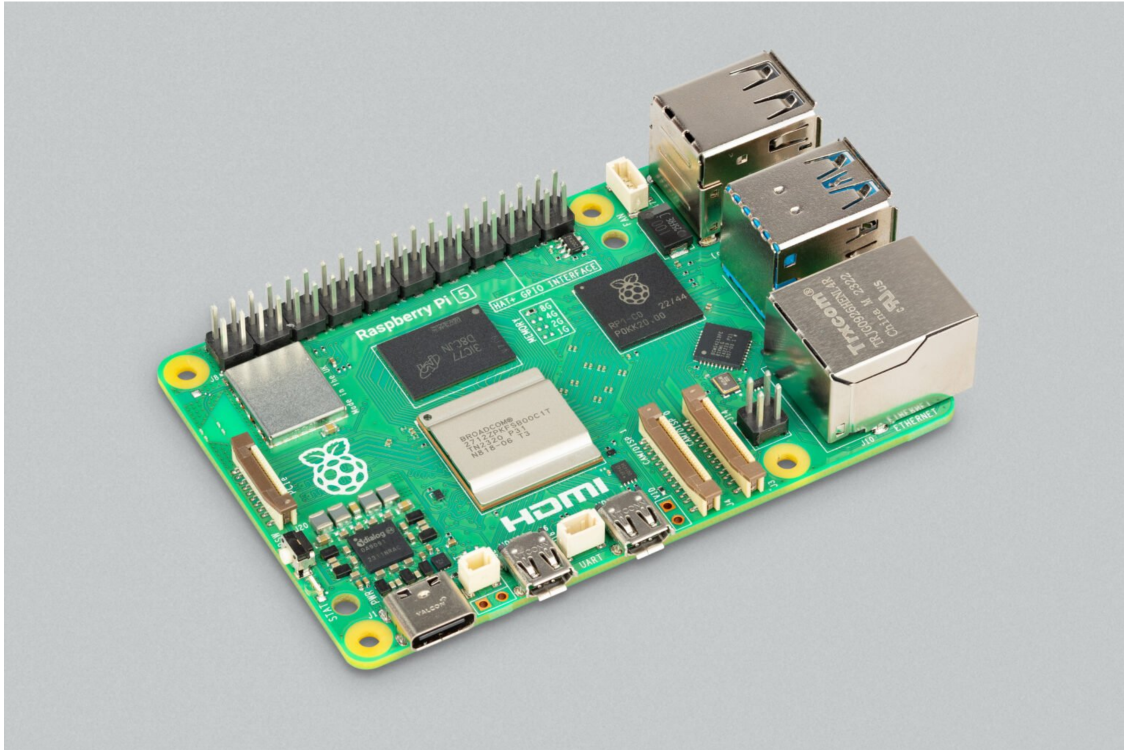


Figure 5.1: Raspberry Pi 5 [26].

The Raspberry Pi 5 illustrated in Figure 5.1 is two to three times faster than its predecessors and it has the following features [26]:

- 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU
- VideoCore VII GPU, supporting OpenGL ES 3.1, Vulkan 1.2
- Dual 4Kp60 HDMI display output
- 4Kp60 HEVC decoder
- Dual-band 802.11ac Wi-Fi
- Bluetooth 5.0 / Bluetooth Low Energy (BLE)
- High-speed microSD card interface with SDR104 mode support
- 2 × USB 3.0 ports, supporting simultaneous 5Gbps operation
- 2 × USB 2.0 ports
- Gigabit Ethernet, with PoE+ support (requires separate PoE+ HAT, coming soon)
- 2 × 4-lane MIPI camera/display transceivers
- PCIe 2.0 x1 interface for fast peripherals
- Raspberry Pi standard 40-pin GPIO header
- Real-time clock

- Power button

In this thesis, three Raspberry Pi 5 models with 4GB of memory are used: one serves as the Manager node, and the other two as worker nodes. All of them run Ubuntu Server 24.04 LTS. The setup includes a 5V power supply, a 32GB SD card, and a case with a cooling system as accessories.

5.1.2 Network Configuration

The network configuration includes three primary components: the Manager node, the Worker node 01, and the Worker node 02, each designated with specific IP addresses:

- Manager node: 192.168.1.120,
- Worker node 01: 192.168.1.119,
- Worker node 02: 192.168.1.118.

All nodes are interconnected through a central switch, facilitating direct communication between the nodes within the Local Areea Network (LAN).This setup is illustrated in illustrated in Figure 5.2.

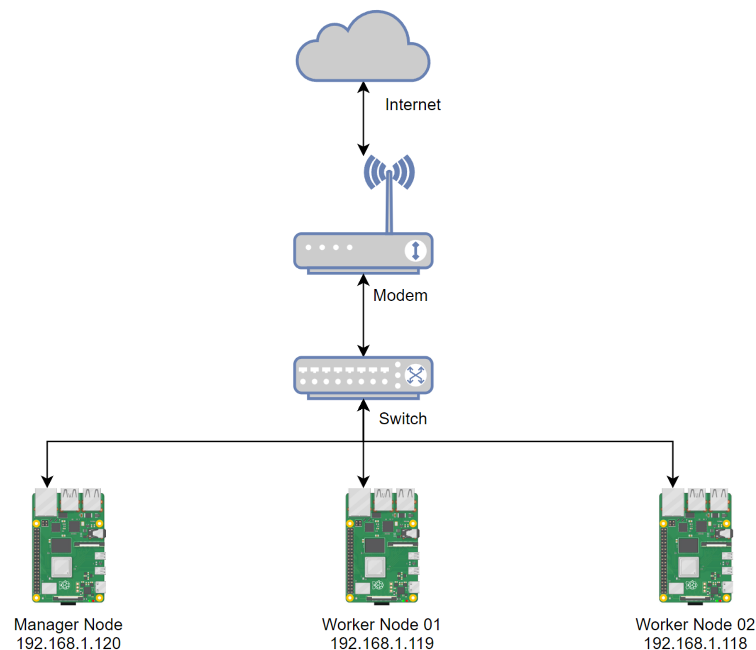


Figure 5.2: Ethernet connection diagram.

This switch is not only pivotal for local communications but also serves as the gateway interface to broader network access. It is connected via an Ethernet cable to a modem, which in turn is connected to the Internet. This arrangement ensures that each node while primarily interacting within the local network also has access to external networks, including the Internet.

The entire network configuration is established using wired connections to minimize latency and maximize reliability and speed of data transfer. The wired setup is particularly beneficial in maintaining stable and secure communications between the nodes and the Internet.

5.2 Docker Swarm Cluster

Setting up the Docker Swarm cluster begins by initializing the Swarm on the Manager node with the docker swarm init command. This command configures the Swarm cluster and generates a unique token that is used to add Worker nodes or additional Manager nodes to the cluster, as needed. The command is as follows:

```
1 docker swarm init --advertise-addr <Manager_node_IP>
```

Listing 5.1: Docker Swarm initialization command.

After the Swarm is initialized, additional nodes can be integrated into the cluster by using the docker swarm join command, which uses the token provided by the Manager node:

```
1 docker swarm join --token <Manager_node_token> <Manager_nodeIP>:2377
```

Listing 5.2: Docker Swarm join command.

Once the cluster is set up and nodes are successfully added, the docker node ls command displays all nodes in the Swarm. This command provides detailed information for each node, including its ID, hostname, status, availability, manager status, and the version of the Docker engine running on it.

```
controller@rpi5-calai02-controller:~$ docker node ls
ID                            HOSTNAME                   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
n3o2iboo68utdkuje805l3w9i     rpi3-node02                Down      Active                          25.0.4
5dtbiw8h3j2wdgtahyrtqxv0u *   rpi5-calai02-controller    Ready     Active         Leader           25.0.4
4sksj32t1ld74d59td512vidf     rpi5-node01                Ready     Active                          25.0.4
31t84jk3tjj3si01cipe2nni8     rpi5-node02                Ready     Active                          27.1.1
```

Figure 5.3: Docker Swarm nodes.

Figure 5.3 shows the output of this command, displaying the nodes in the cluster used in this thesis. In this setup, **rpi3-node02** is an inactive node, **rpi-calai02-controller** is the Manager node, and both **rpi5-node01** (Worker node 01) and **rpi5-node02** (Worker node 02) are active Worker nodes.

5.3 Pause/Unpause Service

Docker Swarm does not provide a built-in method to pause a service. To pause and resume services, all associated containers must be stopped manually. When lower-priority services need to be preempted, the system includes a service running on each worker node that can pause or resume containers as requested by the Swarm RT-Sched service. The command to create this service is:

```
1 $ docker service create \
2 --name pause_unpause_containers_node01 \
3 --replicas 1 \
4 --constraint 'node.hostname == rpi5-node01' \
5 --publish mode=host,target=8770,published=8770 \
6 --mount type=bind,source=/var/run/docker.sock,target=/var/run/docker.
    sock \
7 --entrypoint "python3 /app/pause_unpause_container.py" \
8 --restart-condition any \
9 rafaelcalai633/pause_unpause_container:1.0.0
```

Listing 5.3: Create service pause_unpause.

This command creates the service "pause_unpause_containers_node01" on the "rpi5-node01" worker node, exposing port 8770 and using the Docker Hub image "rafaelcalai633 /pause_unpause_container:1.0.0". The service is set to restart automatically for any reason. After creating this service on both worker nodes, the command `docker service ls` confirms that both services are running, as shown in Figure 5.4:

```
controller@rpi5-calai02-controller:~$ docker service ls
ID             NAME                              MODE         REPLICAS   IMAGE                                      PORTS
bqqmhaoyn3l8   pause_unpause_containers_rpi5-node01   replicated   1/1        rafaelcalai633/pause_unpause_container:1.0.0
a3oy4n6oz2yk   pause_unpause_containers_rpi5-node02   replicated   1/1        rafaelcalai633/pause_unpause_container:1.0.0
```

Figure 5.4: Pause/unpause service list.

The sniped code for the main function is shown in the following listing. It begins by starting an HTTP server on port 8770. When the server receives a 'pause' or 'unpause' request, it attempts to send the corresponding command to the Docker daemon. If the target container is no longer running because it has completed its task, an error may occur. This error is caught as an exception and logged accordingly. No feedback is sent back to the Swarm RT-Sched.

```
1  def main():
2      client = docker.from_env()
3      server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4      server.bind((HOST, PORT))
5      logging.info(f"socket binded to port: {PORT}")
6      server.listen()
7      logging.info("socket is listening")
8      while True:
9          connection, addr = server.accept()
10         logging.info(f"Connetion from {addr}")
11
12         while True:
13             data = connection.recv(1024)
14             if data:
15                 data = eval(data)
16                 logging.info(data)
17
18                 try:
19                     if data["command"] == "pause":
20                         client.containers.get(data["id"]).pause()
21                         logging.info(f"Pause container {data['id']}")
22                     elif data["command"] == "unpause":
23                         client.containers.get(data["id"]).unpause()
24                         logging.info(f"Unpause container {data['id']}")
25                 except Exception as err:
26                     logging.error(f"Not able to {data}" + str(err))
27                 break
```

Listing 5.4: Code of pause_unpause script.

5.4 Income Manager Service

The Income Manager Service establishes a multithreaded network server designed for managing task scheduling and service coordination within a distributed system. It operates two primary components: the Income Manager Server and the Service Response Server. The

Income Manager Server listens for incoming task requests from clients, forwards these requests to the Swarm RT-Sched, and sends the scheduler's response back to the client if the scheduler is not able to handle it on the cluster.

The "service_response_server" listens for completion messages from services and updates a shared dictionary with these responses, while also requesting service removal from the scheduler. Both servers run concurrently in separate threads, utilizing locking mechanisms to ensure thread-safe operations. This design allows the system to handle multiple tasks and connections simultaneously, facilitating efficient management of tasks across a distributed network. The snipped code from the function service_response_server() is shown in Listing 5.5.

```
1 def service_response_server():
2
3       try:
4         client_socket = socket.socket()
5         client_socket.connect((SCHED_HOST, REMOVE_SERVICE_PORT))
6         client_socket.send(data)
7         client_socket.close()
8       except Exception as err:
9         logging.error(f"Error during request of service removal: {str(
    err)}")
10        client_socket.close()
11    server.close()
```

Listing 5.5: Function service response server.

The removal request in lines 3 to 11 was implemented to reduce overhead in the Swarm RT-Sched, as the Docker Engine API can take over two seconds to detect container completion. By immediately notifying the scheduler when a service response is received, the scheduler can promptly recognize that the service has completed, allowing a new service to take its place. This approach reduces the likelihood of missed deadlines and improves the system's determinism. The entirely code is available in the master branch: `https://github.com/rafaelcalai/rt_cluster_income_manager`

5.5 Swarm RT-Sched

Swarm RT-Sched implements a real-time cluster scheduling system based on EDF using Docker Swarm to manage and orchestrate services across multiple worker nodes. It utilizes Docker's API to dynamically create, pause, unpause, and remove services based on node availability and specific scheduling requirements. The system employs detailed logging to track execution and uses multiple threads to manage concurrent tasks such as monitoring services, removing services, handling service requests, and running the scheduler, which will be described in more detail. To ensure efficient service scheduling, it incorporates load balancing mechanisms that can pause lower-priority tasks when necessary to meet the deadlines of higher-priority ones. The system continuously monitors the state of services, updating their status and triggering appropriate actions as they transition to different states. Configuration data, including worker node capacities and service limits, is loaded from a "config.json" file, which guides the system's operations. The main function orchestrates the initialization of all necessary components and threads, ensuring the continuous and efficient management of services within the cluster. The entirely code is available in the master branch: `https://github.com/rafaelcalai/docker_swarm_controller`

5.5.1 Service Request Thread

The Service Request Thread is responsible for handling incoming service requests from the Income Manager Service, which communicates via HTTP port 8767. Upon receiving a new service request, the thread assesses whether the cluster has sufficient resources to meet the requested service's deadlines. This evaluation involves simulating the potential impact of incorporating the new service into the existing workload. If the service is determined to be schedulable, it is added to the scheduling queue based on priority, and a counting semaphore is incremented to synchronize with the producer-consumer pattern, allowing the Service Request Thread to proceed. If the service cannot be accommodated, the request is denied, and the Income Manager Service is informed accordingly, allowing it to notify the client promptly. This approach enhances safety by ensuring the client is quickly informed if the task cannot be processed in time, enabling them to take preemptive measures to mitigate potential risks. This thread plays a crucial role in managing the flow of new tasks and ensuring that the cluster only accepts services it can efficiently handle.

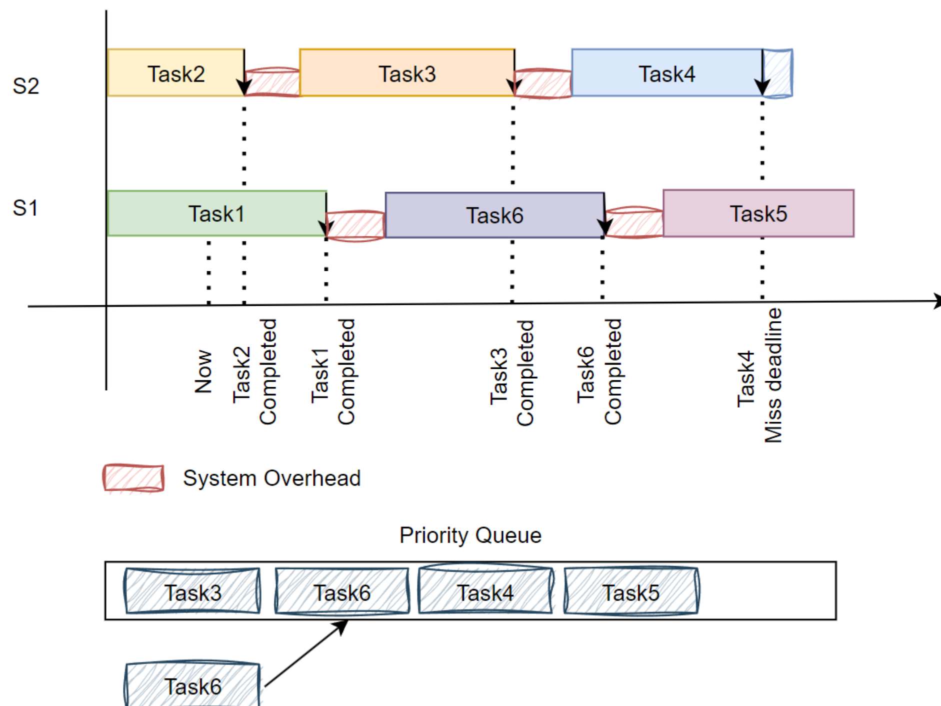


Figure 5.5: Miss deadline task request.

Figure 5.5 illustrates how the Service Request Thread evaluates whether the new request "Task6" can be incorporated into the scheduling queue. The scenario involves task management across two worker nodes, S1 and S2, with a priority queue determining the order of task execution. When Task6 arrives, marked as "Now," two tasks (Task1 and Task2) are already running, and three tasks are waiting in the priority queue. Due to its higher priority, based on execution time and deadline, Task6 is placed ahead of Task4 and Task5 in the queue, taking the second position.

To ensure that the evaluation process does not block other threads, the Service Request Thread creates a copy of both the running services queue and the scheduling priority queue. This allows the thread to simulate the potential impact of adding Task6 without interfering with the ongoing operations of other threads in the system.

As tasks are processed, a system overhead of five seconds is accounted for task creation and the actual start of its container on the worker node. Although this overhead is exaggerated for simulation purposes, it allows for a more conservative estimation. Task6 is successfully scheduled on System S1 without missing its deadline. However, its addition to the schedule shifts the timeline for subsequent tasks, causing Task4 to miss its deadline on System S2. While Task6 meets its own deadline, its inclusion triggers a ripple effect that delays Task4 beyond its acceptable deadline.

This scenario underscores a critical decision in task scheduling: even if a task like Task6 can be executed without missing its own deadline, it might still need to be denied if its inclusion causes other tasks, such as Task4, to miss their deadlines. The figure effectively illustrates the broader impact of scheduling decisions on overall system performance, highlighting the importance of considering the downstream consequences of adding tasks to the queue.

5.5.2 Cluster Scheduler Thread

The Cluster Scheduler Thread is a central component of the Swarm RT-Sched system, responsible for making real-time decisions on the scheduling, pausing, unpausing, and creation of services across the Docker Swarm cluster. This thread ensures that tasks are executed in a manner that meets their deadlines while optimizing resource usage across the cluster. It balances workloads across available nodes, manages the lifecycle of services, implements preemption logic, and handles service creation to ensure that critical tasks are prioritized and executed on time. By reacting to events in real-time and coordinating closely with other threads, the Cluster Scheduler Thread plays a crucial role in maintaining the overall efficiency and effectiveness of the system.

Trigger Mechanism

The Cluster Scheduler Thread operates in a producer-consumer model, where it only runs when an event triggers it. The events that can trigger the thread include:

- The arrival of a new service request.
- A pending service transitioning to a running state.
- The completion of a task that frees up resources in the system.

The code snippet in Listing 5.6 details the trigger mechanism. The semaphore sched_event is initialized on line 1 and waits for an event on line 9. Events are signaled using the semaphore, ensuring that the Cluster Scheduler Thread is activated only when necessary. This approach conserves resources and minimizes unnecessary processing.

```
sched_event = threading.Semaphore(0)

def cluster_scheduler_thread(config_data):
    worker_nodes = config_data["nodes"]["worker_nodes"]
    service_limit = config_data["nodes"]["service_limit"]
    worker_node_addresses = config_data["nodes"]["ip_address"]

    while True:
        sched_event.acquire()
        if sched_queue:
            sched_queue_lock.acquire()
            task_request, pause_service = None, None
            if sched_queue:
                for index, task in enumerate(sched_queue):
                    if task["service_state"] == "paused":
                        work_node, pause_service = check_schedulability(
                            task["work_node"], service_limit, task
                        )
                    else:
                        work_node, pause_service = check_schedulability(
                            worker_nodes, service_limit, task
                        )
                    if work_node:
                        task_request = sched_queue.pop(index)
                        break
            sched_queue_lock.release()

            if task_request:
                if pause_service:
                    pause_service_containers(
                        pause_service, worker_node_addresses[work_node],
 work_node
                    )
                if task_request["service_state"] == "new":
                    sched_service(
                        task_request,
                        worker_node_addresses,
                        work_node,
                    )
                else:
                    logging.info(
                        f"Unpause a service from the sched queue: {
task_request['task_name']}"
                    )
                    unpause_service_containers(task_request)
```

Listing 5.6: Function cluster scheduler thread.

Handling the Scheduling Queue

The Cluster Scheduler Thread interacts with the scheduling queue (sched_queue), which holds tasks waiting to be scheduled. Each task in this queue is prioritized based on its scheduling deadline, ensuring that tasks with the most urgent deadlines are handled first.

```
def check_schedulability(worker_nodes, service_limit, task_request):
    available_worker_nodes = get_available_worker_nodes(worker_nodes)
    services_associated = get_worker_nodes_load(available_worker_nodes)
    work_node, pause_service = load_balance(
                                    available_worker_nodes,
                                    services_associated,
                                    service_limit,
                                    task_request
                                    )
    return work_node, pause_service
```

Listing 5.7: Function check schedulability.

The thread first checks whether the task at the position zero in the queue (high priority) can be scheduled without violating its deadline or impacting the deadlines of other tasks already running in the system. This is done using the check_schedulability () function.

If the task is schedulable, it is removed from the scheduling queue and prepared for execution. For new tasks, they are scheduled to run on the least-loaded worker node. If a task has been paused and its worker node has an available slot, the thread will unpause the task, allowing it to resume execution. If the task remains paused and no slots are available on its worker node, the thread will proceed to the next task in the queue, repeating the process until it reaches the end of the queue.

Load Balancing

The Cluster Scheduler Thread is also responsible for load balancing across the cluster. It ensures that tasks are distributed evenly among the available worker nodes to prevent any single node from becoming overwhelmed. The load_balance() function is used to select the most appropriate worker node for a new or resumed task. If all worker nodes are busy and cannot accommodate the new task without missing its deadline, the thread may pause a lower-priority task currently running on a node. This allows the new task to be scheduled and executed in a timely manner.

```
def pause_lower_priority_service(task_sched_deadline):
    lowest_priority_service = ""
    available_worker_node = ""
    lowest_sched_deadline_priority = task_sched_deadline +
    PAUSE_OVERHEAD
    for service in running_services:
        executed_time = datetime.now() - running_services[service]["
    service_started"]
        if (
            running_services[service]["sched_deadline"] + executed_time
        ) > lowest_sched_deadline_priority and
        running_services[service]["service_state"
        ] == "running":
            lowest_sched_deadline_priority = running_services[service]["
    sched_deadline"]
            lowest_priority_service = service
            available_worker_node = running_services[service]["work_node
    "]
    return available_worker_node, lowest_priority_service
```

Listing 5.8: Function pause lower priority service.

Service Creation

When the Cluster Scheduler Thread determines that a new task can be scheduled, it initiates the service creation process. Service creation involves setting up a Docker service on the selected worker node, which includes specifying the image to use, the command to execute, and any resource constraints. The service is defined using the create_service() function, which configures the Docker service with the necessary parameters. This includes assigning the service to a specific worker node, setting CPU and memory limits, and defining the command to be executed within the service.

```
def create_service(
    service_name, work_node, task_request, worker_node_addresses,
    secrets=None
):
    image_name = "rafaelcalai633/" + task_request["image"]
    command = [
        "python",
        task_request["command"],
        str(task_request["execution_time"]),
        service_name,
    ]
    secrets = secrets or []
    resources = {
        "Limits": {
            "NanoCPUs": int(1 * 1e9),  # Convert CPU to NanoCPUs
            "MemoryBytes": int(256 * 1e6),  # Convert MB to Bytes
        }
    }

    constraints = [f"node.hostname == {work_node}"]
    task_request["work_node"] = work_node
    task_request["service_state"] = "pending"
    task_request["work_node_ip"] = worker_node_addresses[work_node]
    task_request["service_started"] = datetime.now()

    running_services_lock.acquire()
    running_services[service_name] = task_request
    running_services_lock.release()

    logger.info(f"Service {service_name} with Constraints: {constraints}
    created!")
    return client.services.create(
        name=service_name,
        image=image_name,
        constraints=constraints,
        secrets=secrets,
        command=command,
        restart_policy={"Condition": "none"},
        resources=resources,
    )
```

Listing 5.9: Function create service.

The create_service() function assigns the service to a specific worker node by setting the appropriate constraints. It also updates the service's state to pending and records the node's IP address. The Docker service is created using the client.services.create() method, which interacts with the Docker API to deploy the service on the specified worker node.

Once the service is successfully created, it is added to the list of running services, and the system begins monitoring its execution.

Preemption Logic

Preemption is an essential aspect of real-time scheduling, allowing the system to pause and later resume services as needed. However, in the Swarm RT-Sched system, preemption is only possible when a service is in the "running" state. This is because pausing involves stopping the Docker container associated with the service, which can only be done when the service is actively running.

The thread checks if preemption is necessary when a new task is added to the scheduling queue. If a lower-priority task is identified as a candidate for pausing, the thread will pause it and allocate the freed-up resources to the new task.

Service Management

The Cluster Scheduler Thread manages the lifecycle of services in the cluster. It transitions services between different states (new, paused, running) based on real-time scheduling decisions. By carefully managing these transitions, the thread ensures that the cluster remains responsive and capable of handling dynamic workloads.

- Scheduling New Services: For services in the new state, the thread allocates them to the appropriate worker node and initiates their execution.
- Unpausing Services: If a service has been paused, the thread resumes it when resources become available, ensuring that it continues execution without missing its deadline.

Concurrency Control

The thread operates in a multi-threaded environment where various components of the system interact concurrently. To manage this, the thread uses locks `sched_queue_lock` and `running_services_lock` to ensure that shared data structures like the scheduling queue and the list of running services are accessed in a thread-safe manner. This prevents race conditions and ensures consistent system behavior.

5.5.3 Service Monitor Thread

The Service Monitor Thread is crucial in the real-time management of services within the cluster. It continuously monitors the state of pending services, regularly polling the Docker API to track their progress as they transition from startup to full operation. Once a service moves from a pending state to a running state, this thread promptly updates its status in the system's internal records, ensuring that the current state of all services is accurately reflected. Additionally, it triggers any necessary follow-up actions, such as notifying other threads or processes that depend on the updated service status. This thread is essential for maintaining real-time responsiveness, enabling the system to accurately track service statuses and react to changes swiftly.

A key aspect of the system's scheduling logic is that services in a pending state cannot be preempted; preemption is only allowed once a service has entered the running state. This is because the preemption process involves pausing the Docker container associated with the service, which is only possible when the service is actively running. The pending status,

therefore, is crucial for effectively managing the service lifecycle, ensuring that only fully active services are subject to preemption. This approach allows the system to maintain precise control over resource allocation, meet critical deadlines, and efficiently manage the workload across the cluster.

5.5.4 Remove Service Thread

The Remove Service Thread is essential for maintaining the cluster's efficiency and stability. It periodically scans all services running in the Docker Swarm, specifically identifying those that have completed and are marked as shutdown. Upon detecting these services, the thread promptly removes them from the cluster, thereby freeing up valuable resources for other tasks. Additionally, it updates the internal lists of running and pending services and increments the counting semaphore to notify the scheduler thread of the new event. By automating the cleanup of completed or obsolete services, this thread helps prevent resource bottlenecks and ensures optimal resource utilization within the cluster.

In some instances, the scheduler may attempt to pause a service, but the service might complete its execution before the pause command takes effect. Since the pause command does not provide feedback, a completed service may still be incorrectly listed in the scheduling queue as paused. To address this, the Remove Service Thread identifies such discrepancies and removes the service from the queue if it is found in this state. This ensures that the scheduling queue accurately reflects the actual status of services, maintaining the efficiency and integrity of the scheduling process.

5.5.5 Remove Service Server Thread

The Remove Service Server Thread is designed to handle external requests for service removal within the Docker Swarm cluster. It sets up a server socket that listens for incoming connections on a specified port. When a request to remove a specific service is received, the thread processes the request by identifying the relevant service within the cluster and initiating its removal. The thread also updates the internal service management structures to reflect the removal, ensuring consistency across the system.

This approach was implemented to enhance overall system performance by addressing potential delays in the Docker Engine API, which can sometimes struggle to accurately detect when a container has shut down. By directly managing the shutdown process through this thread, triggered by a message from the Income Manager Service, the system bypasses the need to rely on the potentially slow responses from the Docker Engine API. This reduces latency and ensures that resources are promptly freed for other tasks, thereby improving the system's responsiveness and maintaining efficient resource utilization within the cluster. Two threads are capable of performing service exclusion: the first thread that detects the need for removal will execute it, ensuring that the system remains agile and responsive to changes in the service state.

5.6 Client Simulator Script

The Client Simulator Script is designed to simulate multiple device clients sending task requests to the system, specifically targeting the Swarm income manager service. It logs the outcomes for comprehensive monitoring and analysis. The script is equipped to handle multiple task requests concurrently, utilizing threading to simulate the execution of tasks

in a real-time environment. It establishes a network connection with the Swarm income manager service via Transmission Control Protocol (TCP), through which it sends JSON encoded task data. The script then waits for a response from the server, logging the results, including whether the task was accepted or denied and how the response time compares to the task's deadline. This ensures that all interactions are captured and can be reviewed for performance analysis or debugging.

The script is also highly configurable, allowing users to specify different sets of tasks through command-line arguments. It manages the timing and execution of these tasks using a combination of threading and timed delays, making it suitable for scenarios where tasks need to be sent at precise intervals. Logging is a critical component of the script, with comprehensive logs being recorded both to the console and to a file, ensuring that all aspects of the client's operation are tracked in detail. This makes the script a robust tool for testing and evaluating real-time scheduling system, providing insights into how tasks are handled under various conditions.

The following listing presents an example of a task-set in JSON format to be used to test the real-time scheduling system capabilities. Each task (e.g., task_a and task_b) is defined by parameters such as deadline, execution_time, period, repeat, and image. The deadline specifies the maximum time allowed for task completion, while execution_time indicates the duration each task requires to execute. The period defines the interval between consecutive executions of the task, repeat specifies the number of repetitions, and image determines the Docker image to be used for executing the task.

```
{
    "task_a": {
        "deadline": 30,
        "execution_time": 10,
        "period": 30,
        "repeat": 4,
        "image":"wait_for_value:1.1.0",
        "command": "wait_for_value.py"
    },
    "task_b": {
        "deadline": 17,
        "execution_time": 5,
        "period": 15,
        "repeat": 4,
        "image":"wait_for_value:1.1.0",
        "command": "wait_for_value.py"
    }
}
```

Listing 5.10: task-set JSON structure.

For example, task_a has a thirty second deadline and a ten second execution time, with a period equal to its deadline, resulting in executions at regular thirty second intervals. On the other hand, task_b has a shorter seventeen second deadline and a five second execution time, with a fifteen second period, requiring more frequent executions. Both tasks utilize the Docker image "wait_for_value:1.1.0" and command "wait_for_value.py". The entirely code is available in the master branch: `https://github.com/rafaelcalai/real_time_distributed_cluster_client`

Chapter 6

Experimental Results

This chapter provides a detailed evaluation of the system's performance, with a particular emphasis on its capability to schedule and manage task-sets across various scenarios. The analysis is structured into several sections, each examining a distinct aspect of the system's operation, such as its ability to handle schedulable and non-schedulable task-sets, its resilience to hardware faults, and its efficiency in resource utilization.

6.1 Schedulable Task-set

To assess the system's ability to schedule a task-set as intended, the task-set was executed using the Client Simulator Script 5.6. The task-set, detailed in Table 6.1, consists of twelve tasks: six with an execution time of four seconds and a deadline of fifteen seconds, and another six with an execution time of ten seconds and a deadline of forty seconds. Each task is repeated fifty times according to its period. The setup employed two worker nodes, each capable of running up to three services in parallel. Based on this configuration and accounting for a system overhead of five seconds, the system is expected to be schedulable, with no missed deadlines or denied requests.

Table 6.1: Schedulable task-set

Task name	Execution time	Deadline	Period	Repeat
task_a	4	15	15	50
task_b	4	15	15	50
task_c	4	15	15	50
task_d	10	40	40	50
task_e	10	40	40	50
task_f	10	40	40	50
task_g	4	15	15	50
task_h	4	15	15	50
task_i	4	15	15	50
task_j	10	40	40	50
task_k	10	40	40	50
task_l	10	40	40	50

To evaluate whether the system can handle all tasks in table 6.1 without missing their deadlines, it is important to assess the overall utilization and compare it to the available capacity. For tasks a, b, c, g, h, and i, the adjusted execution time becomes nine seconds, while their period remains fifteen seconds. Thus, the utilization for each of these tasks is:

$$U = \frac{9}{15} = 0.6$$

Likewise, tasks d, e, f, j, k, and l have an adjusted execution time of fifteen seconds and a period of forty seconds. The utilization for each of these tasks is:

$$U = \frac{15}{40} = 0.375$$

The total utilization of the system can be calculated by summing the utilization of all twelve tasks:

$$U_{\text{total}} = 0.6 \times 6 + 0.375 \times 6$$

$$U_{\text{total}} = 3.6 + 2.25 = 5.85$$

Once the total utilization is computed, it should be compared to the system's total capacity. Since the system has six slots, each able to handle a utilization of 1.0, the overall capacity is:

$$\text{Total Capacity} = 6.0$$

The utilization as a percentage of the system's capacity is then calculated as:

$$\text{Utilization Percentage} = \left(\frac{5.85}{6.0}\right) \times 100 = 97.5\%$$

This calculation indicates that the system's utilization is 97.5% of its maximum capacity, suggesting that it is operating near full load but within feasible limits. Therefore, the system is expected to handle all tasks without exceeding their deadlines, provided there is no additional overhead or variability.

Figure 6.1 illustrates the expected scheduling behavior of the system, which consists of twelve tasks and two worker nodes. For simplicity, the simulation is demonstrated using only `Task_a` and `Task_d`, as the other tasks would exhibit identical scheduling patterns. The system incurs a consistent overhead of five seconds from the time a service is created and begins running in a container, as depicted by the red shaded areas. This overhead is tracked by the scheduler as the task transitions from the pending state to the running state. In this example, `Task_d` has to start executing for a short duration on forty five seconds before it can be preempted. This period represents the time required by the system to detect the transition from pending to running.

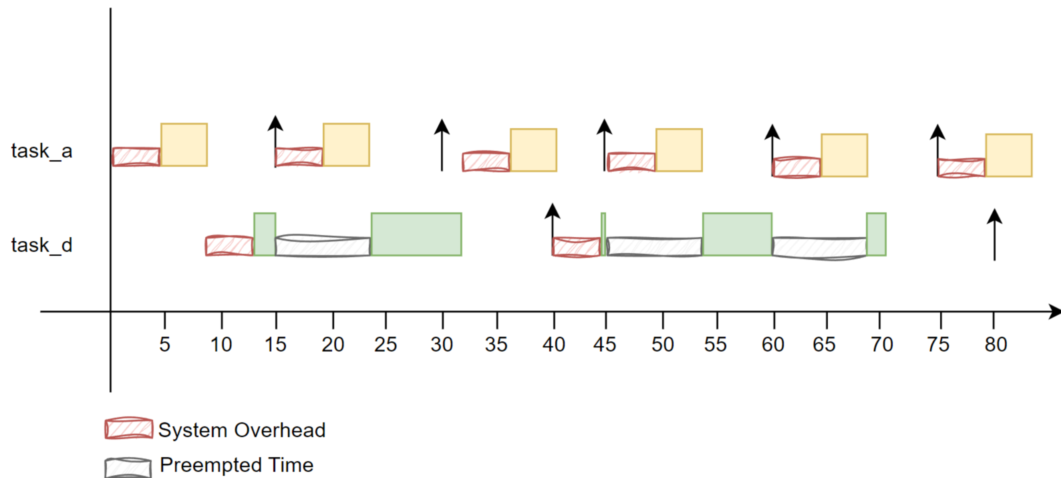


Figure 6.1: Schedulable task-set.

In the timeline, `Task_a` is scheduled periodically every fifteen seconds, with each execution block lasting four seconds, with a five second overhead. `Task_d` follows a similar pattern, with moments of preemption indicated by gray shaded areas. These preemption occur because the scheduler assigns higher priority to `Task_a` due to its shorter remaining time until the deadline. As a result, `Task_d` temporarily halts before resuming execution. The green areas represent periods of uninterrupted active execution for `Task_d`. The scheduling behavior of the other tasks would mirror this pattern, ensuring consistent task management across the entire system.

The total duration of the test was 33 minutes and 33 seconds, starting at 2024-08-31 18:18:56 and ending at 2024-08-31 18:52:29. Throughout the entire test, no missed deadlines or denied requests were observed. Due to its length, the complete log file 2024-08-31_18-18-56.log) is not included in this thesis but is available in the logs folder of the repository. `https://github.com/rafaelcalai/real_time_distributed_cluster_client`.

For tasks with a fifteen seconds deadline, the worst response time recorded was 10.18 seconds at 2024-08-31 18:27:06. The best response time for these tasks was 6.10 seconds at 2024-08-31 18:21:32. The average response time across all tasks with this deadline was 6.72 seconds. Importantly, no deadline misses were observed during the test, indicating that all tasks completed well within the fifteen second deadline.

For tasks with a forty seconds deadline, the worst response time recorded was 28.87 seconds, which occurred at 2024-08-31 18:27:25. The best response time for these tasks was 11.81 seconds, which occurred at 2024-08-31 18:34:28. The average response time for these tasks was 12.13 seconds. Similar to the tasks with the fifteen second deadline, no deadline misses were observed for tasks with the forty second deadline, demonstrating that all tasks were executed well within their required time limits.

Figure 6.2 illustrates the initial response of each task in the log file. The results differ from the simulated behavior. Notably, the average overhead for creating a service on the system was 2.5 seconds. In this test, `task_d` arrived before `task_h`, prompting the system started `task_d`. However, `task_d` was subsequently preempted by `task_h`, which had higher priority. This is evident as the elapsed time for `task_h` was extended by the overhead incurred by

```
2024-08-31 18:19:02,603 - INFO - Elapsed time for task_a request 1/50 was 6.2113s for 15s deadline with response 100
2024-08-31 18:19:02,626 - INFO - Elapsed time for task_c request 1/50 was 6.2316s for 15s deadline with response 100
2024-08-31 18:19:02,648 - INFO - Elapsed time for task_i request 1/50 was 6.2472s for 15s deadline with response 100
2024-08-31 18:19:03,014 - INFO - Elapsed time for task_b request 1/50 was 6.6217s for 15s deadline with response 100
2024-08-31 18:19:03,051 - INFO - Elapsed time for task_g request 1/50 was 6.6505s for 15s deadline with response 100
2024-08-31 18:19:04,758 - INFO - Elapsed time for task_h request 1/50 was 8.3565s for 15s deadline with response 100
2024-08-31 18:19:13,140 - INFO - Elapsed time for task_d request 1/50 was 16.7451s for 40s deadline with response 100
2024-08-31 18:19:15,142 - INFO - Elapsed time for task_e request 1/50 was 18.7449s for 40s deadline with response 100
2024-08-31 18:19:15,250 - INFO - Elapsed time for task_f request 1/50 was 18.8528s for 40s deadline with response 100
2024-08-31 18:19:15,271 - INFO - Elapsed time for task_l request 1/50 was 18.8687s for 40s deadline with response 100
2024-08-31 18:19:15,378 - INFO - Elapsed time for task_k request 1/50 was 18.9755s for 40s deadline with response 100
2024-08-31 18:19:16,899 - INFO - Elapsed time for task_j request 1/50 was 20.4970s for 40s deadline with response 100
```

Figure 6.2: Start schedulable task-set log.

`task_d`, while `task_d` exhibited a shorter elapsed time compared to other tasks with similar timing characteristics.

6.2 Non-Schedulable Task-set

To evaluate the system's capability to schedule a task-set as intended and to reject requests that could not be handled without missing deadlines, a non-schedulable task-set was executed using the Client Simulator Script (see Section 5.6). The task-set, described in Table 6.2, comprises fifteen tasks: six tasks with an execution time of five seconds and a deadline of fifteen seconds, another six tasks with an execution time of ten seconds and a deadline of thirty five seconds, and three tasks with an execution time of ten seconds and a deadline of twenty seconds. Each task was repeated one hundred times according to its respective period. The setup involved two worker nodes, each capable of running up to three services in parallel. Considering this configuration and accounting for a system overhead of five seconds, the system was expected to handle the tasks without missing deadlines while denying some requests to maintain optimal performance.

Given the theoretical overhead of five seconds, the system is considered non-schedulable because its utilization exceeds hundred percent. To determine whether the system can schedule all tasks without missing deadlines, it is crucial to evaluate the system's utilization and compare it with its available capacity. The system consists of two worker nodes, each capable of handling three tasks simultaneously, resulting in a total of six slots. The task set includes fifteen tasks with varying execution times and periods. Each task's execution time is subject to a five second overhead, which needs to be added to the original execution times. For tasks a, b, c, g, h, and i, the adjusted execution time is ten seconds, with each having a period of fifteen seconds. Therefore, the utilization for each of these tasks is calculated as:

$$U = \frac{10}{15} = 0.667$$

Similarly, tasks d, e, f, j, k, and l have an adjusted execution time of fifteen seconds and a period of thirty five seconds. The utilization for each of these tasks is:

$$U = \frac{15}{35} = 0.429$$

Tasks n, o, and p are slightly different, with an adjusted execution time of fifteen seconds and a period of twenty seconds. The utilization for each of these tasks is:

Table 6.2: Non-Schedulable task-set

Task name	Execution time	Deadline	Period	Repeat
task_a	5	15	15	100
task_b	5	15	15	100
task_c	5	15	15	100
task_d	10	35	35	100
task_e	10	35	35	100
task_f	10	35	35	100
task_g	5	15	15	100
task_h	5	15	15	100
task_i	5	15	15	100
task_j	10	35	35	100
task_k	10	35	35	100
task_l	10	35	35	100
task_n	10	20	20	100
task_o	10	20	20	100
task_p	10	20	20	100

$$U = \frac{15}{20} = 0.75$$

To determine the total system utilization, we sum the utilization of all fifteen tasks:

$$U_{total} = 0.667 \times 6 + 0.429 \times 6 + 0.75 \times 3$$

$$U_{total} = 4.002 + 2.574 + 2.25 = 8.826$$

With the total utilization calculated, it is necessary to compare this value with the system's total capacity. Given that the system has six slots, each capable of handling a utilization of 1.0, the total capacity of the system is:

$$\text{Total Capacity} = 6.0$$

To express the total utilization as a percentage of the system's capacity, the calculation is as follows:

$$\text{Utilization Percentage} = \left(\frac{8.826}{6.0}\right) \times 100 = 147.1\%$$

This result demonstrates that the system utilization is 147.1% of its total capacity. This indicates that the system is significantly over-utilized, operating well beyond its available capacity. As a result, the system is likely to face considerable difficulties in scheduling all tasks within their respective deadlines. It should lead no missed deadlines as the system should deny some tasks to ensure that the system remains operational.

The non-schedulable task-set, which ran from 2024-08-23 19:51:31 to 2024-08-23 21:11:37, exhibited specific performance metrics based on task deadlines. The total duration of the test was one hour, twenty minutes, and six seconds.

For tasks with a fifteen seconds deadline, the best response time recorded was 6.71 seconds at 2024-08-23 19:55:56, while the worst response time was 10.62 seconds at 2024-08-23 19:56:55. The average response time for these tasks was 7.11 seconds. In the case of tasks with a 20 second deadline, the best response time observed was 11.81 seconds at 2024-08-23 20:41:15, and the worst response time was 15.81 seconds at 2024-08-23 20:08:16. The average response time for these tasks was 12.13 seconds.

Finally, for tasks with a 35 seconds deadline, the best response time achieved was 11.71 seconds at 2024-08-23 20:53:57, with the worst response time being 28.53 seconds at 2024-08-23 19:53:57. The average response time for these tasks was 13.28 seconds.

Whenever a new request could have caused a missed deadline in the system, the request was denied to maintain system stability. Out of the 1500 total requests, 27 were denied to avoid missed deadlines. In terms of response times, the best response time was 0.0046 seconds at 2024-08-23 19:51:54, the worst response time was 0.21 seconds at 2024-08-23 19:51:31, and the average response time was 0.02 seconds. Throughout the entire test, no deadlines were missed, ensuring that all tasks were completed within their respective deadline constraints.Throughout the entire test, no missed deadlines were encountered, indicating that all tasks were completed within their respective deadline constraints.

6.3 Worker Node Hardware Fault

The test was conducted to evaluate the system's response in the event of a hardware fault occurring during its runtime. To simulate this scenario, a manual reset via Security Shell (SSH) was performed on Worker node 2 while it was actively executing the test. The test used the same task-set described in Table 6.1 to ensure consistency and comparability of results. The complete log file (`2024-08-29_18-09-09.log`) is available in the logs folder of the repository at `https://github.com/rafaelcalai/real_time_distributed_cluster_client`.

By inducing this fault, the objective was to observe and analyze how the system handles unexpected disruptions in a critical node during normal operation. This approach assists in assessing the robustness, fault tolerance, and overall reliability of the system under real-world conditions, where hardware failures may occur unexpectedly.

During this test, the tasks listed in Table 6.3 did not receive any responses. The first four tasks were actively executing on Worker node 2 when the manual reset was performed. The remaining tasks also failed to receive responses due to a delay in the Docker Engine API detecting that the node was no longer available. This delay caused the scheduler to continue perceiving the worker node as operational, mistakenly assigning additional tasks to it despite

Table 6.3: Request Without Response.

Time	Task request
2024-08-29 18:13:11,035	task_c request 12/50
2024-08-29 18:13:11,511	task_f request 6/50
2024-08-29 18:13:13,548	task_l request 6/50
2024-08-29 18:13:24,501	task_g request 12/50
2024-08-29 18:13:24,990	task_h request 12/50
2024-08-29 18:13:43,949	task_b request 13/50
2024-08-29 18:13:44,320	task_i request 13/50
2024-08-29 18:14:02,107	task_h request 14/50
2024-08-29 18:14:04,559	task_a request 14/50
2024-08-29 18:14:21,239	task_i request 15/50

the node having already been shut down. As a result, those tasks were sent to a non-functional node, leading to further failures. This highlights a critical need for improvement in the system's fault detection and recovery mechanisms.

During the period when Worker node 2 was down, the task requests listed in Table 6.4 were denied, as the scheduler identified that it only had one worker node available to assign services. Table 6.4 provides a log of these denied task requests, including the specific tasks ('task_g', 'task_h', 'task_b', 'task_i', and 'task_a') and their corresponding request numbers at the exact times they were attempted. This table shows that tasks were requested multiple times (e.g., 'task_h' and 'task_i'), reflecting the system's ongoing operation during the disruption.

Table 6.4: Denied task requests.

Time	Task request
2024-08-29 18:13:24,507	task_g request 12/50
2024-08-29 18:13:24,995	task_h request 12/50
2024-08-29 18:13:43,954	task_b request 13/50
2024-08-29 18:13:44,326	task_i request 13/50
2024-08-29 18:14:02,113	task_h request 14/50
2024-08-29 18:14:04,568	task_a request 14/50
2024-08-29 18:14:21,246	task_i request 15/50

When Worker node 2 returned after approximately one minute, the system resumed normal scheduling operations. Importantly, no tasks with responses missed their deadlines during this period, demonstrating the system's ability to handle some level of disruption without miss deadlines. However, one issue that occurred was the removal of the "pause/unpause" service on Worker node 2, as the container managing this service was down. This situation

indicates a need for improvement to ensure that the system remains fully functional during unexpected resets, preserving critical services even when node failures occur.

6.4 Soak Test

The soak test was performed to evaluate the system's stability and performance under continuous load conditions over an extended period of time. By running the system with a predefined set of tasks for 24 hours, the test aimed to identify any potential issues, such as missed deadlines or performance degradation, that might occur during prolonged operation. The complete log file (`2024-09-05_21-46-58.log`) is available in the logs folder of the repository at `https://github.com/rafaelcalai/real_time_distributed_cluster_client`.

During this soak test, the task set described in Table 6.1 was used. This task-set included two types of tasks with different periods: tasks with a fifteen seconds period and tasks with a forty seconds period. Over the duration of the test, the tasks with a fifteen seconds period were executed 5,760 times, while tasks with a forty seconds period were executed 2,160 times. Throughout the entire duration of the test, no deadlines were missed. The execution times for the tasks with a fifteen seconds deadline were as follows:

- **Best Execution Time**: 5.70 seconds, recorded on 2024-09-06 at 19:45:22
- **Worst Execution Time**: 13.92 seconds, recorded on 2024-09-06 at 04:51:13
- **Average Execution Time**: 6.61 seconds

Similarly, for the tasks with a forty seconds deadline, the execution times were:

- **Best Execution Time**: 12.43 seconds, recorded on 2024-09-06 at 20:13:12
- **Worst Execution Time**: 31.34 seconds, recorded on 2024-09-05 at 22:11:29
- **Average Execution Time**: 22.16 seconds

The results show that the system met all deadlines during the test, demonstrating reliable performance under sustained load. To ensure stability and prevent overload, the system selectively rejected 32 out of 47,520 requests, resulting in a rejection rate of 0.06%. This small number of rejections, guided by the scheduling algorithm, ensured all accepted tasks met their deadlines, optimizing performance. The low rejection rate confirms the scheduler's effectiveness in managing load while maintaining system stability.

6.5 Service Resources Utilization

This test is crucial for ensuring that each service strictly adheres to the resource limits allocated by the scheduler. By limiting each service to one core and 244 MB of memory, the test examines whether the system can effectively manage resource allocation in a constrained environment. It prevents any task from monopolizing resources, ensuring fair distribution across all services. As each task operates with specific execution times, deadlines, and periods, the test reveals how well the scheduler enforces these constraints and guarantees that tasks complete within the allocated CPU and memory limits without over-utilizing them.

Furthermore, this test is important for validating real-time performance in environments where resources are shared among multiple services. By simulating a high-stress scenario, it

assesses the system's ability to avoid resource contention and prevent performance degradation. The results help confirm that tasks run efficiently within the limits set by the scheduler, ensuring reliable, predictable execution, especially in systems where strict resource control is essential for maintaining stability and preventing crashes due to memory overflow or CPU overload.

Table 6.5: Stress test task-set

Task name	Execution time	Deadline	Period	Repeat
task_a	4	15	15	50
task_b	4	15	15	50
task_c	4	15	15	50
task_d	10	40	40	50
task_e	10	40	40	50
task_f	10	40	40	50

The stress test was conducted using the task set detailed in the provided table 6.5, consisting of six tasks (A through F) with varying execution times, deadlines, and periods. Tasks A, B, and C each have an execution time of four seconds, a deadline of fifteen seconds, and a period of fifteen seconds, repeating fifty times. Tasks D, E, and F, on the other hand, have a longer execution time of ten seconds, with a deadline and period of forty seconds, also repeating fifty times. This task set was executed on Worker node 02, with each task limited to a single core and 244 MB of memory, simulating a resource-constrained environment. The image used for this test was "stress-test:1.2.0," which runs a stress algorithm consuming up to three CPU cores and 180 MB of memory. The test aimed to assess the scheduler's ability to manage the tasks within these tight constraints, ensuring that each service only utilized the resources allocated to it.

Figure 6.3: Resource utilization task-set.

Figure 6.3 shows the results of a stress test being monitored on `rpi.node02` using (`docker stats`), where each service is constrained to onw CPU core and 244 MB of memory, as specified in the task set configuration. The top part of the figure lists containers running on the node, indicating that each task (`task_e`, `task_f`, and `task_b`) is consuming CPU and memory resources. The lower part of the figure shows a system monitoring tool (`top`), displaying real-time CPU and memory usage per core and process. Each core (`Cpu0` to `Cpu3`) is shown with active usage, and individual processes are reported as consuming around 25% of the CPU (indicating the use of one core, as this is a quad-core system).

Figure 6.4: Resource utilization service without limitation.

Figure 6.4 illustrates the resource usage of a single service running with the same Docker image (`stress-test:1.2.0`) but without any imposed limitations on CPU or memory. In contrast to the previous test where each service was constrained to 1 CPU core and 244 MB of memory, here the `stress` process is utilizing significantly more resources, with CPU usage exceeding 99% on multiple cores (indicating that the process can use more than one core) and consuming a substantial portion of the available memory. This comparison demonstrates that the scheduler in the previous tests effectively enforced the intended CPU and memory restrictions, as this unrestricted execution clearly shows the service consuming far more resources when no such constraints are in place.

6.6 Manager Node Resource Utilization

In the figure 6.5, the Manager node of a system is shown in an idle state. The output shows that the system is not processing any significant tasks, as indicated by the low CPU usage percentages in the `top` command output. The Docker container statistics also reflect minimal resource usage, with most containers consuming negligible CPU and memory resources. Notably, the `RT_sched` process is consuming an average of nine percent of the CPU, which is relatively low and consistent with an idle state where background scheduling processes may still be running. The overall system load averages are low, and the memory usage remains well within limits, suggesting that no requests or computationally intensive tasks are currently being handled by the node.

```
controller@rpi5-calai02-controller: ~/app/logs 205x7
CONTAINER ID   NAME                                           CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O        PIDS
adb40990e817   Income_Manager.1.7hac6760u97yeurqn2ebs77o1     0.00%     6.867MiB / 3.834GiB   0.17%     1.14MB / 1.23MB   0B / 1.27MB      3
7ae6153601dc   RT_Cluster_Scheduler.1.py15lrnucqc5vqnpaxhfiejcv   8.99%     21.46MiB / 3.834GiB   0.55%     889kB / 597kB     0B / 1.16MB      6

controller@rpi5-calai02-controller: ~ 205x39
top - 22:35:00 up  6:48,  2 users,  load average: 0.43, 0.31, 0.23
Tasks: 142 total,   1 running, 141 sleeping,   0 stopped,   0 zombie
%Cpu0  :   4.0/0.7      5[
%Cpu1  :   3.7/1.3      5[
%Cpu2  :   3.7/1.0      5[
%Cpu3  :   4.0/1.0      5[
MiB Mem :   3925.8 total,   2323.2 free,    509.8 used,   1183.8 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   3416.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    790 root      rt   0 2920776 140496  53980 S  10.3   3.5  45:24.57 dockerd
  10529 root      20   0  414272  32000  11648 S   8.6   0.8  24:47.07 python
  10676 root      20   0  719896  11784   8192 S   0.3   0.3   0:07.40 containerd-shim
  25606 root      20   0       0      0      0 I   0.3   0.0   0:00.19 kworker/2:3-events
  26225 control+  20   0    9160   4736   2688 R   0.3   0.1   0:00.29 top
  26481 control+  20   0 1696592  25088  16384 S   0.3   0.6   0:00.05 docker
      1 root      20   0  169412  12304   7952 S   0.0   0.3   0:05.72 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.02 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 slub_flushwq
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
     11 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     12 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_kthread
     13 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_rude_kthread
     14 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_trace_kthread
     15 root      20   0       0      0      0 S   0.0   0.0   0:00.23 ksoftirqd/0
     16 root      20   0       0      0      0 I   0.0   0.0   0:01.03 rcu_preempt
     17 root      rt   0       0      0      0 S   0.0   0.0   0:00.05 migration/0
     18 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/0
     19 root      20   0       0      0      0 S   0.0   0.0   0:00.00 cpuhp/1
```

Figure 6.5: Resource utilization Manager node in idle.

In contrast, the second figure 6.6 demonstrates the system in a runtime, where it is handling tasks or undergoing a stress test. The CPU usage across all cores is considerably higher, with some cores reaching over ten percent utilization. Notably, the `RT_sched` process is consuming around seventeen percent of the CPU, further indicating the system's active state. Additionally, memory usage is elevated compared to the idle state, though it remains manageable. These observations clearly indicate that the system is actively processing tasks or simulating a high-load scenario, effectively showcasing the difference between an idle and a runtime state of the manager node.

```
controller@rpi5-calai02-controller: ~/rt_cluster_scheduler/docker_stress_test 205x11
CONTAINER ID   NAME                                           CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O     PIDS
adb40990e817   Income_Manager.1.7hac6760u97yeurqn2ebs77o1     0.10%     7.047MiB / 3.834GiB   0.18%     872kB / 937kB     0B / 950kB    8
7ae6153601dc   RT_Cluster_Scheduler.1.py15lrnucqc5vqnpaxhfiejcv   17.39%    21.71MiB / 3.834GiB   0.55%     588kB / 359kB     0B / 889kB    6

controller@rpi5-calai02-controller: ~ 205x27
top - 18:55:17 up  3:08,  3 users,  load average: 0.58, 0.44, 0.42
Tasks: 146 total,   1 running, 145 sleeping,   0 stopped,   0 zombie
%Cpu0  :   6.7/2.0      9[
%Cpu1  :   7.1/2.0      9[
%Cpu2  :   5.5/2.0      8[
%Cpu3  :   7.7/2.0     10[
MiB Mem :   3925.8 total,   2310.1 free,    530.4 used,   1176.3 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   3395.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    790 root      rt   0 2920520 145368  53724 S  19.9   3.6  21:58.23 dockerd
  10529 root      20   0  414272  32000  11648 S  15.6   0.8   5:29.80 python
  18738 control+  20   0 1770384  26676  16768 S   0.7   0.7   0:00.37 docker
    714 root      20   0 1418460  43592  27648 S   0.3   1.1   0:14.72 containerd
  10508 root      20   0  719896  11420   8192 S   0.3   0.3   0:03.54 containerd-shim
  10676 root      20   0  719896  12380   8192 S   0.3   0.3   0:03.37 containerd-shim
  16774 root      20   0       0      0      0 I   0.3   0.0   0:00.18 kworker/3:0-events
  19026 control+  20   0    9152   4736   2688 R   0.3   0.1   0:00.47 top
      1 root      20   0  169264  12176   7952 S   0.0   0.3   0:05.46 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.01 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 slub_flushwq
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
     11 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     12 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_kthread
     13 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_rude_kthread
```

Figure 6.6: Run-time resource utilization Manager node.

Chapter 7

Conclusions

This thesis focused on the development of a real-time scheduling algorithm for Docker Swarm, aimed at overcoming the challenges of meeting strict temporal requirements in distributed critical systems. The core objective was to enhance Docker Swarm's ability to orchestrate tasks under time constraints, addressing a growing need in fields like autonomous vehicles, healthcare, and industrial automation, where even minor delays can have severe consequences.

The research gap identified lay in the inadequacies of existing container orchestration platforms, such as Docker Swarm, Kubenetes, which were designed primarily for cloud environments and lack the necessary support for real-time performance. These platforms, while effective for general distributed computing, fall short in guaranteeing the strict timing, fault tolerance, and resource optimization that critical systems demand. By focusing on this gap, the thesis aimed to integrate real-time scheduling capabilities into Docker Swarm to bridge this critical shortfall.

The results achieved through this research were significant. The newly developed algorithm successfully extended Docker Swarm's scheduling capabilities, ensuring that tasks could meet their deadlines while optimizing the use of available resources. Rigorous experimental testing demonstrated the system's capacity to handle complex task management, including preemption and load balancing, in distributed environments. The results showed reliable performance even in scenarios with fluctuating workloads and network conditions. However, challenges such as scalability, fault tolerance, and multi-manager node support remain areas for future improvement.

7.1 Practical and Theoretical Implications

The practical implications of this research are substantial, particularly for industries that require strict real-time performance, such as autonomous vehicles, healthcare, traffic management, and train traffic control. In autonomous systems, real-time scheduling can enhance decision-making and safety-critical operations like collision avoidance and navigation. In healthcare, this system can ensure the reliable operation of medical devices where delays could lead to life-threatening consequences. In traffic management and train traffic control, real-time orchestration is crucial to avoid collisions, optimize routes, and ensure smooth operations across complex networks. Moreover, the distributed nature of the proposed solution allows edge devices, which often have limited processing power, to offload computation to other nodes in the system. This can significantly reduce costs by distributing tasks across multiple devices, while the combination of various data sources in real-time can even enable the development of new features or capabilities for these systems. The algorithm's ability

to optimize resource allocation also makes the system more efficient and cost-effective in various real-world applications.

From a theoretical perspective, this research contributes to the discussion of real-time performance in distributed systems. By integrating real-time scheduling into Docker Swarm, the study challenges traditional container orchestration models that prioritize load balancing over timing requirements. This opens new avenues for developing orchestration frameworks that better meet the demands of edge and fog computing environments, where distributed solutions leverage a combination of limited-edge and more powerful devices to create a more robust and flexible system.

7.2 Limitations

This thesis achieved remarkable results with an algorithm based on EDF, capable of handling several requests without missing deadlines. However, to make it feasible for a real world application, several features need to be implemented or improved as outlined below:

- **Improve Fault Tolerance**: As noted in the tests described in Section 6.3, when a worker node experiences a hardware fault, the system is currently unable to reschedule tasks or notify the client that the deadline cannot be met. Enhancing the system's fault tolerance is crucial to ensure reliability and timely notifications in such scenarios. Additionally, the system should be capable of properly recovering and managing the pause/unpause service, which is currently not functioning as expected.
- **Implement Multi Manager Node Capability**: Enhancing the system to support multiple manager nodes would significantly increase its reliability. These manager nodes could employ a leader election mechanism, using the built-in functionality of a Docker Swarm to determine the current leader. The number of manager nodes should always be an odd number (e.g., 3, 5, up to 7) to ensure consensus. Additionally, these nodes should be physically distributed across different networks, power supplies, and Ethernet connections to increase the likelihood of withstanding a power outage or cyberattack. For this to be possible, information about waiting and running queues must be shared among the manager nodes.
- **Implement Manager Node Recovery Mechanism**: To minimize system disruption when a manager node resets, a recovery mechanism should be implemented. This could be achieved by retaining critical information in files rather than storing it solely in memory. This approach would allow the manager node to recover and resume operations with minimal impact on the overall system.
- **Reduce System Overhead**: There are several ways to improve the system's performance overhead. One approach is to use programming languages that offer better performance than Python, such as Go, C++, or, preferably, Rust, which is more suitable for critical systems.
- **Replace Docker Engine API**: Consider replacing the Docker Engine API with an implementation that prioritizes performance and provides greater control over the API. This change would improve not only performance but, more importantly, predictability. With the current API, response times vary, making it difficult to achieve predictable system behavior.

7.3 Future Work

While this thesis has presented a robust algorithm based on EDF that effectively handles multiple requests without missing deadlines, there remain several areas that could be explored further to enhance the system's applicability in real world scenarios. Future work could focus on addressing the following improvements:

- **Enhancing Fault Tolerance**: Improve the system's fault tolerance to handle hardware faults more gracefully. This includes implementing mechanisms to automatically reschedule tasks when a worker node fails and providing timely notifications to clients about potential deadline breaches. Furthermore, improving the ability to recover from faults in the pause/unpause service is critical to ensuring continuous service availability and reliability.
- **Support for Multiple Manager Nodes**: A significant extension to the current system would be enabling support for multiple manager nodes. Future work could implement a distributed management model with leader election mechanisms to ensure system reliability and fault tolerance, making the system more robust against failures, blackouts, or cyberattacks. To achieve this, synchronization of waiting and running queues between manager nodes would be necessary.
- **Manager Node Recovery Mechanisms**: To minimize disruption during unexpected resets of manager nodes, future work could focus on developing a recovery mechanism that allows the system to resume with minimal impact. This could involve persisting critical state information in a more durable form, such as on disk or in a distributed file system, rather than relying solely on memory.
- **Performance Optimization and Overhead Reduction**: To enhance performance, future work should investigate replacing the current implementation with more efficient programming languages, such as Go, C++, or Rust. Rust, in particular, is well suited for critical systems due to its performance and safety features. Additionally, optimizing the overall system architecture to reduce overhead and latency will be an essential part of improving the responsiveness and scalability of the system.
- **Replacing Docker Engine API for Predictability and Performance**: Future work should also consider replacing the Docker Engine API with a more performance oriented solution that offers better predictability and control. This would allow for more precise management of containerized services, reducing variability in response times and ensuring more consistent system behavior under various loads.
- **Transition from Docker Swarm to Kubernetes**: Another potential area of improvement involves transitioning from Docker Swarm to Kubernetes, which is a more comprehensive and market aligned solution for container orchestration. Kubernetes provides a wide range of built-in features, such as automatic scaling, advanced networking capabilities, and self-healing, which would enhance the system's reliability, flexibility, and manageability. Adopting Kubernetes could also better align the system with industry standards, making it more suitable for deployment in modern, large-scale environments.
- **Investigating Alternative Scheduling Methods**: Although this thesis focuses on an EDF algorithm, future research could explore alternative scheduling algorithms that may provide better performance or flexibility in different contexts. Options like LLF, RMS, or hybrid approaches combining several scheduling techniques could be

evaluated for their effectiveness in handling diverse workloads, reducing latency, or improving resource utilization. Exploring machine learning based scheduling methods, where algorithms adapt dynamically based on historical data and current workloads, could also offer more intelligent and efficient scheduling solutions.

- **Developing a User Interface for Configuration Management**: To enhance usability and system management, future work could include developing a User Interface (UI) that allows for easy configuration and management of the system. This UI could provide functionalities to add or remove worker nodes, disable or enable specific nodes, add new Docker images, and reconfigure services assigned to each worker node. Such an interface would simplify the management process, making it more accessible to users who may not be familiar with commandline tools or underlying system details, ultimately improving operational efficiency and adaptability.
- **Implementing Dynamic Resource Usage for Docker Images**: A more dynamic approach to resource usage could greatly enhance the efficiency and flexibility of the system. Future work could involve implementing mechanisms that dynamically adjust the number of services running on each worker node based on the specific services in use and their resource requirements. This would involve monitoring the resource consumption of each Docker image and dynamically scaling services up or down on worker nodes to optimize resource usage, reduce bottlenecks, and ensure that high demand services have adequate resources available. This approach could lead to better utilization of hardware resources and improved overall system performance.

In conclusion, this research not only tackled a crucial gap in real-time container orchestration but also established a strong foundation for future advancements in the field. The developed solution demonstrates the potential to significantly enhance orchestration platforms, making them better equipped to support critical, time-sensitive applications across various industries. By addressing the unique demands of real-time systems, this work contributes meaningfully to the broader evolution of edge and fog computing, paving the way for more efficient, scalable, and reliable distributed systems in environments where strict timing and resource optimization are essential.

Bibliography

[1] J.C. Knight. "Safety Critical Systems: Challenges and Directions". In: *IEEE* 1007998.12 (May 2002), pp. 4477–4479. url: https://ieeexplore.ieee.org/document/1007998.

[2] David J Smith and Kenneth GL Simpson. *Safety Critical Systems Handbook*. Butterworth-Heinemann, 2016.

[3] *symmetry electronics*. Last accessed on February 02, 2024. url: https://www.symmetryelectronics.com/blog/fog-computing-vs-edge-computing/.

[4] Yogeswaranathan Kalyani and Rem Collier. "A Systematic Survey on the Role of Cloud, Fog, and Edge Computing Combination in Smart Agriculture". In: *Sensors* 21.17 (2021). issn: 1424-8220. doi: 10.3390/s21175922. url: https://www.mdpi.com/1424-8220/21/17/5922.

[5] Microsoft. *What is cloud computing*. Last accessed on January 2, 2024. url: https://azure.microsoft.com/pt-pt/resources/cloud-computing-dictionary/what-is-cloud-computing.

[6] Rajkumar Buyya Chii Chang Satish Narayana Srirama. *Internet of Things (IoT) and New Computing Paradigms*. Wiley, 2019.

[7] Chunchi Liu Yinhao Xiao Yizhen Jia and Xiuzhen Cheng. "Edge Computing Security: State of the Art and Challenges". In: *PROCEEDINGS OF THE IEEE | Vol. 107, No. 8* (2019).

[8] Václav Struhár et al. "REACT: Enabling Real-Time Container Orchestration". In: (2021), pp. 1–8. doi: 10.1109/ETFA45728.2021.9613685.

[9] Václav Struhár et al. "Hierarchical Resource Orchestration Framework for Real-time Containers". In: *ACM Trans. Embed. Comput. Syst.* 23.1 (Jan. 2024). issn: 1539-9087. doi: 10.1145/3592856. url: https://doi.org/10.1145/3592856.

[10] Luis Miguel Pinho Maria A. Serrano Cesar A. Marin. "An Elastic Software Architecture for Extreme-Scale Big Data Analytics". In: *Springer* (2022).

[11] *COMP Superscalars*. Last accessed on February 04, 2024. url: https://www.bsc.es/research-and-development/software-and-apps/software-list/comp-superscalar/documentation.

[12] Shahidullah Kaiser et al. "Container Technologies for ARM Architecture: A Comprehensive Survey of the State-of-the-Art". In: *IEEE Access* 10 (2022), pp. 84853–84881. doi: 10.1109/ACCESS.2022.3197151.

[13] James Turnbull. *The Docker Book: Containerization is the New Virtualization*. Self-published, 2014.

[14] Amazon. *What is Docker?* Last accessed on January 03, 2024. url: https://aws.amazon.com/pt/docker/?nc1=h_ls.

[15] *Docker overview*. Last accessed on august 17, 2024. url: https://docs.docker.com/guides/docker-overview/.

[16] Emiliano Casalicchio. "Container Orchestration: A Survey". In: *Systems Modeling: Methodologies and Tools*. Ed. by Antonio Puliafito and Kishor S. Trivedi. Cham: Springer International Publishing, 2019, pp. 221–235. isbn: 978-3-319-92378-9. doi:

10.1007/978-3-319-92378-9_14. url: https://doi.org/10.1007/978-3-319-92378-9_14.

[17] *Kubernetes*. Last accessed on 10 de janeiro de 2024. url: https://kubernetes.io/.

[18] *Docker Swarm*. Last accessed on January 10, 2024. url: https://docs.docker.com/engine/swarm/swarm-tutorial/.

[19] *Nomad*. Last accessed on January 10, 2024. url: https://www.nomadproject.io/.

[20] *Mesos*. Last accessed on January 10, 2024. url: https://mesosphere.github.io/marathon/.

[21] *Docker Swarm*. Last accessed on June 22, 2024. url: https://docs.docker.com/engine/swarm/.

[22] *Docker SDK for Python*. Last accessed on august 12, 2024. url: https://docker-py.readthedocs.io/en/stable/.

[23] Arezou Mohammadi and Selim G. Akl. "Scheduling Algorithms for Real-Time Systems". In: *School of Computing Queen's University Kingston, Ontario Canada K7L 3N6* 21 (2005).

[24] Neil C. Audsley et al. "Fixed Priority Pre-emptive Scheduling: An Historical Perspective". In: *Real-Time Systems* 8.2 (Mar. 1995), pp. 173–198. issn: 1573-1383. doi: 10.1007/BF01094342. url: https://doi.org/10.1007/BF01094342.

[25] Jens Hildebrandt, Frank Golatowski, and Dirk Timmermann. "Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems". In: *Journal of Real-Time Systems* (2002).

[26] *introducing Raspberry PI 5*. Last accessed on august 11, 2024. url: https://www.raspberrypi.com/news/introducing-raspberry-pi-5/.