N° d'ordre :
N° de série :

PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
Ministry of Higher Education and Scientific Research

ECHAHID HAMMA LAKHDAR UNIVERSITY - EL OUED
FACULTY OF EXACT SCIENCES
Computer Science department

End of Study Memory
Presented for the Diploma of

ACADEMIC MASTER

Domain : **Mathematics and Computer Science**
spinneret: **Computer Science**
Speciality : **Artificial Intelligence and Distributed Systems**

Presented by :

- **Khadraoui Khouloud**
- **Melazem Rym**

Theme

Automatic Container Scaling in Fog Computing

defended in: 08-06- 2023 In front of jury:

Dr.	Ben Ali Abd Alkamel	President
Dr.	Khelaifa Abdennacer	Reporter
Mr.	Bali Mouadh	Supervisor

Année Universitaire: 2022/2023

ABSTRACT

Fog Computing complements Cloud Computing in meeting the specific requirements of the IoT. It enables local data processing near IoT devices, reducing latency and improving real-time responsiveness. However, managing scalability in a Fog environment can be a challenge. Infrastructure management and orchestration tools play a crucial role in addressing this challenge by ensuring seamless deployment, monitoring, and scalability of services in a Fog infrastructure.

In this work, We proposed an auto-scaling algorithm specifically designed for Fog Computing infrastructure that utilize a containerized architecture and orchestration framework. This algorithm addresses the need to efficiently manage resources and dynamically adjust the number of containers based on workload demands. Our algorithm incorporates capacity planning to determine the maximum and minimum number of services replicas (containers) that can be deployed based on available resources, performance requirements, and workload in dynamical way.

Our solution is implemented on a system managing a service based on Docker and SWARM Orchestration frameworks, including the use of other tools such as, Prometheus, cAcvisor, Apache Bench. Finally, the experimental results have a significant impact on the availability, waiting time, processing, and total time(response time), of a service charged constantly with an increasing workload.

key words: Container , Fog Computing , Docker swarm , Scalability

الملخص

تكمل الحوسبة الضبابية الحوسبة السحابية في تلبية المتطلبات المحددة لإنترنت الأشياء. إنه يتيح معالجة البيانات المحلية بالقرب من أجهزة إنترنت الأشياء ، مما يقلل من زمن الوصول ويحسن الاستجابة في الوقت الفعلي. ومع ذلك ، فإن إدارة قابلية التوسع في بيئة الضباب يمكن أن يمثل تحديًا. تلعب إدارة البنية التحتية وأدوات التنسيق دورًا حاسمًا في مواجهة هذا التحدي من خلال ضمان النشر السلس ، والمراقبة ، وقابلية التوسع للخدمات في البنية التحتية للضباب.

في هذا العمل ، اقترحنا خوارزمية التوسيع تلقائي لاستخدام الموارد ,وهي مصممة خصيصًا للبنية التحتية للحوسبة الضبابية التي تستخدم بنية حاوية وإطار عمل منسق . تتناول هذه الخوارزمية الحاجة إلى توسيع الموارد بكفاءة وضبط عدد الحاويات ديناميكيًا بناءً على متطلبات عبء العمل على الخدما ت . تتضمن الخوارزمية الخاصة بنا تخطيط السعة لتحديد الحد الأقصى والحد الأدنى لعدد النسخ المتماثلة للخدمات (الحاويات) التي يمكن نشرها بناءً على الموارد المتاحة ومتطلبات الأداء وعبء العمل بطريقة ديناميكية.

تم تنفيذ حلنا على نظام يدير خدمة يعتمد على أطر عمل Docker و Swarm Orchestration ، بما في ذلك استخدام أدوات أخرى مثل Prometheus و cAcvisor و Apache Bench.

أخيرًا ، إن النتائج التجريبية كان لها تأثير كبير على التوافر ووقت الانتظار والمعالجة والوقت الإجمالي (وقت الاستجابة) لخدمات محملة باستمرار بعبء عمل متزايد.

الكلمات المفتاحية: الحاوية ، حوسبة الضباب ، Docker Swarm ، قابلية التوسع .

KNOWLEDGMENT

“In the name of Allah, the most Beneficent, the most Merciful, efforts are nothing without his will. All praises to Allah who gave us strength to continue this work, many thanks to him and his last prophet Mohamed (peace be upon him)”

We would first like to thank our supervisor ***Bali Mouadh*** *for his invaluable effort. Thank you for your patience, support and every piece of advice.*

we would like also to thank ***Bali Ahmed*** *and* ***Ghenabzia Ahmed*** *for their assistance and advice .*

Our dearest appreciation to our families for their sacrifice and their support during this long journey.

Thank you.

Contents

List of Figures

LIST OF ALGORITHMS

Listings

List of Tables

GENERAL INTRODUCTION

IoT technology enables smart devices to connect to the internet, forming a network of interconnected devices. In a typical IoT application architecture, wireless sensors or autonomous vehicles are connected to a central cloud through which an IoT application server collects data from and sends instructions to these devices. However, this architecture often experiences significant latency between the server and devices, as well as high traffic load on the backhaul network. To address these issues, especially when low latency is crucial or when a large number of IoT devices are involved, it becomes necessary to deploy IoT servers in close proximity to the devices themselves. This requirement has led to the emergence of fog computing as a solution that brings computational resources and data processing capabilities closer to IoT devices, thereby reducing latency and alleviating network congestion.

On the other hand, Fog computing extends cloud services to the edge of the internet, bringing them closer to cloud users. In the context of Infrastructure as a Service (IaaS), cloud services are typically provided through virtual machines (VMs) or containers. Containers are a lightweight virtualization technology that utilizes cgroups and Linux namespaces to isolate the execution environment of applications. Compared to VMs, containers consume fewer resources, have faster loading and starting times, and are easier to manage and control. As a result, container technology has been increasingly utilized in various approaches to build IoT platforms. These platforms leverage containers' advantages to enhance the efficiency and flexibility of IoT applications at the edge of the network.

Besides that, Container-based fog computing architectures require scalability to meet the evolving demands of edge computing environments. With the increasing number of edge devices

and the need for distributed computing, containers provide a lightweight and flexible framework for deploying applications at the edge. Scalability is crucial to ensure that container-based fog computing architectures can handle the growing volume of data generated by edge devices and support latency-sensitive applications. By scaling the fog computing infrastructure, the resource allocation system needs to dynamically allocate resources, such as CPU, memory, and storage, to accommodate varying workloads. This scalability enables efficient load balancing, fault tolerance, and optimal resource utilization across distributed fog nodes.

Also, Scalability refers to a computer infrastructure's ability to handle increasing workloads efficiently, ensuring high availability and optimal resource utilization. There are two categories of scaling methods: vertical scaling and horizontal scaling. Vertical scaling involves adding more resources (such as disk, memory, CPU cores) to existing worker nodes, while horizontal scaling entails adding new worker nodes to the system. Conversely, vertical and horizontal down-scaling involve reducing resources. For a web application to be scalable, it should effectively handle a growing or decreasing number of user requests without significant downtime. Scalability often arises as an infrastructure issue, where insufficient compute resources can lead to performance degradation when faced with increased workloads.

Motivation

In this work, we consider the Auto-scaling process as a mechanism that adapts to the real-time resource demands of running applications by dynamically adjusting the number of resources allocated to them. However, effectively implementing auto-scaling requires continuous monitoring of resources per service and autonomous decision-making on scaling without human intervention. The primary objective of this thesis is to present a container-based architecture for fog computing, orchestrated by an automated scaling mechanism using Docker Swarm, capable of efficiently managing resources in a fog computing environment. The motivation behind this research stems from the desire to create a dynamic system that can automatically scale and optimize container resources within a fog computing environment. By implementing this approach, we aim to enable the system to seamlessly adapt to evolving computing demands, enhance overall performance, and ensure the delivery of seamless and responsive services.

Problem Statement

The Containerization frameworks lack an official mechanism to address the challenge of the dynamic and automated services scaling. For instance, in the case of the Docker Swarm environment, service providers have the option to manually scale services by adjusting the number of containers through a specific API (Docker CLI) initiated by the service provider.

The most reliable approach to address the fluctuating workload in a system is to scale up or down the available resources, utilizing vertical or horizontal scaling techniques within the system's services. Unfortunately, Docker Swarm lacks an inherent capability for automatically scaling services. Additionally, the absence of a well-established third-party solution often leads developers to develop their autos-scalars to achieve an automatic scaling. Our work focuses on tackling the challenges of automatic service scalability by dynamic infrastructure scalability in Fog Computing system. The monitoring of this system is within containerized environment managed by Docker Swarm.

Outline

The rest of this thesis is organized as follows:

- **Chapter II** This chapter offers a comprehensive overview of IoT and various computing paradigms.
- **Chapter III** offers a comprehensive overview of the technologies and concepts crucial for fog computing application development.
- **Chapters IV** an overview of the system environment and its components.
- **Chapter V** describes the establishment of a functional prototype and the experimentation that was conducted to evaluate its effectiveness.

CHAPTER I

INTERNET OF THINGS AND COMPUTING PARADIGMS

I.1 Introduction

In this chapter, we provide an overview of the key concepts and technologies related to the Internet of Things (IoT), cloud computing, edge computing, and fog computing. We start by defining IoT and describing its architecture and communication models. Next, we discuss cloud computing, including its characteristics and benefits. We then introduce edge computing, its definition, architecture, and characteristics. Finally, we discuss fog computing, including its definition, architecture, and resource management techniques. .

I.2 Internet of Things IoT

I.2.1 Definition

The Internet of Things (IoT) is a concept that describes a global network of intelligent objects, capable of autonomously organizing, sharing information, data, and resources. These objects can react and adapt to changes in their environment, forming a comprehensive network where human-to-human, human-to-things, and things-to-things communication takes place. Through the integration of sensors and actuators, physical objects can sense their surroundings, communicate data, and contribute to a vast pool of information for analysis. The IoT enables a new level of connectivity and intelligence, where everyday objects become machine-readable and traceable on the Internet, facilitating understanding, responsiveness, and automation. The deployment of IoT systems brings about significant advancements with reduced human intervention, revolutionizing the way we interact with and harness the potential of interconnected physical information systems.[5]

I.2.2 Architecture

The Internet of Things (IoT) architecture typically consists of three layers: the Perception layer, Network layer, and Application layer. However, some variations incorporate two additional layers: the Middleware layer and the Business layer.This five layerarchitecture is described in figure I.1:

- **The perception layer**, also known as the physical layer, consists of sensors that sense and gather information about the environment. It can collect data related to physical parameters

such as temperature, humidity, light, and also identify other smart objects present in the environment. The sensors in this layer provide input to the IoT system by capturing real-world data.[1]

- **The network layer** is responsible for transferring information from smart objects in the perception layer to the middleware layer for further processing. It establishes communication channels and utilizes various wired or wireless technologies such as 3G, RFID, ZigBee, NFC, etc., to facilitate data transmission between devices. The network layer ensures connectivity and enables data exchange between different components of the IoT system.[6]

- **The middleware layer** plays a crucial role in storing, analyzing, and processing the large volume of data collected from the network layer. It acts as an intermediary between the lower layers and the upper application layer. The middleware layer employs technologies such as databases, cloud computing, and big data processing modules to manage and provide a diverse set of services. It enables data storage, data transformation, and the implementation of various IoT functionalities.[1]

- **The application layer** provides the services and functionalities requested by users or specific use cases. It utilizes the processed data from the middleware layer to deliver relevant information or perform specific tasks. The application layer can cover a wide range of domains, including smart homes, buildings, transportation systems, industrial automation, and more. It leverages the data and capabilities of the IoT system to offer tailored services to end-users.[7]

- **The business layer** is responsible for managing the entire IoT system, including the applications and services provided. It involves defining business models, creating graphs and flowcharts based on the data received from the application layer, and overseeing the overall functioning and optimization of the IoT system. The business layer ensures that the IoT system aligns with the organization's goals, objectives, and strategies.[7]

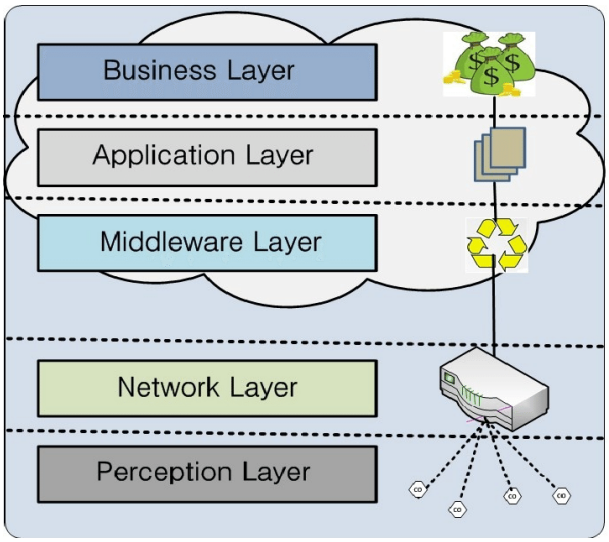

Figure I.1: Internet of Things layers [1]

I.2.3 Communications Models

The Internet Architecture Board (IAB) has outlined four common communication models used by IoT devices. These models provide a technical perspective on how IoT devices connect and communicate. Here are the four models:

- **Device-to-Device Communications:** In this model, IoT devices directly communicate with each other without the need for intermediate entities. Devices can share data, exchange commands, and collaborate locally within their network. This model is useful in scenarios where devices need to interact in a peer-to-peer manner, such as in smart home systems or industrial automation.[8]

- **Device-to-Cloud Communications:** In this model, IoT devices send data and receive commands through the internet to a cloud-based platform. Devices collect sensor data, process it locally (if capable), and then transmit the data to the cloud for further analysis, storage, and processing. Cloud-based applications can provide insights, perform data analytics, and enable remote control and management of IoT devices.[8]

- **Device-to-Gateway Model:** In this model, IoT devices communicate with a gateway device that acts as an intermediary between the devices and the cloud or back-end systems. The gateway aggregates data from multiple devices, performs data preprocessing or filtering, and then forwards the relevant data to the cloud or other back-end systems. Gateways can also enable local control and automation, reducing latency and offloading some processing tasks from the cloud.[9]

- **Back-End Data-Sharing Model:** In this model, data collected from IoT devices is shared and exchanged among different back-end systems or services. Multiple cloud-based applications, databases, or platforms can access and utilize the IoT data to perform specific tasks or provide specialized services. This model facilitates interoperability and data reuse across different applications or services.[9]

I.3 Cloud Computing

I.3.1 Definition

Cloud computing is a model of delivering on-demand computing resources over the internet, including servers, storage, databases, software, and more, with pay-as-you-go pricing. It enables organizations to access resources quickly and easily, without investing in expensive hardware and infrastructure. Cloud computing has several deployment models, including public, private, and hybrid clouds, each with its own advantages and challenges. Public clouds are provided by third-party cloud service providers and offer scalability, affordability, and ease of use, but may not meet specific security and compliance requirements. Private clouds are hosted internally or by a third-party provider and offer greater control, security, and customization, but require significant investment and maintenance. Hybrid clouds combine elements of both public and private clouds to offer flexibility and cost-effectiveness. Cloud computing also offers several service models, including Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS), each offering different levels of control and management.[10]

I.3.2 Characteristics

Cloud computing offers a range of characteristics that make it a highly desirable technology.

- **On-demand self-service** is an essential characteristic of cloud computing systems. It allows consumers to automatically allocate various cloud services, including computing power (CPU time), storage, network access, server time, web applications, and more, as needed without the need for human interaction. [11]

- **Cost effectiveness** Cloud service providers offer services that are highly cost effective, and in some cases, even free. The billing model typically follows a pay-as-you-go approach, where users are charged based on their actual resource usage. This eliminates

the need for users to invest in purchasing and maintaining costly infrastructure, leading to significant cost savings. [12]

- **Broad network access**, or mobility, enables consumers to access cloud resources via the Internet from any location and at any time. This ubiquitous access is available on various devices such as mobile phones, laptops, and PDAs. Users can seamlessly connect to and utilize cloud services, fostering flexibility and convenience in accessing and utilizing cloud resources.[11]

- **Resource pooling** involves consolidating physical and virtual computing resources within the cloud, independent of their specific location. Customers have no control or knowledge of the exact resource location, as they are abstracted and managed collectively within the cloud infrastructure. This pooling enables efficient resource utilization and scalability for cloud services.[12]

- **Rapid elasticity** allows for the quick and flexible provisioning and release of computing resources based on consumer demand. These resources are perceived as infinite and can be obtained in any quantity at any time. This characteristic provides scalability and adaptability to meet changing workload requirements in the cloud.[12]

- **Customization** in the cloud refers to the ability to adapt and tailor the environment, including infrastructure and applications, according to user demands. The cloud provides a reconfigurable platform where customization can be implemented dynamically in response to changing requirements.[13]

- **Efficient** resource utilization in the cloud is achieved by providing resources for the required duration, ensuring optimal usage of these resources. This approach minimizes resource wastage and enhances overall efficiency in resource allocation.[13]

- **Virtualization** in cloud computing enables users to access services from anywhere using various devices, as resources are obtained from the cloud rather than a specific entity. Users can securely accomplish tasks conveniently through network services on laptops or mobile phones. This flexibility allows for collaborative and complex tasks that surpass the capabilities of a single computer.[14]

- **reliability**,Cloud computing ensures reliability through the utilization of multiple redundant sites, making it an ideal solution for business-critical tasks and disaster recovery. The

availability of redundant infrastructure enhances reliability and minimizes the risk of data loss or service disruption.[13]

- **Multitenancy**in cloud computing enables simultaneous provision of services to multiple users. These users share cloud resources at various levels, such as the network, host, and application. However, each user operates within their isolated and personalized virtual application instance, ensuring privacy and customization.[14]

- **scalability**,Cloud computing exhibits high scalability, allowing for the easy expansion of its infrastructure. Cloud providers can seamlessly incorporate additional nodes and servers into the cloud environment with minimal adjustments to the existing infrastructure and software. This scalability feature enables efficient handling of increasing workloads and accommodates the growing demands of users.[13]

I.4 Edge Computing

I.4.1 Definition

Edge computing refers to the utilization of enabling technologies that enable computation to be performed at the edge of the network. It involves processing downstream data for cloud services and upstream data for IoT services. In this context, the term ”edge” encompasses computing and network resources located along the path between data sources and cloud data centers. For instance, smartphones act as the edge between body-worn devices and the cloud, gateways in smart homes serve as the edge between home devices and the cloud, and Micro Data Centers (MDCs) or Cloudlets act as the edge between mobile devices and the cloud. The fundamental principle behind edge computing is that computation should occur in close proximity to the data sources themselves, allowing for reduced latency and improved efficiency by bringing computing capabilities closer to the data sources.[15]

I.4.2 Architecture

The edge computing architecture is a federated network structure that extends cloud services to the edge of the network. It achieves this by introducing edge devices between terminal devices and cloud computing. The collaboration between the cloud and edge is structured into three layers:

- **Terminal Layer:** The terminal layer encompasses all devices connected to the edge network, including mobile terminals and various Internet of Things (IoT) devices such as sensors, smartphones, smart cars, cameras, etc. In this layer, devices serve as both data consumers and providers. The focus is on data perception rather than computing power, aiming to reduce service delays. Millions of devices in the terminal layer collect raw data, which is then uploaded to the upper layers for storage and processing.[16]

- **Boundary Layer:** The edge layer, also known as the boundary layer, is the core of the three-tier architecture. It is situated at the edge of the network and consists of widely distributed edge nodes positioned between terminal devices and the cloud. This layer typically includes base stations, access points, routers, switches, gateways, and other network infrastructure components. The edge layer supports the downward access of terminal devices, storing and processing the data uploaded by them. It connects with the cloud and uploads the processed data. Due to its proximity to the users, the edge layer is suitable for real-time data analysis and intelligent processing, offering efficiency and security advantages over cloud computing.[9]

- **Cloud Layer:** Within the federated services of cloud-edge computing, the cloud computing layer remains the most powerful data processing center. It comprises multiple high-performance servers and storage devices, boasting significant computing and storage capabilities. The cloud layer excels in areas requiring extensive data analysis, such as regular maintenance and business decision support. It serves as a permanent storage for data reported by the edge computing layer and handles analysis and processing tasks that the edge layer may not be capable of. Additionally, the cloud module can dynamically adjust the deployment strategy and algorithm of the edge computing layer based on control policies.[16]

I.4.3 Characteristics

- **Dense Geographical Distribution :** Edge computing achieves proximity between cloud services and users by deploying multiple computing platforms in edge networks. The dense geographical distribution of these infrastructure offers several advantages. Firstly, network administrators can provide location-based mobility services without the need to traverse the entire wide area network (WAN). This enables efficient and localized mobility services. Secondly, big data analytics can be performed rapidly and with enhanced accuracy due to the proximity of edge systems to the data sources. This reduces latency and enables real-time analysis of large datasets. Lastly, edge systems enable real-time analytics at scale, facilitating quick processing of data and enabling applications such as sensor networks for environmental monitoring and pipeline monitoring.[17]

- **Mobility Support:** With the increasing number of mobile devices, Edge computing provides support for mobility through protocols like the Locator ID Separation Protocol (LISP). LISP allows direct communication with mobile devices by separating the location identity from the host identity and implementing a distributed directory system. By decoupling the host identity from the location identity, Edge computing enables seamless mobility support. This key principle ensures that mobile devices can maintain connectivity and communicate effectively within the Edge computing environment, enhancing the overall user experience and enabling efficient utilization of mobile resources.[18]

- **Location Awareness:**

  The location-awareness feature of Edge computing enables mobile users to access services from the Edge server that is closest to their physical location. This capability is made possible through the use of technologies such as cell phone infrastructure, GPS, or wireless access points, which allow the determination of the location of electronic devices. This location awareness is beneficial for various Edge computing applications, including Fog-based vehicular safety applications and Edge-based disaster management. By leveraging the knowledge of a user's location, these applications can provide personalized and context-aware services, enhancing safety and efficiency in scenarios such as vehicular communication and emergency response management.[19]

- **Proximity:** Edge computing brings computation resources and services closer to users, resulting in an enhanced user experience. The proximity of these resources enables users to

utilize network context information to make informed decisions regarding offloading computation tasks and utilizing services. By leveraging local computational resources, users can optimize their service usage based on their specific needs and network conditions. Additionally, service providers can benefit from the mobile user's information, extracting device details and analyzing user behavior to enhance their services and allocate resources more efficiently. This data-driven approach enables service providers to deliver improved services, personalized experiences, and optimized resource allocation, ultimately enhancing the overall effectiveness of Edge computing.[17]

- **Low Latency:**

  Edge Computing paradigms reduce latency by bringing computation resources and services closer to users. This low latency allows users to run resource-intensive and delay-sensitive applications on Edge devices such as routers, access points, base stations, or dedicated servers. By leveraging these resource-rich Edge devices, users can experience faster and more responsive application execution, improving overall performance and user satisfaction.[18]

- **Context-Awareness** :

  Context-awareness is a key characteristic of mobile devices, closely related to location awareness. In Edge computing, the context information of mobile devices plays a crucial role in making offloading decisions and accessing Edge services. Real-time network information, including network load and user location, can be leveraged to provide context-aware services to Edge users. Service providers can utilize this context information to enhance user satisfaction and improve the quality of experience. By leveraging the context-awareness of mobile devices, Edge computing environments can deliver personalized and optimized services, enhancing overall user satisfaction and engagement.[17]

- **Heterogeneity:**

  Heterogeneity in Edge computing encompasses varied platforms, architectures, infrastructures, computing technologies, and communication technologies used by the Edge computing elements. End device heterogeneity arises from variations in software, hardware, and technology. Edge server heterogeneity stems from differences in APIs, custom-built policies, and platforms. These differences give rise to interoperability challenges, posing

a significant hurdle in the effective deployment of Edge computing. Network heterogeneity pertains to the diverse communication technologies that influence the delivery of Edge services. Managing and addressing these heterogeneities is crucial for achieving seamless integration and efficient operation in Edge computing environments.[17]

I.5 Fog Computing

I.5.1 Definition

Fog computing, initially introduced by Cisco in 2012, is a highly virtualized platform that facilitates storage, computation, and network services between smart devices and cloud servers. It is primarily deployed at the network edge, although not exclusively. Instead of replacing cloud computing, fog computing is considered an extension of it, creating a hierarchical infrastructure where local data analysis occurs at the fog layer and global analysis takes place in the cloud. Over time, the definition of fog computing has been refined to describe a scenario where numerous heterogeneous and distributed fog devices collaborate without relying on third-party involvement to carry out data storage and computation tasks. These tasks support essential network functions and enable the execution of new services and applications within sandboxed environments. A more general definition is given: fog computing is a distributed computing platform with a resource pool consisting of one or more ubiquitous and heterogeneous fog devices located at the network edge. This platform is not exclusively reliant on cloud servers and aims to provide flexible storage, computation, and network services to a large-scale user base.[20]

I.5.2 Architecture

The hierarchical architecture of fog computing is composed of the following three layers (as show in figure II.2:

- **The terminal layer**, also known as the edge layer, is the layer closest to the end user and the physical environment. It interacts directly with various IoT devices, such as sensors, mobile phones, smart vehicles, card readers, and more. While mobile phones and smart vehicles possess computing power, in this context, we consider them as smart sensing devices. These devices are typically distributed geographically and have the primary responsibility of sensing and measuring data related to physical objects or events. They

transmit the sensed data to the upper layers for processing and storage.[2]

- **Fog layer**

  The core layer, situated at the network's edge, is composed of multiple fog nodes, typically including routers, gateways, switches, access points, base stations, and dedicated fog servers. These fog nodes are widely distributed and positioned between the end devices and the cloud. They can be found in coherent places and geographic zones such as cafes, shopping centers, bus terminals, streets, parks, etc. Fog nodes can be either static, located at fixed positions, or mobile, mounted on moving carriers.

  End devices are connected to fog nodes to access services. Fog nodes possess the capability to compute, transmit, and temporarily store the sensed data received from the end devices. The fog layer enables real-time analysis and supports latency-sensitive applications. Additionally, fog nodes are connected to cloud services through the IP core network. They are responsible for facilitating interaction and collaboration with the cloud, leveraging its more powerful computing and storage capabilities. [21]

- **Cloud layer**

  This layer comprises multiple high-performance servers and storage devices, offering a range of application services such as smart home, smart transportation, smart factory, and more. It possesses robust computing and storage capabilities to support extensive computational analysis and permanent storage of vast amounts of data. However, unlike traditional cloud computing architectures, not all computing and storage tasks are routed through the cloud.

  To enhance the utilization of cloud resources, efficient management and scheduling of the cloud core modules are implemented through various control strategies. These strategies take into account the demand load and optimize the allocation of cloud resources. By dynamically managing and scheduling tasks, the cloud core modules ensure efficient utilization of available computing and storage resources within the cloud infrastructure.

  [22]

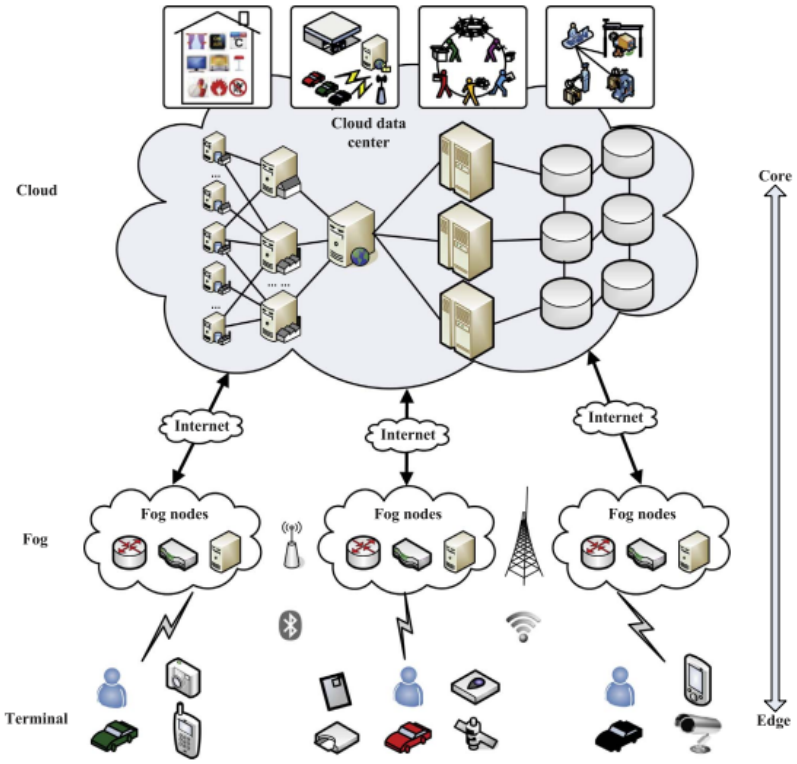

Figure I.2: The hierarchical architecture of fog computing [2]

I.5.3 Characteristics & Advantages

The architecture of Fog Computing is designed to provide numerous benefits and address the vulnerabilities associated with Cloud Computing. It offers several key characteristics that contribute to its advantages:

- **Distribution:** Fog Computing enables the distribution of computing resources across different nodes in the architecture. This decentralized approach provides increased flexibility and scalability, allowing for efficient processing and analysis of data closer to the edge devices [21].

- **Interoperability:** Fog Computing demonstrates high technical capability in facilitating interoperability among multiple systems, whether they are similar or vastly different. It allows different systems to communicate and operate together seamlessly without ambiguity or conflicts, enhancing overall system efficiency and integration [23].

- **Low latency:** Fog Computing significantly reduces latency by processing data close to the devices that collect it. This near-real-time data analysis is crucial in time-sensitive applications, such as preventing manufacturing line closures or restoring electrical services

promptly. The availability of computational resources within the Fog layer enables faster and less complex analyses, leading to quicker decision-making [21].

- **Redundancy:** Fog Computing incorporates redundancy features to ensure data integrity and availability. In case of connectivity loss, the latest data can be downloaded and synchronized with the Cloud when connectivity is restored. This redundancy allows for the reconstruction of data chronologically, even if previous data with timestamps are received after current data [24].

- **Security:** Fog Computing prioritizes data security for Internet of Things (IoT) applications. It supports various security measures such as cryptography, multi-level authentication, and authorization requests within the Fog nodes themselves. These security policies can be executed locally without relying on the Cloud, enhancing the overall security of the system [25].

I.5.4 Resource Management

Resource Management plays a crucial role in Fog Computing systems. It can be classified into six sub-dimensions: resource provisioning, application placement, scheduling, resource allocation, task offloading, and load balancing.

I.5.4.1 Resource provisioning

Resource provisioning involves allocating and providing resources based on user requests. There are three main provisioning models: user-centric, dynamic, and static. User-centric provisioning caters to user demands but can be costly and lead to resource underutilization. Dynamic provisioning adapts to fluctuating workloads through auto-scaling, but accurate predictions are needed to avoid over or under-provisioning. Static provisioning allocates resources for a fixed period but can result in wastage. Auto-scaling and policies like reactive, proactive, and hybrid enable dynamic provisioning. Reactive policy responds in real-time, proactive policy estimates future demands, and hybrid policy combines both. Proactive policy relies on predictive approaches, such as ML-based techniques like NN, RL, and DL, to improve resource provisioning accuracy for future user demands.[26]

I.5.4.2 Application placement

Application placement refers to the optimal distribution of applications across different layers of FC (Fog/Edge/Cloud) nodes. Its goal is to achieve improved Quality of Experience (QoE) and Quality of Service (QoS) by efficiently mapping application modules to available resources. Key components of application placement include resource types (VMs, containers, bare metals), placement strategies (static and dynamic), orientations, mapping techniques, placement controller, and placement metrics. Static placement involves placing applications once and keeping them running, while dynamic placement adapts to fluctuating workloads by adding, terminating, or migrating application instances. The distributed architecture of FC nodes cannot ignore node orientations, as they impact QoS factors like communication delay. Mapping policies based on priority, optimization, or multi-objective trade-off can be adopted, with priority assigning applications to specific nodes, optimization aiming to minimize or maximize certain metrics, and multi-objective trade-off considering multiple metrics. Placement controllers manage placement activities and can be centralized or decentralized, with centralized assigning management responsibility to a common node and decentralized allowing nodes or brokers to manage it. Ultimately, the objective of application placement is to optimize QoE and QoS.[27]

I.5.4.3 Scheduling

The scheduling problem in FC (Fog/Edge/Cloud) systems can be divided into two categories: resource scheduling and task scheduling. Although the terms resource and task scheduling are sometimes used interchangeably, resource scheduling focuses on finding the best resources for client requests to achieve scheduling goals such as optimal resource utilization. On the other hand, task scheduling involves assigning a set of tasks to available resources considering the QoS requirements of the tasks. Resource scheduling primarily serves the goals of service providers, while task scheduling aims to enhance the end-user experience.

Scheduling algorithms can be classified as static, dynamic, or hybrid. Static algorithms require prior information about the number of tasks and available resources, which may not always be feasible or practical. In contrast, dynamic or hybrid approaches are more suitable as they can react accurately and in real-time to schedule incoming tasks without relying on predetermined information. These dynamic and hybrid algorithms can adapt to changing conditions and optimize scheduling decisions based on real-time data and requirements.[27]

I.5.4.4 Resource Allocation

Resource allocation in FC (Fog/Edge/Cloud) networks involves assigning tasks with varying QoS requirements to heterogeneous fog nodes, aiming to improve response time, resource utilization, energy consumption, and cost. With the dynamic and dense nature of fog networks, resource allocation becomes a challenging NP-hard problem. Factors like application heterogeneity, stochastic workload, and mobility must be considered for efficient resource allocation. Applications can be real-time or delay tolerant, and their task arrival rates and durations are stochastic, leading to changing resource requirements. Mobility introduces constraints related to speed, time, and distance, which can impact user experience. Various methods have been proposed for resource allocation, including auction-based and optimization-based approaches, with sub-categories such as game theory, heuristics, meta-heuristics, fuzzy logic, and ML-based solutions.[28]

I.5.4.5 Task offloading

Offloading in a computing environment involves transferring resource-intensive tasks from devices with limited resources to devices with ample resources. The decision of what to offload can involve data, computation, or the entire application. Determining where to offload tasks depends on the offloading policy, which can include options such as no offloading, horizontal offloading (device-to-device or fog-to-fog), or vertical offloading (device-to-fog or fog-to-cloud). To address the challenges associated with offloading, various techniques have been proposed in the literature, including AI, approximation methods, mathematical modeling, heuristics, meta-heuristics, game theoretic approaches, and fuzzy-based approaches. These techniques aim to optimize offloading decisions and improve overall system performance.[27]

I.5.4.6 Load Balancing

Load balancing (LB) mechanisms play a crucial role in reducing response time, energy consumption, and improving throughput in FC systems. LB architectures can be classified into centralized, decentralized, and semi-decentralized approaches. In the centralized approach, a central node acts as the global LB node, while the decentralized approach involves multiple LB nodes, typically used for network clusters. The semi-decentralized approach employs both global and local LB nodes. However, the effectiveness of LB mechanisms in FC depends not only on the architectural perspective but also on LB strategies (static, dynamic, hybrid), algorithms, and

performance objectives. In static LB, the workload is evenly distributed among fog nodes without considering their capabilities. Dynamic LB continuously shares resource information and updates the LB knowledge base accordingly, leading to improved response time and throughput but higher bandwidth consumption. Hybrid LB combines aspects of both static and dynamic strategies to achieve a balance between bandwidth consumption and performance objectives.[28]

I.6 Conclusion

In conclusion, this chapter provides a comprehensive overview of the fundamental concepts and technologies related to IoT, cloud computing, edge computing, and fog computing. The chapter explains the characteristics, architectures, benefits, and limitations of each of these concepts. IoT is a rapidly growing field that involves the interconnection of a vast number of devices and systems. Cloud computing enables the storage and processing of large amounts of data generated by IoT devices, but it can also lead to latency, bandwidth, and security issues. Edge computing and fog computing are distributed computing paradigms that bring computing resources closer to the source of data, enabling faster processing and analysis. The chapter also discusses key resource management techniques used in fog computing, including resource provisioning, application placement, scheduling, resource allocation, task offloading, and load balancing. The next chapter goes deeper into the key technologies and concepts involved in the distributed architecture of fog computing.

CHAPTER II

DISTRIBUTED ARCHITECTURE FOR FOG COMPUTING

II.1 Introduction

In this chapter the key technologies and concepts involved in the distributed architecture of fog computing are discussed in depth. This includes virtualization, containerization, Docker, container orchestration, and resource scaling. The chapter also explores the differences between virtual machines and containers and highlights the advantages of containerization. Additionally, the chapter introduces the concept of microservices, which is an architectural style that structures an application as a collection of loosely coupled services, each of which can be developed, deployed, and scaled independently. Overall, this chapter provides a comprehensive overview of the technologies and concepts involved in the development of fog computing applications.

II.2 Virtualization

II.2.1 Virtual Machine

Virtualization offers numerous advantages, including flexible allocation of physical resources to virtualized applications. This flexibility allows for dynamic adjustments in the mapping of virtual to physical resources and the allocation of resources to each application, effectively adapting to changing workloads. Additionally, virtualization enables multi-tenancy, allowing multiple instances of virtualized applications, referred to as ”tenants,” to share a single physical server. By consolidating and packing applications into a smaller set of servers, data centers can reduce operating costs. Another benefit of virtualization is simplified replication and scaling of applications. In data center environments, there are two common types of server virtualization technologies: hardware-level virtualization and operating system level virtualization.[29]

A virtual machine (VM) is a software or hardware-based emulation of a physical computer. VMs contain applications, libraries, dependencies, and a complete operating system with its own memory management. Multiple VM instances, running the same or different operating systems, can operate simultaneously on a single host. Hypervisors manage the VMs and allocate dedicated resources such as CPU, memory, and system resources to each VM. This allows for efficient utilization of hardware and the ability to run diverse operating systems on a single machine.[30]

II.2.2 Hardware virtualization

Hardware virtualization entails the virtualization of server hardware, where virtual machines

are created to emulate the functionality of physical machines. This is achieved by running a hypervisor, or virtual machine monitor (VMM), either directly on the bare metal server (Type-I Hypervisor) or on top of the existing server operating system (Type-II Hypervisor). The hypervisor simulates virtual hardware components such as the CPU, memory, I/O, and network devices for each virtual machine. Each virtual machine operates independently, running its own operating system and applications. The hypervisor is responsible for efficiently distributing the underlying physical resources among the virtual machines.[29]

Modern hypervisors offer various strategies for resource allocation and sharing of physical resources within virtual machines (VMs). These strategies include strict partitioning, where each VM is allocated dedicated physical resources, as well as best-effort sharing, where resources are shared among VMs based on their dynamic needs. Additionally, the hypervisor ensures isolation among VMs by intercepting privileged hardware access requests from guest operating systems and handling them within the hypervisor itself. Notable examples of hardware virtualization platforms are VMware ESXi, Linux KVM, and VirtualBox.[31]

II.2.3 Operating system virtualization

Operating system virtualization focuses on virtualizing the OS kernel instead of the physical hardware. This approach utilizes containers as OS-level virtual machines, where each container encapsulates a set of processes that are isolated from other containers or system processes. The container abstraction is implemented by the OS kernel, which handles the allocation of CPU shares, memory, and network I/O to each container. Additionally, the kernel can provide file system isolation for the containers. This form of virtualization allows for efficient resource utilization and isolation at the OS level.[29]

Similar to hardware virtualization, operating system virtualization through containers supports various allocation strategies such as dedicated, shared, and best effort. Containers offer lightweight virtualization as they do not run their own OS kernels but instead leverage the underlying kernel for OS services. In certain scenarios, the underlying OS kernel may emulate a different version of the OS kernel for processes within a container. This feature is commonly utilized to facilitate backward OS compatibility or emulate different OS application programming interfaces (APIs). [32]

II.3 Microservices

Microservices are small, independent services used to build complex systems. Unlike traditional architectures, microservice architecture relies on loosely coupled and modular services. The key benefits of microservices include easy development, deployment, and scalability due to their independent nature.[33]

Microservices contribute to the robustness of systems by ensuring that failures in one component do not affect the usability of others. If a service instance fails, other instances can take over its responsibilities. Additionally, microservices allow for efficient scaling as individual services can be scaled based on demand, optimizing resource utilization and minimizing bottlenecks.[34]

To ensure performance isolation, microservices should be run in isolated environments. Virtualization is one approach, but the overhead can be high for small microservices. Containers, such as Docker, provide a more lightweight and cost-efficient solution, offering isolated execution environments for microservices.[35]

microservices provide independence, robustness, and scalability to complex systems. They can be efficiently managed through containerization, enabling isolated execution environments and reducing overhead.[34]

II.4 Containerization

Containerization technology, as a lightweight alternative to virtualization, focuses on abstracting the operating system (OS) level instead of virtualizing the hardware stack with virtual machines. Containers allow applications to run inside isolated environments, providing efficiency in interconnection between containers and portability. They have a lower start-up time compared to virtual machines, as they can share resources with the host machines, leading to enhanced resource utilization.[36]

Containers offer independent provisioning of spatial isolation, processes, file systems, namespaces, and hardware resources for each running instance. Their ability to facilitate fast application deployment has brought a shift in computing operations, especially in scientific programming. Containers have demonstrated near-native performance when compared to bare-metal execution of CPU-intensive applications, making them increasingly popular in high-performance computing applications.[37]

container paradigms in computing enable rapid application deployment, achieve efficient re-

source utilization, and provide isolation for running instances. Their lightweight nature and performance benefits have made them highly valuable in scientific and high-performance computing domains.[38]

There are several important container tools available in the realm of containerization. Containerization tools are software platforms that simplify the creation, deployment, and management of containers. These tools play a critical role in implementing containerization technology by providing features and functionalities for containerization workflows. Some popular containerization tools include:

- **Docker** - Open-source containerization platform .[39]
- **Microsoft Azure Container Registry** - Managed OCI distribution [40]
- **Vagrant** - CL utility for VMs lifecycle management.[41]
- **Buildah** - Command-line tool for OCI images and containers[42]
- **Podman** - Open-source, daemonless container engine for Linux[43]
- **Containerd** - A simple yet robust container runtime[44]
- **LXD** - Linux daemon by Ubuntu [45]
- **ZeroVM** - Open-source virtualization with sandbox support[46]

II.4.1 Docker

Docker is a comprehensive platform that facilitates the construction, deployment, and execution of distributed applications. By providing a framework for complete application runtime encapsulated as an image, Docker liberates applications from dependency on specific infrastructure. This independence empowers developers and IT operations teams to fully leverage their capabilities, fostering a collaborative and innovative environment. With Docker, the need for traditional installations is eliminated, as the portable runtime system is readily available within the image itself.[47]

II.4.2 Docker Image

A Docker Image is a self-contained and unchangeable representation of a container, created by combining a series of layers. These layers are specified in the Dockerfile, with each layer

capturing only the modifications from the previous layer. Docker Images are versatile and can be utilized at every stage of the deployment process.[48]

II.4.3 Docker Containers

Docker Containers are generated from Docker images, serving as self-contained units that encapsulate all the necessary components required for running an application in an isolated manner. To ensure a confined environment, the container includes all the essential dependencies and resources specific to the application. In the case of an application comprising an Ubuntu OS and Nginx server, a Docker image is constructed by incorporating these components. By executing the ”docker run” command, a container is instantiated from the Ubuntu OS image, which includes the Nginx server, allowing it to commence running. [49]

II.4.4 Docker Machine

Docker Machine is a convenient tool that allows users to easily provision and manage multiple Docker hosts remotely from a personal computer. These Dockerized hosts serve as servers to run Docker containers, enabling the deployment and execution of containerized applications.[50]

Docker Machine supports a wide range of backend cloud service providers, including popular ones like Amazon Web Services, Microsoft Azure, Digital Ocean, Google Compute Engine, and OpenStack, as well as others such as Exoscale, Generic, Rackspace, IBM Softlayer, and VMware vCloud Air. This broad compatibility makes Docker Machine a versatile choice for centrally managing macroservices across different locations and vendors.[50]

By combining Docker Machine with Docker Swarm, users can create a virtual system of systems spanning multiple clouds. This allows for the orchestration and coordination of containerized applications across various cloud environments, providing a scalable and flexible infrastructure for distributed systems.

The combination of Docker Machine and Docker Swarm offers a powerful solution for managing containerized workloads in a distributed and cloud-agnostic manner. It simplifies the management and deployment of containers across diverse cloud platforms, enabling users to build and scale complex systems effectively.

II.4.5 Dockerfile

A Dockerfile is used to define the environment inside a container. It specifies the operating system, networking interfaces, disk drives, port mappings, required software, folder permissions, files to be copied, and the execution of specific scripts. By creating an image from the Dockerfile, it ensures that the resulting builds from this image will be identical regardless of where they are run. This guarantees consistency and reproducibility, allowing containers to be deployed and executed reliably across different environments.[51]

II.4.6 Docker Advantages

- Portability: Docker enables applications to run on any system or server that has Docker installed. The containerized application remains consistent and can be deployed across different environments without compatibility issues.[52]

- Composability: Docker simplifies the composition of complex web applications that comprise multiple software components. These components, such as servers and databases, can be combined into a single container or distributed across multiple containers. Each container can be independently managed, facilitating modular development and deployment.[53]

- Isolation: Docker ensures that each application operates in isolation within its own container. This isolation provides security and prevents conflicts between different applications running on the same system. Additionally, Docker containers are isolated from the underlying system, reducing dependencies and ensuring consistent behavior.[53]

- Orchestration and Scaling: Docker containers are lightweight and easily scalable. Multiple containers can be created on a single server or scaled across a cluster of servers effortlessly. Docker's orchestration tools, like Docker Swarm or Kubernetes, simplify the management and scaling of containerized applications, allowing for efficient resource utilization and improved performance.[53]

II.5 Virtual Machines vs Containers

Containers are composed of the necessary libraries and dependencies required to run an application's logic. Unlike virtual machines (VMs), containers do not have isolated operating systems

but share the underlying host operating system among multiple containers. This distinction between VMs and containers makes containers more lightweight, cost-effective, and scalable.[29] Compared to VMs, containers require fewer resources. They can be created from a container image within seconds and are well-suited for running microservice applications in a distributed architecture.[3]

Containers and VMs are not mutually exclusive but can coexist and complement each other. By running containers within VMs, you can harness the benefits of containerization while leveraging the isolation and management capabilities provided by virtualization.[3]

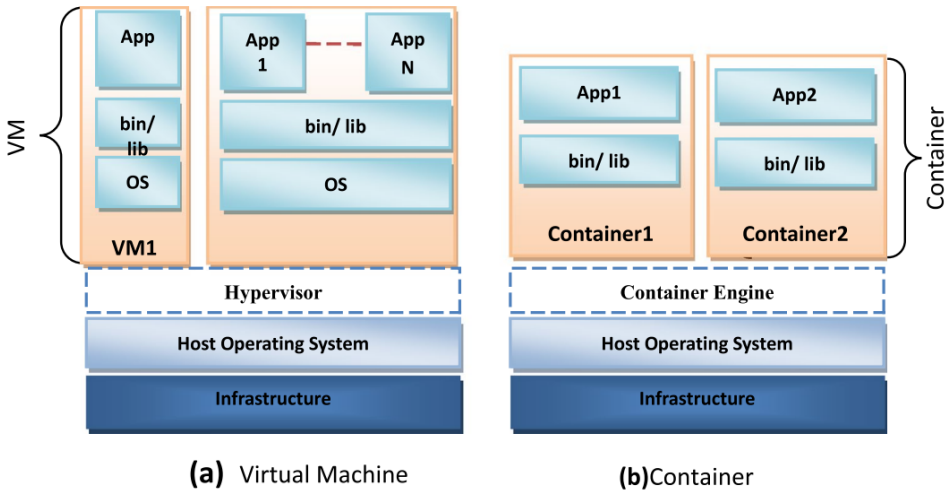

Figure II.1: Architecture comparison virtual machine v.s. container [3]

II.6 Container orchestration

Container orchestration is the process of automating the deployment, scaling, and management of containerized applications. It involves using specialized software tools, called container orchestration systems or platforms, to manage the complex infrastructure required to run containerized applications.[54]

Container orchestration systems provide a wide range of features for managing containers at scale, such as container deployment, load balancing, scaling, service discovery, and networking. These features enable developers to deploy and manage applications in a way that is efficient, resilient, and scalable. Moreover, container orchestration systems allow organizations to respond quickly to changes in demand or resource availability, thus enabling them to optimize resource utilization and reduce operational costs.[55]

In fog computing, container orchestration solutions are used to manage container life cycle, scale them up or down, perform self-healing, migrate them, and manage a heterogeneous infrastructure. Containerization provides a lightweight and small footprint form of virtualization, making it more appropriate for resource-restricted devices in a fog computing environment compared to traditional virtual machines. Container orchestration solutions such as Kubernetes, Docker Swarm,and Marathon on Mesos, enable efficient deployment, scaling, and management of containerized applications across multiple fog nodes or hosts, while ensuring high availability and scalability of the applications. These solutions abstract the underlying physical devices where the containers are executed, enabling fog computing environments to implement and automate the essential characteristics of Fog Computing such as heterogeneity, geographic distribution, and scalability.[56]

II.6.1 Kubernetes

Kubernetes is an open-source platform created by Google in 2014 that simplifies the management of containerized applications across a cluster of machines. The platform is based on a master/slave architecture where a developer submits a list of applications to a master node, and the platform deploys them across slave and master nodes.[57]

The master node serves as the control plane of the cluster, and it can be replicated to ensure high availability and fault tolerance by utilizing the scheduling layer. The slave nodes, also known as minions, are where application containers are executed. Kubernetes provides a containerized application as a set of containers, each specific for a single microservice.[57]

A pod is a fundamental unit in Kubernetes that represents a group of containers that are co-scheduled. All containers within a pod are controlled as a single application, and thus, they share the same environment. A pod may have one or more containers, and because they run in a shared context, containers within the pod can be scaled as a unique application.[57]

The Replication Controller is a Kubernetes component responsible for managing pod replication to ensure that a certain number of pods are currently providing a specific service. If the current state departs from expectations, such as in the case of a node outage, the Replication Controller automatically starts scheduling a new instance on a different slave node. The heartbeat controller mechanism is designed with a subscriber notification system and is not overly aggressive.[57]

II.6.2 Apache Mesos

Apache Mesos is a significant open-source project initially developed by the University of California, Berkeley. It adopts a master/slave design pattern, where slave nodes execute tasks delegated to them by the master process. The master process runs on a manager node in the cluster and is responsible for managing and monitoring the entire cluster architecture. It communicates with frameworks that aim to schedule jobs on slave nodes.To provide orchestration functionalities to the entire Mesos cluster, Mesos solutions typically include installing an application-level management tool called Marathon on top of the Mesos cluster. Marathon interacts with the master component, and in case of slave faults, it starts a new instance to ensure fault-tolerance. Mesos ensures high availability by replicating master nodes and providing failover mechanisms in case of master failures. To achieve this, it relies on Apache Zookeeper, which employs an election algorithm to select a new node to play the master role.[56]

II.6.3 Docker Swarm

One of the most widely used container orchestration engines is Docker Swarm, which was first released in 2016 under the Docker brand. As its name suggests, Docker Swarm is designed to work seamlessly with the Docker API, offering a straightforward way to manage and configure containers within the Docker environment.[57]

With Docker Swarm, users can manage individual containers or large clusters of containers distributed across a network. The engine facilitates container cooperation, which is necessary since Docker containers are intended for standalone operation and cannot work together without assistance.[4]

Docker Swarm offers advanced planning features that enable users to select the most suitable nodes within a container cluster for deploying specific containers. Developed in Go, Docker Swarm is tightly integrated with the Docker API, enabling users to leverage all of the powerful features available in the Docker environment.[4]

Using YAML serialization language, container deployment is simplified, and users familiar with the Docker ecosystem can quickly adapt to managing their container infrastructure with Docker Swarm, without the need to learn new orchestration concepts from other vendors.[4]

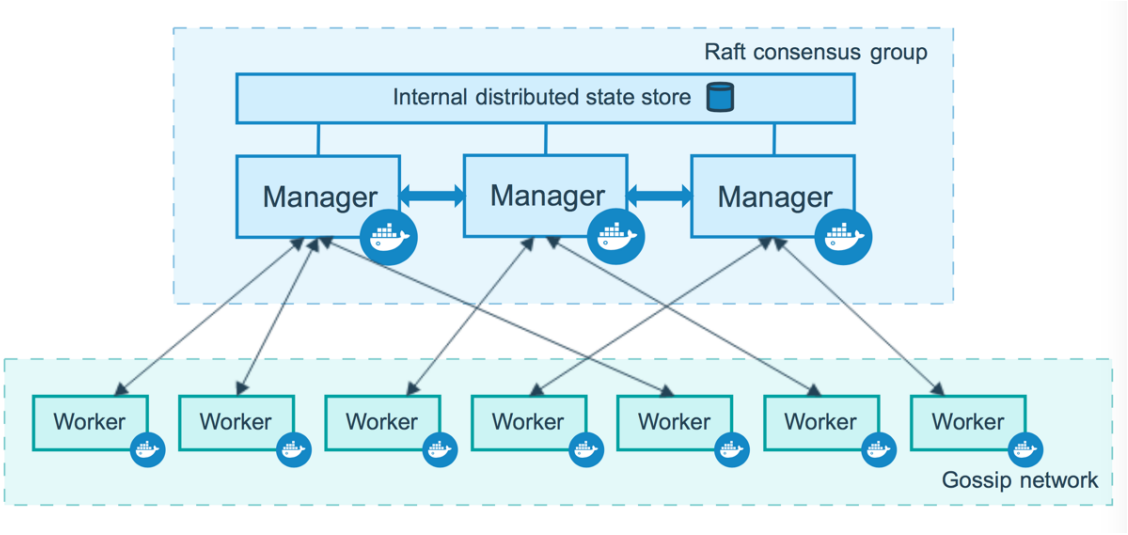

Figure II.2: Architecture Docker Swarm [4]

II.6.3.1 nodes

A node is an instance of the Docker engine that participates in the swarm. Nodes can be thought of as Docker hosts or machines that run the Docker engine and join a cluster managed by the swarm.[4]

Nodes can run on a single physical computer or cloud server, but in production swarm deployments, nodes are typically distributed across multiple physical and cloud machines. Nodes can be designated as either manager or worker nodes.[58]

Manager nodes are responsible for managing the swarm state and orchestration, while worker nodes receive and execute tasks assigned by manager nodes. By default, manager nodes also act as worker nodes, but this can be configured to have separate sets of manager-only and worker-only nodes.[58]

Each node in the swarm has a unique identifier and can communicate with other nodes through a secure gossip protocol. Nodes can also be added or removed from the swarm as needed, allowing for dynamic scaling and high availability.[4]

II.6.3.2 service and tasks

In Docker Swarm, a service is a definition of the tasks to be executed on the manager or worker nodes. It's the central structure of the swarm system and the primary point of interaction with the user. When a service is created, the user specifies which container image to use and which commands to execute inside running containers.[4]

There are two types of services in Docker Swarm: replicated and global. In the replicated services model, the swarm manager distributes a specific number of replica tasks among the nodes based on the desired state. This means that the manager node assigns tasks to worker nodes according to the number of replicas set in the service scale. Each task carries a Docker container and the commands to run inside the container. The task is the atomic scheduling unit of the swarm.[4]

In contrast, global services run one task for the service on every available node in the cluster. This means that the swarm runs a single task for the service on each worker node in the swarm.[4]

Once a task is assigned to a node, it cannot be moved to another node. The task can only run on the assigned node or fail. The manager node monitors the tasks and takes appropriate action in case of any failure or unavailability of a node.[4]

II.6.3.3 Load balancing

In swarm mode, the swarm manager utilizes ingress load balancing to expose services externally within the swarm. The service can be automatically assigned a PublishedPort by the swarm manager, or a specific PublishedPort can be configured. If no port is specified, the swarm manager assigns a port within the range of 30000-32767 to the service. External components, including cloud load balancers, can access the service through the PublishedPort on any node in the cluster, regardless of whether the node is currently running the task for the service. All nodes in the swarm handle ingress connections and route them to a running task instance. Swarm mode also incorporates an internal DNS component that assigns a DNS entry to each service in the swarm. The swarm manager utilizes internal load balancing based on the DNS name of the service to distribute requests among services within the cluster.[58]

II.7 Resource scaling

Resource scaling in fog computing refers to the dynamic allocation of resources based on the varying demands of the system. It is a crucial aspect of achieving scalability and efficient resource utilization in fog computing environments. Managing different requests and adapting resource allocation accordingly is essential for achieving scalability in fog computing.There are two key concepts in scaling resources: vertical scaling and horizontal scaling. [59]

II.7.1 Vertical Scaling

In vertical scaling, or scaling up/down, involves modifying the allocated resources of a running VM. This can include increasing or reducing the CPU power, memory, or other resources assigned to the VM. However, many common operating systems do not support making such changes without requiring a reboot, even if the VM is involved. As a result, most cloud providers primarily focus on offering horizontal scaling capabilities. [59]

In the case of containers, vertical scaling might involve adding more CPU cores, increasing memory allocation, or expanding storage capacity for a single container instance. Vertical scaling is useful when an application or container requires more resources to handle increased workload demands. However, it has limitations as there is a maximum limit to the available resources on a single instance.

II.7.2 Horizontal Scaling

In horizontal scaling, also known as scaling out/in, the resource unit being adjusted is the server replica, which runs on a virtual machine (VM). Additional replicas are added or released as needed to handle the workload. This approach allows for dynamically increasing or decreasing the number of server replicas based on demand.[59]

Horizontal scaling is more prevalent and commonly offered by cloud providers because it provides greater flexibility in dynamically adjusting resources based on demand. It allows for the addition or removal of server replicas without interrupting the overall system. Vertical scaling, although possible, may involve downtime or disruption due to the need for restarting or reconfiguring the VM or underlying hardware.[59]

In the case of containers, horizontal scaling would mean running multiple container instances of the same service or application, usually behind a load balancer, to distribute incoming traffic across the instances. Horizontal scaling provides better scalability as it allows for adding more instances to handle increased demand and ensures better fault tolerance and availability.

II.8 Related Work

The autoscaling problem has received significant attention in the domains of cloud computing and IoT. However, the techniques employed to address this problem are not directly applicable to Fog computing environments due to their distinct characteristics. Consequently, numerous platforms, emulators, and frameworks have been developed thus far to tackle the autoscaling challenge specifically in Fog computing environments. In the following section, we will provide a summary of the relevant works that closely pertain to the autoscaling problem within Fog computing environments.

In their study, Yu et al. [60] introduced a framework designed to facilitate fog-enabled data processing in IoT systems, aimed at catering to real-time data applications that require low-latency services. The framework incorporates a preprocessing mechanism that operates within the fog computing platform. This preprocessing step efficiently handles the sensory data received from sensors, prior to forwarding it to the cloud. As a result, this approach significantly reduces the volume of data that needs to be transmitted onwards.

In the high-scalability system described in reference [61] for OpenStack, a master node is utilized to manage VMs running a specific application while also acting as the sole entry point for all application requests. Requests are queued and dispatched by the master node to the VMs. To ensure stability and load balancing, an automatic scaling algorithm is employed. However, a limitation arises as the master node becomes a performance bottleneck. To overcome this, a mechanism is required to separate the auto-scaling controller from the traffic entry points, allowing for improved scalability, performance, and avoidance of master node bottleneck issues.

Chang et al. [62] presented a monitoring approach for the Kubernetes container platform. Their approach involved a monitoring mechanism that offered comprehensive insights into system resource utilization ratios and application quality-of-service (QoS) metrics. This detailed information facilitated the implementation of a sophisticated resource provisioning strategy, enabling dynamic resource dispatch. However, it is worth noting that the system model utilized a single machine and lacked an explanation regarding the parameter settings employed in their research.

II.9 Conclusion

In conclusion, virtualization and containerization technologies provide powerful tools for building, deploying, and managing modern applications. Virtual machines provide strong isolation and security but can be resource-intensive, while containerization platforms like Docker offer better performance and resource utilization. Container orchestration tools like Kubernetes, Apache Mesos, and Docker Swarm allow you to manage and automate container deployment, scaling, and networking. Resource scaling can be done through vertical scaling or horizontal scaling, depending on the application's needs. Microservices architecture offers benefits such as improved scalability and flexibility, but also introduces some challenges, such as service discovery and data consistency. Overall, these technologies offer a powerful toolkit for building and managing modern, scalable applications.

CHAPTER III

AUTOSCALING ALGORITHM FOR CONTAINERS SYSTEM

III.1 Introduction

In This chapter will be addressed the design and propose a autoscaling algorithm for IoT container systems. The chapter begins with a discussion of the general architecture of the system and the internal architecture of the cluster with an overview of the model to understand its underlying concepts. The proposed autoscaling algorithm is then presented, which is responsible for dynamically scaling the number of container instances based on the current workload. The algorithm is designed to ensure that the system can efficiently handle varying workloads .

III.2 General Architecture

In the fog computing architecture based on container orchestration, the system consists of three main layer ,As shown in the figure III.1:

the edge layer, the fog layer, and the cloud layer. The edge layer represents the devices at the edge of the network, such as sensors, actuators, and gateways. These devices interact directly with the physical environment and generate data.

The fog layer resides between the edge and cloud layers. It comprises fog nodes, which are clusters of container swarms. These fog nodes are managed by container orchestration technologies, and are responsible for processing and analyzing data at the network edge. By leveraging containerization and orchestration, the fog layer enables efficient management and deployment of containerized applications and services.

The load balancer distributes incoming network traffic among multiple fog nodes. It prevents any single fog node from becoming overwhelmed with traffic, thus ensuring a balanced workload distribution.

The cloud layer represents the traditional cloud infrastructure, which consists of data centers and remote servers. It provides extensive computational and storage resources and can be used for more resource-intensive processing tasks or long-term data storage.

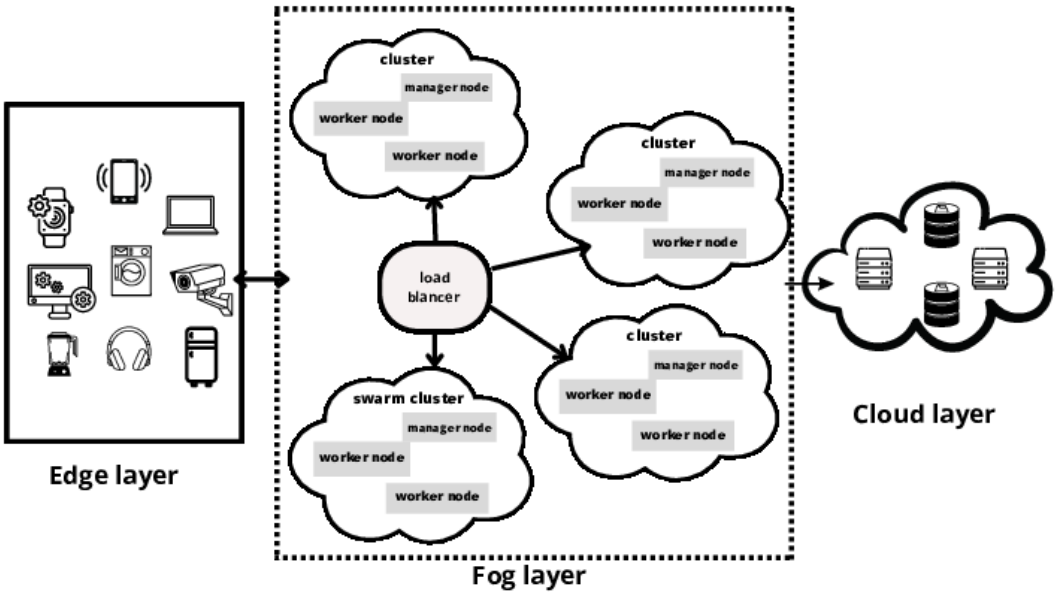

Figure III.1: The general architecture of fog computing based on container orchestration (personal)

III.3 Cluster Internal architecture

In cluster internal architecture , the Manager Node is responsible for managing and coordinating the worker nodes. The worker nodes are where the services run. The system includes two main modules as show in figure III.2: Monitoring and Autoscaler.

III.3.1 Monitoring /Autoscaler module

Is responsible for scraping and monitoring CPU metrics for the services. It collects CPU usage data from the worker nodes. The Monitoring module periodically collects CPU metrics from the worker nodes to monitor the resource usage and performance of the services. It tracks CPU utilization, load average, and individual process CPU usage to gather insights into the CPU resource utilization of the services running on the worker nodes.

The Autoscaler module is responsible for scaling the services in the Swarm cluster. It periodically fetches the CPU metrics from the Monitoring module to make scaling decisions. The Autoscaler module uses the collected CPU metrics as input to determine whether to scale up or scale down the services based on predefined scaling rules or policies.

if the Autoscaler module detects high CPU usage above a certain threshold, it may trigger a scaling action to add more instances or replicas of the service to handle the increased workload. On the other hand, if the CPU usage is consistently low, the Autoscaler module may scale down the services by reducing the number of instances or replicas to optimize resource utilization.

By leveraging the CPU metrics obtained from the Monitoring module, the Autoscaler module dynamically adjusts the number of service instances or replicas to maintain the desired performance, resource efficiency, and scalability of the services in the Swarm cluster.

the Monitoring and Autoscaler modules work together to ensure efficient resource management, effective scaling, and optimal performance of the services running in the Swarm cluster based on the collected CPU metrics.

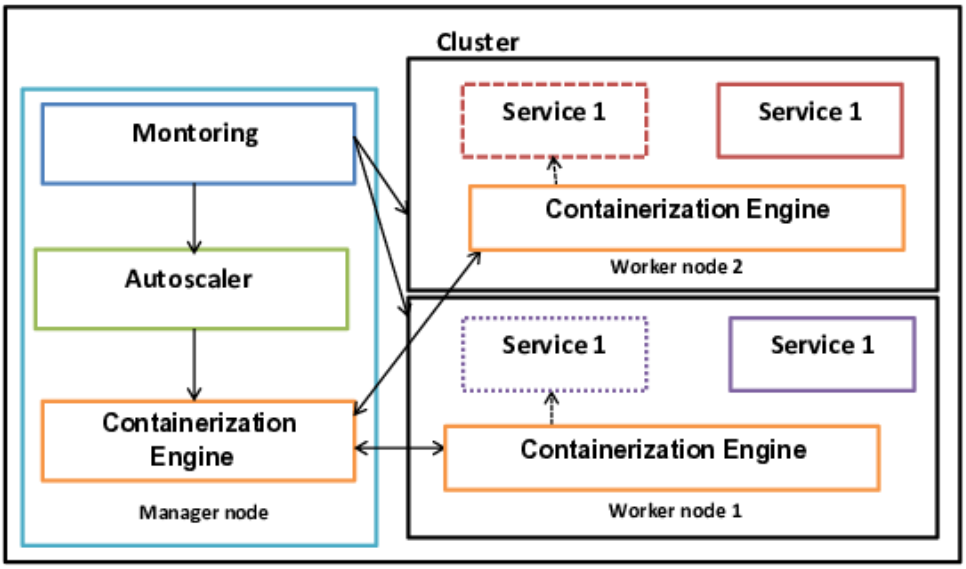

Figure III.2: Autoscaling in internal cluster using a Autoscaler(personal)

III.3.2 Illustrative Scenario

In the depicted scenario (figure III.3), three nodes are present: a management node and two worker nodes. One of the worker nodes hosts Service 1, which consists of three replicas, exhibiting a CPU usage of 84%. Meanwhile, the other worker node accommodates a Service 2, with three replicas and a CPU usage of 12%.

In the figure III.4, the CPU usage undergoes a change, reducing to 54% through an automated up scaling process. This adjustment involves the addition of a new service , which effectively distributes the load and ensures that the CPU usage remains within the defined range.

in the other hand a replica is removed from the second worker node, resulting in a CPU usage of 24 %.through an automated down scaling process to align the CPU usage within the specified minimum limit of 20% while optimizing resource utilization.

These images effectively highlight the dynamic nature of the system, where automatic scaling mechanisms respond to CPU usage thresholds. By dynamically adjusting the number of replicas and distributing the workload, the system efficiently utilizes resources while adhering to the defined performance limits.

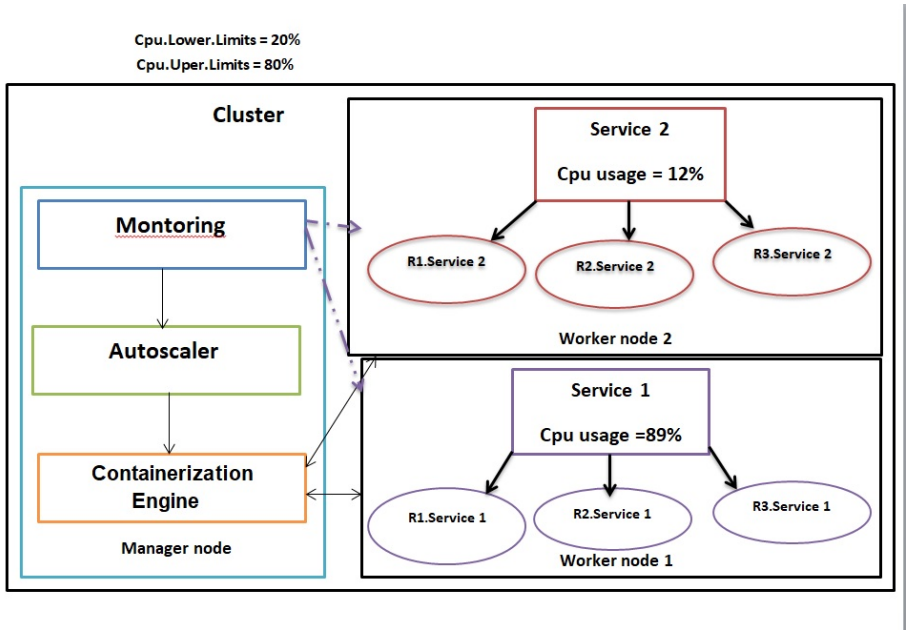

Figure III.3: Cluster before autoscaling (personal)

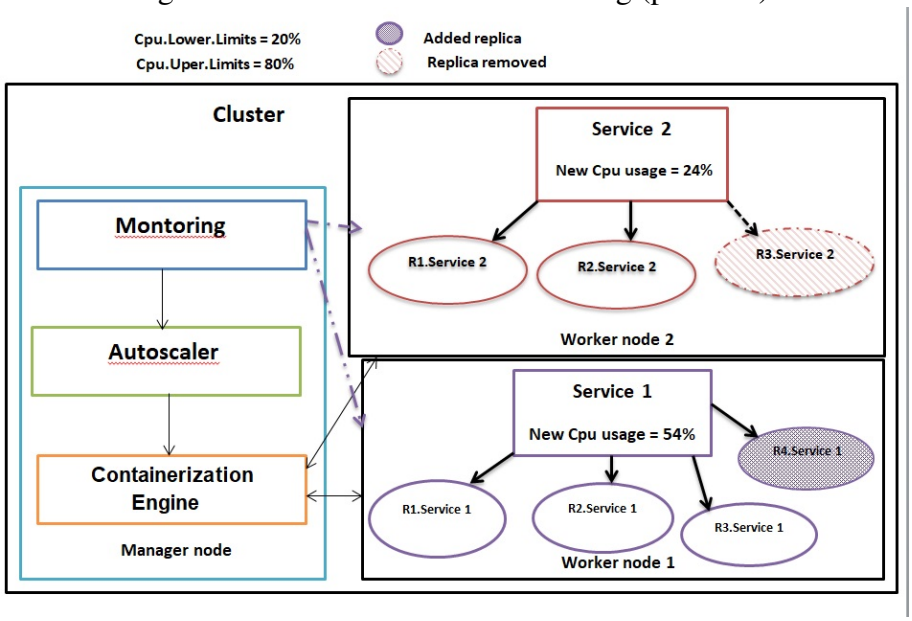

Figure III.4: Cluster after autoscaling (personal)

III.3.3 System Modules Interactions

In Figure III.5 the sequence diagram depicts the interactions between various components of a system during the autoscaling process. The key components involved in the sequence are the Autoscaler, Monitoring component, Manager Node, and Worker Nodes. Here's a breakdown of the interactions shown in the diagram:

1. The Autoscaler initiates the autoscaling process by sending a query to the Monitoring component (MM) for nodes CPU usage .
2. The Monitoring component (MM)receives the query and request services CPU usage from the worker nodes (WN) where the services are running.
3. The Monitoring component (MM)receives the services CPU usage from the worker nodes (WN) .
4. The Monitoring component (MM) responds to the Autoscaler (MA) with the nodes CPU usage data.
5. Upon receiving the CPU usage data, the Autoscaler (MA) processes the data along with service information to determine scaling actions.
6. If the CPU usage exceeds a predefined upper limit (CPU UPPER LIMIT), the Autoscaler sends a scale-up request to the Manager Node (MN) .
   (a) The Manager Node (MN) receives the scale-up request from the Autoscaler(MA) .
   (b) The Manager Node (MN) forwards the request for replicas augmntation to the appropriate Worker Nodes (WN) responsible for handling the respective services.
   (c) The Worker Nodes(WN) handle the scale-up request and create additional replicas of the service as necessary.
7. Conversely, if the CPU usage falls below a predefined lower limit (CPU LOWER LIMIT), the Autoscaler(MA) sends a scale-down request to the Manager Node (MN).
   (a) The Manager Node (MN)receives the scale-down request from the Autoscaler (MA) .
   (b) The Manager Node (MN) forwards the request minimization to the relevant Worker Nodes (WN) .

(c) The Worker Nodes (WN) handle the scale-down request and remove unnecessary replicas of the service based on the scaling action.

The sequence diagram illustrates the flow of interactions between the Autoscaler, Monitoring component, Manager Node, and Worker Nodes during the autoscaling process. It shows how the Autoscaler queries for CPU metrics, processes the received data, and triggers scaling actions based on predefined upper and lower limits, with the Manager Node coordinating the creation or removal of service replicas on the Worker Nodes.

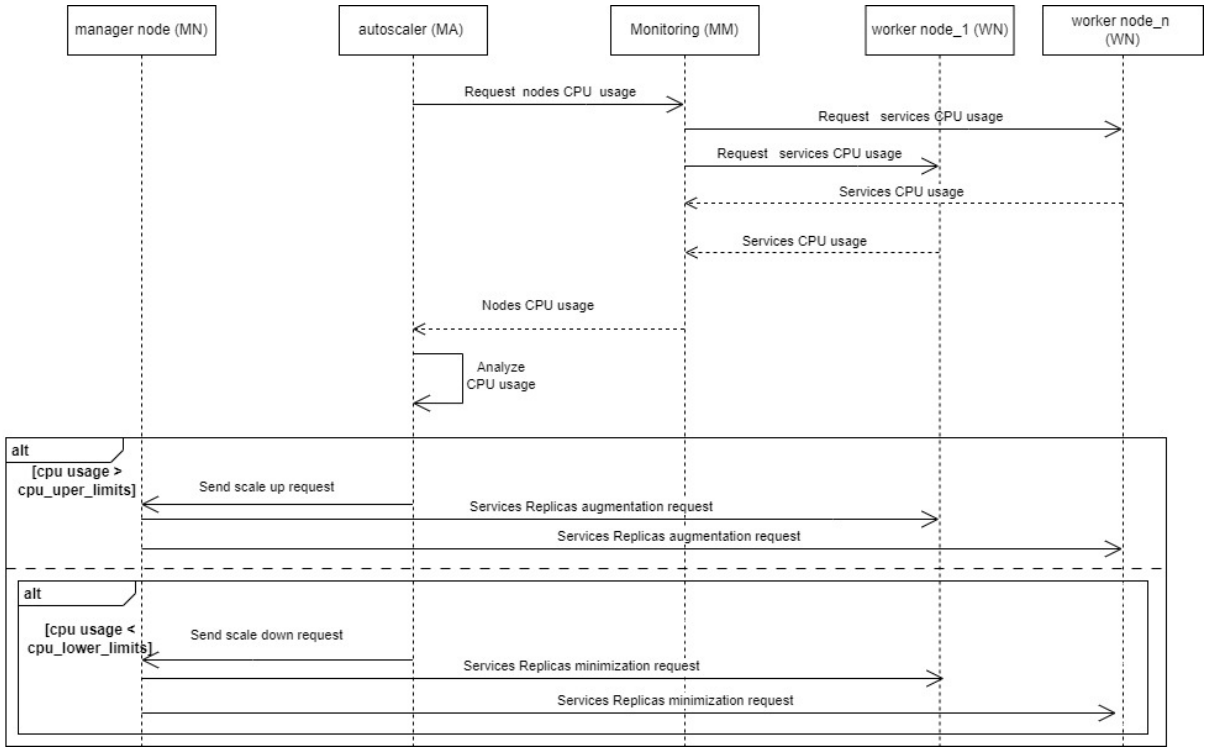

Figure III.5: sequence diagram illustrates the flow of interactions between the system modules (personal)

Table III.1: Interactions during Autoscaling Process

N°	Acte	Sender	Receiver	Content
1	Request nodes CPU usage	MA	MM	
2	Request services CPU usage	MM	WN	
3	Services CPU usage	WN	MM	list of {service_id ,CPU_usage}
4	Nodes CPU usage	MM	MA	list of {node_id,service_id ,CPU_usage}
6	Scale-up request	MA	MN	{node_id,service_id ,nbr of replicas to be added}
7	Services Replicas augmentation request	MN	WN	{service_id ,nbr of replicas to be added}
8	Scale-down request	MA	MN	{node_id,service_id ,nbr of replicas to be removed }
9	Services Replicas minimization request	MN	WN	{service_id ,nbr of replicas to be removed }

III.4 Proposed Algorithm

In this section description of the algorithms that are used in this project will be provided.

III.4.1 Autoscaler

As described in Algorithm 1 , is designed to handle the scaling of services based on their CPU usage. It takes a list of services with their corresponding CPU usage as input. The algorithm first sets the lower and upper limits for CPU usage.

The algorithm iterates over each service in the list and calls the default_scale function for each service. This function ensures that each service is scaled according to its autoscale label, replica minimum, and replica maximum values.

Following that, the algorithm performs another iteration over the services and checks if their CPU usage is below the lower limit. If a service's CPU usage is below the limit, the scale_down function is called to reduce the number of replicas for that service. The function also takes into account the autoscale label and ensures that the new number of replicas does not fall below the replica minimum value.

Lastly, the algorithm performs a third iteration over the services and checks if their CPU usage exceeds the upper limit. If a service's CPU usage is above the limit, the scale_up function is called to increase the number of replicas. The function considers the autoscale label and ensures that the new number of replicas does not exceed the replica maximum value.

Algorithm 1: autoscaler

Input : S: list of services with CPU usage

```
1  foreach service in S do
2      default_scale(service);
3  end
4  foreach service in S do
5      if service CPU usage < LOWER_LIMIT then
6          scale_down(service);
7          else
8              if service CPU usage > UPER_LIMIT then
9                  scale_up(service);
10             end
11         end
12     end
13 end
```

III.4.2 Scale Up

The ”Scale Up” algorithm 2 ,is designed to increase the number of replicas for a specified service while adhering to the replica maximum constraint. It determines the maximum number of replicas allowed for the service and retrieves the current number of replicas. By invoking the calculate_new_up_replicas() function, it calculates the desired number of new replicas based on specific criteria. If the replica maximum is greater than or equal to the calculated new replicas, the algorithm scales up the service by setting the number of replicas to the value of newReplicas. This controlled scaling approach ensures that the service can expand its capacity within the defined limits, promoting efficient resource allocation and maintaining service performance.

Algorithm 2: **function** Scale Up

Input : service: name of the service

```
1 replica_maximum ← get_replica_maximum(S);
2 current_replicas ← get_current_replicas(S);
3 new_replicas ← calculate_new_up_replicas(service);
4 if replica_maximum ≥ new_replicas then
5 |   scale_service(service, new_replicas);
6 end
```

III.4.3 Scale Down

The ”Scale Down” algorithm 3, is designed to decrease the number of replicas for a given service while adhering to the replica minimum constraint. It retrieves the replica minimum and the current number of replicas for the service, and then calculates the desired number of new replicas based on specific criteria using the calculate_new_down_replicas(service) function. The algorithm checks if the replica minimum is satisfied by comparing it to the calculated new replicas. If the condition holds true, indicating that scaling down can be done while still meeting the replica minimum, the algorithm scales down the service by setting the number of replicas to the value of newReplicas using the scale_service(service, newReplicas) function. By following this approach, the algorithm ensures controlled scaling down of the service, taking into account the replica minimum requirement and optimizing resource allocation accordingly.

Algorithm 3: **function** Scale Down :

Input : service: name of the service

```
1 replica_minimum ← get_replica_minimum(service);
2 current_replicas ← get_current_down_replicas(service);
3 new_replicas ← calculate_new_replicas(service);
4 if replica_minimum ≤ new_replicas then
5 |   scale_service(service, new_replicas);
6 end
```

III.4.4 Calculate the number of required replicas for scaling up

The ”calculate_new_up_replicas” algorithm computes the desired number of new replicas for

scaling up a service. It uses the CPU_PERCENTAGE_UPPER_LIMIT, current_replicas, and replica_maximum values to calculate ratios and a scaling factor. The new_replicas value is then determined by adding half of the scaling factor to the current number of replicas. This algorithm facilitates controlled scaling up of the service while considering resource constraints.

Algorithm 4: **function** calculate_new_up_replicas :

Input : service: name of the service, current_replicas ,
CPU_PERCENTAGE_UPPER_LIMIT, replica_maximum

Output: new_replicas

1 $n \leftarrow \frac{CPU\_PERCENTAGE\_UPPER\_LIMIT}{replica\_maximum}$;

2 $m \leftarrow \frac{CPU\_PERCENTAGE\_UPPER\_LIMIT}{current\_replicas}$;

3 $X \leftarrow \frac{m}{n}$;

4 $new\_replicas \leftarrow current\_replicas + \frac{X}{2}$;

5 **return** *new_replicas*

III.4.5 Calculate the number of required replicas for scaling down

The ”calculate_new_down_replicas” algorithm calculates the desired number of new replicas for scaling down a service based on the CPU_PERCENTAGE_LOWER _LIMIT, current_replicas, and replica_minimum. It computes the ratios n and m by dividing the CPU_PERCENTAGE_LOWER _LIMIT by the replica_minimum and current_replicas, respectively. The scaling factor X is then determined by dividing n by m. The desired number of new replicas is obtained by subtracting half of X from the current number of replicas. This algorithm enables controlled scaling down of the service while considering the lower limit of CPU usage and the minimum replica requirement.

Algorithm 5: **function** calculate_new_down_replicas

Input : service: name of the service, current_replicas,
CPU_PERCENTAGE_LOWER_LIMIT, replica_minimum

Output: new_replicas

$n \leftarrow \frac{CPU\_PERCENTAGE\_LOWER\_LIMIT}{replica\_minimum}$;

$m \leftarrow \frac{CPU\_PERCENTAGE\_LOWER\_LIMIT}{current\_replicas}$;

$X \leftarrow \frac{n}{m}$;

$new\_replicas \leftarrow current\_replicas - \frac{X}{2}$;

return *new_replicas*

III.5 Conclusion

This chapter presented the general architecture of fog computing based on container orchestration for IoT container systems. The internal architecture of the cluster was discussed, highlighting the roles of the Manager Node, Worker Nodes, Monitoring module, and Autoscaler module. An illustrative scenario demonstrated the dynamic nature of the system, where autoscaling mechanisms respond to CPU usage thresholds to optimize resource utilization. The interactions between system modules during the autoscaling process were depicted in a sequence diagram. Finally, the proposed autoscaling algorithm was introduced, consisting of the Autoscaler , Scale Up algorithm, Scale Down algorithm, and auxiliary functions for calculating the desired number of replicas.

CHAPTER IV

IMPLEMENTATION AND EXPERIMENTAL RESULTS

IV.1 Introduction

In the previous chapter, we provided an overview of the system environment and its components. In this chapter, we will delve into the essential details required to create a functional prototype. Additionally, we will include an experiment to evaluate the effectiveness of our work.

IV.2 Architecture implementation

The architecture implementation (IV.1)consists of four parts: exposing container metrics, collecting them, calculating desired replicas, and scaling services accordingly. cAdvisor exposes container CPU usage metrics. Prometheus collects metrics from cAdvisor. The Autoscaler calculates desired replicas using Prometheus metrics. Autoscaler sends scaling decisions to the Docker engine. Docker engine executes scaling commands.

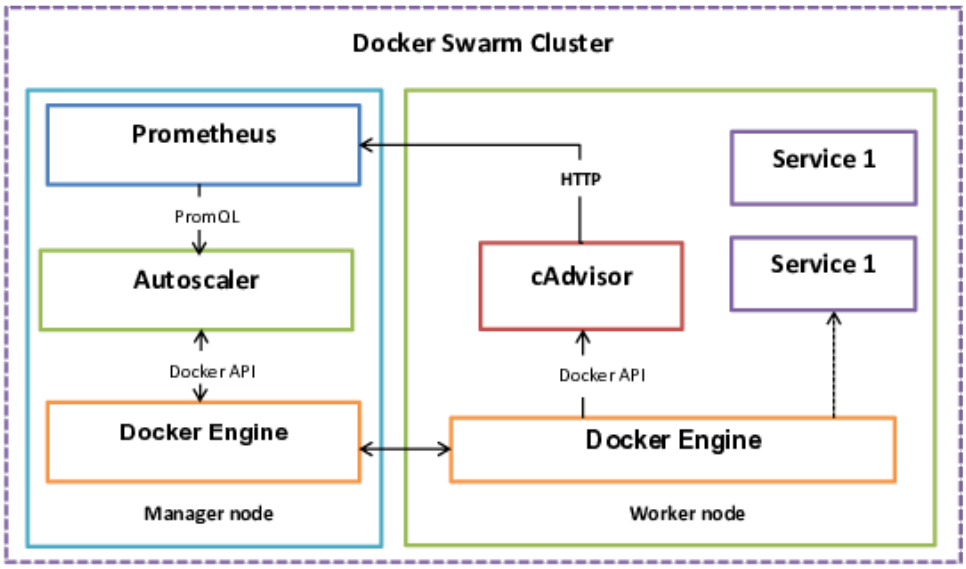

Figure IV.1: Architecture implementation (personal)

IV.2.1 Swarm cluster Configuration

In the described architecture, three VMs or nodes are created using the virtualbox driver: one Manager VM and two Worker VMs. The specifications and software installed on each VM are

as follows:

Virtual Machine Specification:

- Driver: virtualbox
- Operating System: Linux 2.6/3.x/4.x/5.x (64-bit)

Manager VM (node-1):

- Docker Engine: Installed
- Docker Swarm: Initialized as the Swarm Manager

Worker VMs (node-2 and node-3):

- Docker Engine: Installed
- Joined to Docker Swarm: Joined as worker nodes to the Swarm Manager (node-1)

in the description in Listing IV.4 the script creates three VMs using the virtualbox driver. The first VM, node-1, serves as the Manager VM and is initialized as the Docker Swarm Manager. It has Docker Engine installed. The remaining two VMs, node-2 and node-3, act as Worker VMs. They also have Docker Engine installed and are joined to the Docker Swarm Manager (node-1) as worker nodes.also sets the Docker environment for each VM using docker-machine env before executing relevant commands. The final command, docker node ls, lists the nodes in the Docker Swarm cluster, providing information about the status of the swarm.

```
#!/bin/sh

for i in 1 2 3; do
  docker-machine create -d virtualbox node-$i
done

eval $(docker-machine env node-1)

docker swarm init --advertise-addr $(docker-machine ip node-1)

TOKEN=$(docker swarm join-token -q worker)

for i in 2 3; do
  eval $(docker-machine env node-$i)
```

```
  docker swarm join --token $TOKEN $(docker-machine ip node-1):2377
done

echo "Swarm cluster has been successfuly created !";

eval $(docker-machine env node-1)

docker node ls
```

Listing IV.1: Swarm cluster Configuration

IV.2.2 Prometheus and Cadvisor

To enable scaling of a service based on CPU usage, the service's resource usage metrics need to be exposed to Prometheus. cAdvisor, a component developed by Google, is specifically designed to provide users with easy access to container resource usage insights. It has native support for Docker containers and can be run as a Docker container itself. Importantly, cAdvisor seamlessly integrates with Prometheus, allowing Prometheus to collect and scrape services metrics from cAdvisor running on each swarm node. Once the metrics are ingested by Prometheus, they can be queried and aggregated in real-time using Prometheus' query language, PromQL. To determine the average CPU usage level of the service, the metric is further aggregated using PromQL. The provided configuration IV.2is a snippet of a Prometheus configuration file (prometheus.yml) that demonstrates how to scrape metrics from two different jobs: 'prometheus' and 'cadvisor'.

```
global:
  scrape_interval:     30s
  evaluation_interval: 30s
scrape_configs:
  - job_name: 'prometheus'
    dns_sd_configs:
    - names:
      - 'tasks.prometheus'
      type: 'A'
      port: 9090
  - job_name: 'cadvisor'
    dns_sd_configs:
    - names:
```

```
14      - 'tasks.cadvisor'
15      type: 'A'
16      port: 8080
```

Listing IV.2: Prometheus configuration

IV.2.3 Autoscaler implementation

Autoscaler is a daemon process that periodically monitors Prometheus metrics and scales services in a Docker Swarm cluster. It utilizes service labels to control its scaling behavior. The following service labels are used:

swarm_autoscaler: This label enables or disables automatic service scaling. If the value is not set to "true," autoscaling will not be enabled for the service.

swarm_autoscalerminimum: This label sets the minimum number of replicas desired for a service. The autoscaler will not downscale the service below this number.

swarm_autoscaler_maximum: This label sets the maximum number of replicas desired for a service. The autoscaler will not scale up the service beyond this number.

The Autoscaler detects services that have these labels and fetches their metrics from Prometheus. based on this metrics, it dynamically scales the services by interacting with the Docker API. The Autoscaler can be implemented in programming languages that can utilize the PromQL API to retrieve metrics from Prometheus and the Docker API to scale the services.

In the proof-of-concept implementation mentioned, a shell script was used to connect to Prometheus and interact with the Docker API using the Docker client package. This allowed the script to retrieve metrics and scale the services accordingly based on the specified minimum and maximum replica values.

IV.2.3.1 Autoscaling variables and configuration parameters

The provided sets up default values for various configuration parameters required for an autoscaling process ,as show in ListingIV.3 .It defines upper and lower threshold limits for CPU usage through the CPUPERCENTAGE_UPPER-LIMIT and CPU_PERCENTAGE_LOWER_LIMIT variables, respectively,with default values of 75% and 25% if not explicitly set. Additionally, establishes the Prometheus API endpoint through the PROMETHEUS_API variable and defines the specific query, stored in PROMETHEUS_QUERY, for retrieving CPU usage metrics of labeled services within a Docker Swarm.

```
#!/bin/bash
CPU_PERCENTAGE_UPPER_LIMIT=75
CPU_PERCENTAGE_LOWER_LIMIT=25
PROMETHEUS_API="api/v1/query?query="
PROMETHEUS_QUERY="sum(rate(container_cpu_usage_seconds_total%7
    Bcontainer_label_com_docker_swarm_task_name%3D~%27.%2B%27%7D%5B1m%5D)
    )BY(container_label_com_docker_swarm_service_name%2Cinstance)*100"
```

Listing IV.3: initial variables and configuration parameters for an autoscaling process

IV.2.3.2 Collection of service names

The get_all_services function (listig IV.4) extracts service names from the prometheus-results parameter, which holds the results obtained from Prometheus. It utilizes the jq command-line tool to retrieve the service names by matching the container-label-com-docker-swarm-service-name metric label. The extracted service names are then stored in the services variable, encompassing all the services present in the Prometheus results.

```
get_all_services () {
  local prometheus_results="${1}"
  local services=""
  for service in $(printf "%s$prometheus_results" | jq ".data.result[].
   metric.container_label_com_docker_swarm_service_name" | sed 's/"//g'
   | sort | uniq); do
    services="$services $service"
  done
  echo $services
}
```

Listing IV.4: the function provides a comprehensive list of all the services available in the Docker Swarm environment

IV.2.3.3 Identify the high and low CPU usage services

As shown in the listingIV.5,the functions get-high-cpu-services and get-low-cpu-services process Prometheus results to identify services with CPU usage exceeding or below specific thresholds, respectively. The functions extract the service names by filtering and processing the results using jq, sorting them, and making them unique. The result is a list of service names exhibiting high or low CPU usage, respectively. These functions facilitate efficient resource allocation in a

Docker Swarm environment by enabling the identification of services that require scaling based on their CPU usage.

```
get_high_cpu_services () {
  local prometheus_results="${1}"
  local services=""
  for service in $(printf "%s$prometheus_results" | jq ".data.result[] |
   select( all(.value[1]|tonumber; . > $CPU_PERCENTAGE_UPPER_LIMIT) ) |
   .metric.container_label_com_docker_swarm_service_name" | sed 's/"//g'
    | sort | uniq); do
    services="$services $service"
  done
  echo $services
}
get_low_cpu_services () {
  local prometheus_results="${1}"
  local services=""
  for service in $(printf "%s$prometheus_results" | jq ".data.result[] |
   select( all(.value[1]|tonumber; . < $CPU_PERCENTAGE_LOWER_LIMIT) ) |
   .metric.container_label_com_docker_swarm_service_name" | sed 's/"//g'
    | sort | uniq); do
    services="$services $service"
  done
  echo $services
}
```

Listing IV.5: The two functions identify the high and low CPU usage services for further processing

IV.2.3.4 The scale up/down process

The "scale_up()" function (listing IV.6) performs scaling up for a specified Docker service. It starts by retrieving the necessary parameters such as CPU_PERCENTAGE_UPPER_LIMIT, replica_maximum, and current_replicas. then calculates ratios and a scaling factor based on these parameters. The desired number of new replicas is determined by adding half of the scaling factor to the current number of replicas. If the current replicas are already at the maximum allowed, a message is displayed. Otherwise, if the new replicas fall within the replica_maximum limit, the function scales up the service by invoking the "docker service scale" command.

```
1
2  scale_up () {
3    service_name=$1
4    auto_scale_label=$(docker service inspect $service_name | jq '.[].Spec.
      Labels["swarm.autoscaler"]')
5    replica_maximum=$(docker service inspect $service_name | jq '.[].Spec.
      Labels["swarm.autoscaler.maximum"]' | sed 's/\"//g')
6    if [[ "${auto_scale_label}" == "\"true\"" ]]; then
7      current_replicas=$(docker service inspect $service_name | jq ".[].
      Spec.Mode.Replicated | .Replicas")
8      n=$(expr $CPU_PERCENTAGE_UPPER_LIMIT / $replica_maximum)
9      m=$(expr $CPU_PERCENTAGE_UPPER_LIMIT / $current_replicas)
10     x=$(expr $m / $n)
11     new_replicas=$(expr $current_replicas + $x / 2)
12     if [[ $current_replicas -eq $replica_maximum ]]; then
13       echo Service $service already has the maximum of $replica_maximum
      replicas
14     elif [[ $replica_maximum -ge $new_replicas ]]; then
15       echo Scaling up the service $service_name to $new_replicas
16       docker service scale $service_name=$new_replicas
17     fi
18   fi
19 }
```

Listing IV.6: scale up process

The ”scale_down()” function (listing IV.7) is responsible for scaling down a Docker service. It sets the CPU_PERCENTAGE_LOWER_LIMIT and retrieves the necessary parameters. If scaling is enabled, it calculates ratios and a scaling factor based on the CPU_PERCENTAGE _LOWER_LIMIT and replica_minimum. The desired number of new replicas is determined by subtracting half of the scaling factor from the current number of replicas. If the new replicas fall within the replica_minimum limit, the function scales down the service. It also handles the case when the current replicas are already at the minimum. This algorithm ensures controlled scaling down of the service based on resource constraints and the defined CPU lower limit.

```
1
2  scale_down () {
3
4    service_name=$1
```

```
  auto_scale_label=$(docker service inspect $service_name | jq '.[].Spec.
   Labels["swarm.autoscaler"]')
  replica_minimum=$(docker service inspect $service_name | jq '.[].Spec.
   Labels["swarm.autoscaler.minimum"]' | sed 's/\"//g')
  if [[ "${auto_scale_label}" == "\"true\"" ]]; then
    current_replicas=$(docker service inspect $service_name | jq ".[].
   Spec.Mode.Replicated | .Replicas")
    n=$(expr $CPU_PERCENTAGE_LOWER_LIMIT / $replica_minimum)
    m=$(expr $CPU_PERCENTAGE_LOWER_LIMIT / $current_replicas)
    x=$(expr $n / $m)
    new_replicas=$(expr $current_replicas - $x / 2)
    if [[ $replica_minimum -le $new_replicas ]]; then
      echo Scaling down the service $service_name to $new_replicas
      docker service scale $service_name=$new_replicas
    elif [[ $current_replicas -eq $replica_minimum ]]; then
      echo Service $service_name has the minumum number of replicas.
    fi
  fi

}
```

Listing IV.7: scale down process

IV.2.3.5 Default scaling behavior for services

This default-scaling function (listing IV.8) ensures that services are automatically adjusted to meet their desired replica count based on the specified minimum and maximum values. When the swarm.autoscaler label is set to ”true”, the function checks if the current replica count falls below the minimum threshold and scales the service up accordingly. Similarly, if the replica count exceeds the maximum threshold, the function scales the service down to maintain the desired replica count. This mechanism ensures that services are efficiently scaled within the defined boundaries.

```
default_scale () {
  service_name=$1
  auto_scale_label=$(docker service inspect $service_name | jq '.[].Spec.
   Labels["swarm.autoscaler"]')
  replica_minimum=$(docker service inspect $service_name | jq '.[].Spec.
   Labels["swarm.autoscaler.minimum"]' | sed 's/\"//g')
```

```
5   replica_maximum=$(docker service inspect $service_name | jq '.[].Spec.
     Labels["swarm.autoscaler.maximum"]' | sed 's/\"//g')
6   if [[ "${auto_scale_label}" == "\"true\"" ]]; then
7     echo Service $service has an autoscale label.
8     current_replicas=$(docker service inspect $service_name | jq ".[].
    Spec.Mode.Replicated | .Replicas")
9     if [[ $replica_minimum -gt $current_replicas ]]; then
10      echo Service $service_name is below the minimum. Scaling to the
    minimum of $replica_minimum
11      docker service scale $service_name=$replica_minimum
12    elif [[ $current_replicas -gt $replica_maximum ]]; then
13      echo Service $service_name is above the maximum. Scaling to the
    maximum of $replica_maximum
14      docker service scale $service_name=$replica_maximum
15    fi
16  else
17    echo Service $service does not have an autoscale label.
18  fi
19
20 }
```

Listing IV.8: This function provides the default scaling behavior for services based on the autoscale label and the specified minimum and maximum replica values.

IV.3 Experimental

The basic goal of the experimental procedure that we followed, is to prove that our autoscaler works correctly and that it has the desirable effect on the system, which means achieving lower response time for a great concurrency number through automatically scaling up the service in the system, and vice versa for the respective case. In order to acquire the necessary results, we selected to use a CPU intensive testing application. We expand on the subject of this testing application and we analyse the results of our evaluation in the following sections.

IV.3.1 Client side (Request generator: abache bench)

To evaluate the performance of our system under realistic stress conditions, we use Apache Bench [63], a tool specifically designed for benchmarking HTTP Web servers. This software

enables us to simulate multiple concurrent users and stress our application accordingly, generating a two CPU-intensive scenarios(normal scenario ,stressed scenario). By executing a number of requests, we can measure the system's performance based on the response time for each request. This approach allows us to assess the systems capability to handle high loads and gauge its efficiency under such conditions.

IV.3.2 Performance metrics

The performance metrics in the experiments are average of waiting time (W_Avg), average of processing time (P_Avg), average of total time(T_Avg).

The average of processing time (P_Avg) represents the time taken to process a request, while average of waiting time (W_Avg) represents the time a request spends waiting in the queue before being processed. The mean total time (response time) represents the sum of the mean processing and mean waiting times.

$$T_Avg = W_Avg + P_Avg$$

IV.3.3 Hardware environment

Dell OptiPlex 3050 Micro Desktop PC.

Table IV.1: Characteristics of the computer used

Processor	Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz (4 CPUs), 2.4GHz
RAM memory	8 GB
System type	64-bit
Operating system	Ubuntu 22.10 (Kinetic Kudu)

IV.3.4 Experimental set-up

Table IV.2: Service parameter

service name	initial_replicas	min_replicas	max_replicas	cpu_uper_limits %	cpu_lower_limits %
ctest_app	3	3	15	60%	20%

Table IV.3: Test parameter

	description	test1:normal scenario	test2 : stressed scenario
-n requests	Number of requests to perform for the benchmarking session	400 requests	400 requests
-c concurrency	Number of multiple requests to perform at a time.	10	100

IV.3.5 Test results for Our Approach versus static approach

To evaluate our approach, we compare it to the static approach by conducting several tests .

In the static approach , scaling up or down involves adding or removing one instance of the service, respectively. The scaling is done in a fixed manner, incrementing or decrementing the number of service instances by one unit at a time.[64]

In the first test, we load the service once by applying the static approach and once by applying our dynamic approach to autoscaling. The chart bar in the figure IV.2 show the test results. We see that our approach has the lowest response time than the fixed approach. The static approach had a mean processing time(P_Avg) of 40,708 ms, Additionally, the mean waiting time was 20,417 ms . The total mean time, combining processing and waiting, was 40,709 ms.

In contrast, the dynamic approach showed improved performance. It achieved a mean processing time(P_Avg) of 32,285 ms, indicating faster processing compared to the static approach. The mean waiting time for the dynamic approach was also 32,284 ms, demonstrating a more balanced workload distribution. The total mean time for the dynamic approach was 32,288 ms. The provided results show (figure IV.3)the percentage of requests served within a certain time for our dynamic approache and the static approache. Overall, the results highlights the improved performance of our dynamic approach, as it consistently exhibits lower response times across different percentiles compared to the static approach. our dynamic approach demonstrates better scalability and efficiency in serving requests within the specified time thresholds. These results suggest that the our autoscling dynamic approach was able to optimize the connection times. By considering these mean results and their implications, it is evident that the our approach outperformed in terms of response time.

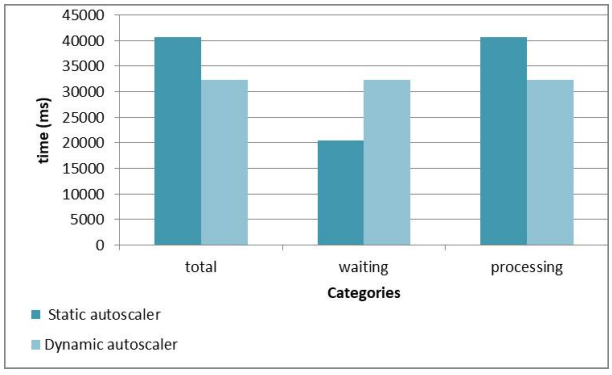

Figure IV.2: Average time result for test 1

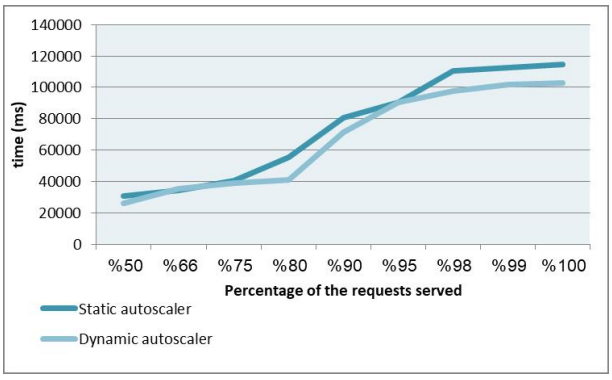

Figure IV.3: Request Time Percentile Distribution result for test 1

The dynamic autoscaler showcased its efficiency and effectiveness in handling increased con-

currency in test 2.

The bar chart in figure IV.4 clearly illustrates the comparison between the dynamic autoscaler and the static approach in terms of response time. The dynamic autoscaler consistently achieved the lowest response time across all measures. This indicates that the dynamic autoscaler allows the service to process requests more quickly and provide a faster response to users compared to the static approach.

Additionally, the curve chart (figure IV.4) provides a representation of the percentage of requests served within a certain time for both approaches. It clearly shows that the dynamic autoscaler maintained lower response times consistently throughout the test duration, demonstrating its ability to handle increased concurrency without sacrificing performance.

Among These results, the effectiveness of our approach (specifically the dynamic autoscaler) is highlighted in managing high levels of requests concurrency By dynamically scaling of resources based on the workload. Overall, the dynamic autoscaler is able to optimize Fog computing Infrastructures performance and provide a superior user experience for Fog Service providers.

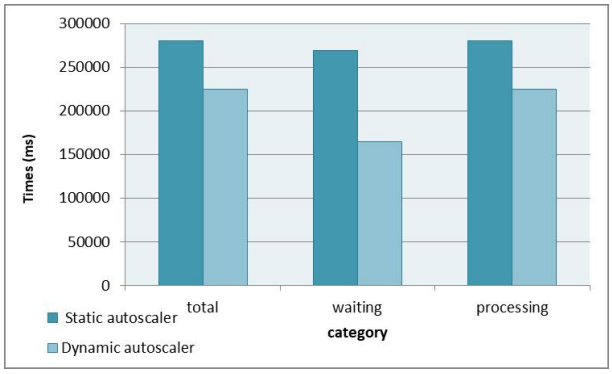

Figure IV.4: Average time result for test 2

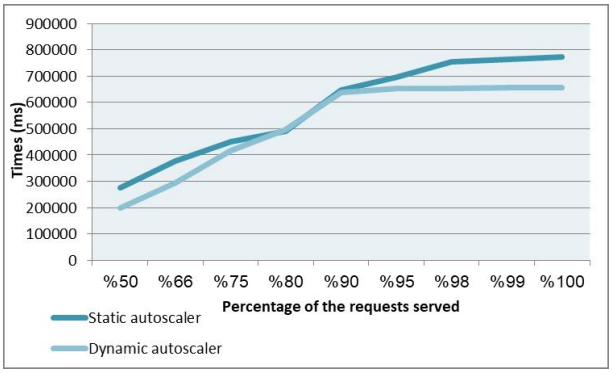

Figure IV.5: Request Time Percentile Distribution result for test 2

IV.4 Conclusion

Based on the implementation and experimental results presented in this Chapter ,our autoscaling approach shows promise in providing better performance The implementation of the autoscaler using Prometheus and Cadvisor, along with the identification of high and low CPU usage services, allowed for efficient scaling up and down of services in response to changes in demand.

GENERAL CONCLUSION

In conclusion, We have developed an auto-scaling algorithm specifically designed for Fog Computing infrastructures that utilize a containerized architecture. This algorithm addresses the need to efficiently manage resources and dynamically adjust the number of containers based on workload demands. Our algorithm incorporates continuous monitoring of crucial metrics such as CPU utilization. By setting thresholds, we are able to determine the optimal triggers for scaling actions. When a workload metric exceeds its threshold, our algorithm initiates a scale-out action, which involves launching additional containers to effectively distribute the workload and accommodate the increased demand. Conversely, when the workload decreases and remains below a certain threshold, our algorithm initiates a scale-in action to remove excess containers. This process ensures optimal resource utilization and avoids unnecessary overhead.

To further enhance performance, our algorithm incorporates capacity planning to determine the maximum and minimum number of containers that can be deployed based on available resources, performance requirements, and expected workload variations. By defining scaling policies, we specify the number of containers to be added or removed during scale-up and scale-down actions. This allows for fine-tuning the autoscaling behavior to align with specific workload characteristics and performance objectives. To automate these scaling operations, we leverage popular container orchestration frameworks such as Docker Swarm. This frameworks provide the necessary tools and mechanisms to seamlessly manage the deployment and scaling of containers based on our algorithm's policies. Continuous monitoring and feedback play a vital role in our algorithm. We continuously evaluate the performance and resource usage of the scaled infrastructure, ensuring that our scaling decisions align with the desired outcomes. Adjustments

to scaling policies and thresholds are made based on real-time feedback and observed behavior, allowing for continuous optimization and improved performance.

Experimental results show that our mechanism can dynamically scale Docker containers by provisioning sufficient resources to meet the required demand and reduce response time .

Future Work The following are important issues for future work:

- Extend our system capabilities by adding machine learning in order to gain insight as to what would be the best threshold value for triggering the auto-scaling.
- Enable our system to make decisions regarding scaling, we need to incorporate multiple metrics such as memory usage, read and write operations, and other relevant factors, in addition to CPU usage. By considering a broader range of metrics, our system can make more informed decisions when determining whether to expand or reduce resources.
- Provide a mechanisms to allow automatic scaling up and down for worker and manger nodes .
- Add a graphical interface to facilitate the monitoring of the automatic scaling process.

REFERENCES

[1] Mohammad Aazam, Imran Khan, Aymen Abdullah Alsaffar, and Eui-Nam Huh. Cloud of things: Integrating internet of things and cloud computing and the issues involved. In *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences & Technology (IBCAST) Islamabad, Pakistan, 14th-18th January, 2014*, pages 414–419. IEEE, 2014.

[2] Pengfei Hu, Sahraoui Dhelim, Huansheng Ning, and Tie Qiu. Survey on fog computing: architecture, key technologies, applications and open issues. *Journal of network and computer applications*, 98:27–42, 2017.

[3] Qi Zhang, Ling Liu, Calton Pu, Qiwei Dou, Liren Wu, and Wei Zhou. A comparative study of containers and virtual machines in big data environment. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 178–185. IEEE, 2018.

[4] Swarm mode key concepts. `https://docs.docker.com/engine/swarm/key-concepts/`. Accessed: 2023.03.12.

[5] Somayya Madakam, Vihar Lake, Vihar Lake, Vihar Lake, et al. Internet of things (iot): A literature review. *Journal of Computer and Communications*, 3(05):164, 2015.

[6] Sadiq Ur Rehman Aqeel-ur Rehman, Iqbal Uddin Khan, Muzaffar Moiz, and Sarmad Hasan. Security and privacy issues in iot. *International Journal of Communication Networks and Information Security (IJCNIS)*, 8(3):147–157, 2016.

[7] Rafiullah Khan, Sarmad Ullah Khan, Rifaqat Zaheer, and Shahid Khan. Future internet: the internet of things architecture, possible applications and key challenges. In *2012 10th international conference on frontiers of information technology*, pages 257–260. IEEE, 2012.

[8] Karen Rose, Scott Eldridge, and Lyman Chapin. The internet of things: An overview. *The internet society (ISOC)*, 80:1–50, 2015.

[9] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the internet of things. *IEEE access*, 6:6900–6919, 2017.

[10] Mahantesh N Birje, Praveen S Challagidad, RH Goudar, and Manisha T Tapale. Cloud computing review: concepts, technology, challenges and security. *International Journal of Cloud Computing*, 6(1):32–57, 2017.

[11] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 27–33. Ieee, 2010.

[12] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[13] Aaqib Rashid and Amit Chaturvedi. Cloud computing characteristics and services: a brief review. *International Journal of Computer Sciences and Engineering*, 7(2):421–426, 2019.

[14] Ahmed E Youssef. Exploring cloud computing services and applications. *Journal of Emerging Trends in Computing and Information Sciences*, 3(6):838–847, 2012.

[15] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[16] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020.

[17] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235, 2019.

[18] Farah Ait Salaht, Frédéric Desprez, and Adrien Lebre. An overview of service placement problem in fog and edge computing. *ACM Computing Surveys (CSUR)*, 53(3):1–35, 2020.

[19] Mahadev Satyanarayanan. How we created edge computing. *Nature Electronics*, 2(1):42–42, 2019.

[20] Peng Zhang, Joseph K Liu, F Richard Yu, Mehdi Sookhak, Man Ho Au, and Xiapu Luo. A survey on access control in fog computing. *IEEE Communications Magazine*, 56(2):144–149, 2018.

[21] Resul Das and Muhammad Muhammad Inuwa. A review on fog computing: issues, characteristics, challenges, and potential applications. *Telematics and Informatics Reports*, page 100049, 2023.

[22] Subhadeep Sarkar and Sudip Misra. Theoretical modelling of fog computing: a green computing paradigm to support iot applications. *Iet Networks*, 5(2):23–29, 2016.

[23] Mithun Mukherjee, Rakesh Matam, Lei Shu, Leandros Maglaras, Mohamed Amine Ferrag, Nikumani Choudhury, and Vikas Kumar. Security and privacy in fog computing: Challenges. *IEEE Access*, 5:19293–19304, 2017.

[24] Mohammad Aazam and Eui-Nam Huh. Dynamic resource provisioning through fog micro datacenter. In *2015 IEEE international conference on pervasive computing and communication workshops (PerCom workshops)*, pages 105–110. IEEE, 2015.

[25] Ivan Stojmenovic and Sheng Wen. The fog computing paradigm: Scenarios and security issues. In *2014 federated conference on computer science and information systems*, pages 1–8. IEEE, 2014.

[26] Olena Skarlat, Stefan Schulte, Michael Borkowski, and Philipp Leitner. Resource provisioning for iot services in the fog. In *2016 IEEE 9th international conference on service-oriented computing and applications (SOCA)*, pages 32–39. IEEE, 2016.

[27] Muhammad Fahimullah, Shohreh Ahvar, and Maria Trocan. A review of resource management in fog computing: Machine learning perspective. *arXiv preprint arXiv:2209.03066*, 2022.

[28] Mostafa Ghobaei-Arani, Alireza Souri, and Ali A Rahmanian. Resource management approaches in fog computing: a comprehensive review. *Journal of Grid Computing*, 18(1):1–42, 2020.

[29] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference*, pages 1–13, 2016.

[30] Jenni Susan Reuben. A survey on virtual machine security. *Helsinki University of Technology*, 2(36), 2007.

[31] Christian Plessl and Marco Platzner. Virtualization of hardware-introduction and survey. In *ERSA*, pages 63–69, 2004.

[32] Max Plauth, Lena Feinbube, and Andreas Polze. A performance survey of lightweight virtualization techniques. In *Service-Oriented and Cloud Computing: 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings 6*, pages 34–48. Springer, 2017.

[33] Tetiana Yarygina and Anya Helene Bagge. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 11–20. IEEE, 2018.

[34] Sam Newman. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[35] Adam Irufaan, Hemalata Vasudavan, and Palvinderjit Kaur Harnek Singh. Microservice dynamic resource provision for small and medium-sized enterprises. *Journal of Applied Technology and Innovation (e-ISSN: 2600-7304)*, 5(1):15, 2021.

[36] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, 2017.

[37] Ouafa Bentaleb, Adam SZ Belloum, Abderrazak Sebaa, and Aouaouche El-Maouhab. Containerization technologies: Taxonomies, applications and challenges. *The Journal of Supercomputing*, 78(1):1144–1181, 2022.

[38] Nguyen Dinh Nguyen, Linh-An Phan, Dae-Heon Park, Sehan Kim, and Taehong Kim. Elasticfog: Elastic resource provisioning in container-based fog computing. *IEEE Access*, 8:183879–183890, 2020.

[39] Moby Dock. Docker: Accelerated, containerized application development. https://www.docker.com/, May 2022. Accessed: 2023-5-22.

[40] Azure container registry. https://azure.microsoft.com/en-us/products/container-registry/. Accessed: 2023-5-22.

[41] Vagrant by HashiCorp. https://www.vagrantup.com/. Accessed: 2023-5-22.

[42] Cedric Clyburn. Buildah. https://buildah.io/. Accessed: 2023-5-22.

[43] Podman. https://podman.io/. Accessed: 2023-5-22.

[44] Containerd. https://containerd.io/. Accessed: 2023-5-22.

[45] Run system containers with LXD. https://ubuntu.com/lxd. Accessed: 2023-5-22.

[46] ZeroVM. https://www.zerovm.org/. Accessed: 2023-5-22.

[47] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[48] James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[49] Amit M Potdar, DG Narayan, Shivaraj Kengond, and Mohammed Moin Mulla. Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171:1419–1428, 2020.

[50] Hamzeh Khazaei, Rajsimman Ravichandiran, Byungchul Park, Hadi Bannazadeh, Ali Tizghadam, and Alberto Leon-Garcia. Elascale: autoscaling and monitoring as a service. *arXiv preprint arXiv:1711.03204*, 2017.

[51] Yang Zhang, Gang Yin, Tao Wang, Yue Yu, and Huaimin Wang. An insight into the impact of dockerfile evolutionary trajectories on quality and latency. In *2018 ieee 42nd annual computer software and applications conference (compsac)*, volume 1, pages 138–143. IEEE, 2018.

[52] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[53] Tuomas Vase. Advantages of docker. 2015.

[54] Yao Pan, Ian Chen, Francisco Brasileiro, Glenn Jayaputera, and Richard Sinnott. A performance comparison of cloud-based container orchestration tools. In *2019 IEEE International Conference on Big Knowledge (ICBK)*, pages 191–198. IEEE, 2019.

[55] Emiliano Casalicchio. Container orchestration: A survey. *Systems Modeling: Methodologies and Tools*, pages 221–235, 2019.

[56] Saiful Hoque, Mathias Santos De Brito, Alexander Willner, Oliver Keil, and Thomas Magedanz. Towards container orchestration in fog computing infrastructures. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 294–299. IEEE, 2017.

[57] Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, and Amedeo Palopoli. Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2019.

[58] Marek Moravcik and Martin Kontsek. Overview of docker container orchestration tools. In *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 475–480. IEEE, 2020.

[59] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12:559–592, 2014.

[60] Tianqi Yu, Xianbin Wang, and Abdallah Shami. A novel fog computing enabled temporal data reduction scheme in iot systems. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–5. IEEE, 2017.

[61] David De La Bastida and Fuchun Joseph Lin. Openstack-based highly scalable iot/m2m platforms. In *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 711–718. IEEE, 2017.

[62] Chia-Chen Chang, Shun-Ren Yang, En-Hau Yeh, Phone Lin, and Jeu-Yih Jeng. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[63] ab - apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/2.4/programs/ab.html`. Accessed: 2023-5-23.

[64] Jacob Cody Wimer. docker-swarm-autoscaler: Autoscale docker swarm services based on cpu utilization.