NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

PEDRO NOBRE OLIVEIRA DOS SANTOS

Degree in Computer Science

# TOWARDS AN AUTOMATIC MICROSERVICES MANAGER FOR HYBRID CLOUD EDGE ENVIRONMENTS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
July, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

# TOWARDS AN AUTOMATIC MICROSERVICES MANAGER FOR HYBRID CLOUD EDGE ENVIRONMENTS

**PEDRO NOBRE OLIVEIRA DOS SANTOS**

Degree in Computer Science

**Adviser:** Maria Cecília Farias Lorga Gomes
*Prof. Auxiliar, Faculdade de Ciências e Tecnologia da UNL - Dep. de Informática*

**Co-adviser:** Vítor Manuel Alves Duarte
*Prof. Auxiliar, Faculdade de Ciências e Tecnologia da UNL - Dep. de Informática*

**Examination Committee**

**Chair:** Fernando Pedro Reino da Silva Birra
*Prof. Auxiliar, Faculdade de Ciências e Tecnologia da UNL - Dep. de Informática*

**Adviser:** Maria Cecília Farias Lorga Gomes
*Prof. Auxiliar, Faculdade de Ciências e Tecnologia da UNL - Dep. de Informática*

**Member:** Carlos Jorge de Sousa Gonçalves
*Prof. Adjunto, ISEL - Dep. Eng. Electrotécnica e Telecomunicações e de Computadores*

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
July, 2022

**Towards an Automatic Microservices Manager for Hybrid Cloud Edge Environments**



This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.15) [1].

*Mãe, foste és e serás sempre o meu orgulho..*

# Acknowledgements

Primeiro gostaria de agradecer à minha orientadora Prof.ª Maria Cecília Farias Lorga Gomes e ao meu co-orientador Vítor Manuel Alves Duarte pela oportunidade, pelo conhecimento transmitido, pela disponibilidade e muita paciência, por terem acreditado nas minhas capacidades, pelas palavras e motivação. Principalmente pela orientação e disponibilidade ao longo do projecto e por isso, um muito obrigado.

Gostaria de agradecer à Faculdade de Ciências e Tecnologia, que muito bem me acolheu ao longo deste meu percurso, pelo conhecimento, e grandes amizades que me proporcionou.

Aos grandes amigos que conheci na faculdade pelo apoio e por todo o incentivo que me deram nas mais diferentes situações. Que todos os momentos incríveis que passamos, sejam eternos na nossa memória. Obrigado pelo vosso apoio e companhia.

A todos os amigos que não estiveram comigo na Faculdade, mas que me acompanham e acompanharam nesta corrida que é a vida, muitos são parte da família. Por todas as palavras de apoio, por toda a companhia e ajuda, por todos os momentos, mesmo que menos bons, um muito obrigado, contem sempre comigo.

Por fim e muito importante, aos meus irmãos (primo incluído) e família, por todo o carinho e amor nas diferentes formas e por todo o apoio. Ao meu pai por ser mais do que um exemplo, por ter segurado a família nos bons e maus momentos, um eterno obrigado pelo amor, carinho e apoio. Família de sangue e sem ser de sangue, um grande obrigado por acreditarem nas minhas capacidades e por se orgulharem de mim.

Todos ajudaram a tornar isto possível, mesmo que não saibam, Obrigado.

*"A society grows great when old men plant trees*
*whose shade they know they shall never sit in."*
*(Greek Proverb)*

# Abstract

Cloud computing came to make computing resources easier to access thus helping a faster deployment of applications/services benefiting from the scalability provided by the service providers. It has been registered an exponential growth of the data volume received by the cloud. This is due to the fact that almost every device used in everyday life are connected to the internet sharing information in a global scale (ex: smartwatches, clocks, cars, industrial equipment's). Increasing the data volume results in an increased latency in client applications resulting in the degradation of the QoS (Quality of service).

With these problems, hybrid systems were born by integrating the cloud resources with the various edge devices between the cloud and edge, Fog/Edge computation. These devices are very heterogeneous, with different resources capabilities (such as memory and computational power), and geographically distributed.

Software architectures also evolved and microservice architecture emerged to make application development more flexible and increase their scalability. The Microservices architecture comprehends decomposing monolithic applications into small services each one with a specific functionality and that can be independently developed, deployed and scaled. Due to their small size, microservices are adquate for deployment on Hybrid Cloud/Edge infrastructures. However, the heterogeneity of those deployment locations makes microservices' management and monitoring rather complex. Monitoring, in particular, is essential when considering that microservices may be replicated and migrated in the cloud/edge infrastructure.

The main problem this dissertation aims to contribute is to build an automatic system of microservices management that can be deployed in hybrid infrastructures cloud/fog computing. Such automatic system will allow edge enabled applications to have an adaptive deployment at runtime in response to variations in workloads and computational resources available. Towards this end, this work is a first step on integrating two existing projects that combined may support an automatic system. One project does the automatic management of microservices but uses only an heavy monitor, Prometheus, as a cloud monitor. The second project is a light adaptive monitor. This thesis integrates the light monitor into the automatic manager of microservices.

**Keywords:** Microservices, cloud, edge, fog, microservice deployment, microservice management, microservice monitoring

# Resumo

A computação na Cloud surgiu como forma de simplificar o acesso aos recursos computacionais, permitindo um deployment mais rápido das aplicações e serviços como resultado da escalabilidade suportada pelos provedores de serviços.

Computação na *cloud* surgiu para facilitar o acesso aos recursos de computação provocando um facultamento no *deployment* de aplicações/serviços sendo benéfico para a escalabilidade fornecida pelos provedores de serviços. Tem-se registado um crescimento exponencial do volume de data que é recebido pela *cloud*. Este aumento deve-se ao facto de quase todos os dispositivos utilizados no nosso quotidiano estarem conectados à internet (exemplos destes são, relogios, maquinas industriais, carros). Este aumento no volume de dados resulta num aumento da latência para as aplicações cliente, resultando assim numa degradação na qualidade de serviço *QoS*.

Com estes problemas, nasceram os sistemas híbridos, nascidos pela integração dos recursos de *cloud* com os variados dispositivos presentes no caminho entre a *cloud* e a periferia denominando-se computação na *Edge*/*Fog* (Computação na periferia). Estes dispositivos apresentam uma grande heterogeneidade e são geograficamente muito distribuídos.

As arquitecturas dos sistemas também evoluíram emergindo a arquitectura de micro serviços que permitem tornar o desenvolvimento de aplicações não só mais flexivel como para aumentar a sua escalabilidade. A arquitetura de micro serviços consiste na decomposição de aplicações monolíticas em pequenos serviços, onde cada um destes possuí uma funcionalidade específica e que pode ser desenvolvido, lançado e migrado de forma independente. Devido ao seu tamanho os micro serviços são adequados para serem lançados em ambientes de infrastructuras híbridas (*cloud* e periferia). No entanto, a heterogeneidade da localização para serem lançados torna a gestão e monitorização de micro serviços bastante mais complexa. A monitorização, em particular, é essencial quando consideramos que os micro serviços podem ser replicados e migrados nestas infrastruturas de *cloud* e periferia (*Edge*).

O problema abordado nesta dissertação é contribuir para a construção de um sistema automático de gestão de micro serviços que podem ser lançados em estruturas hibridas. Este sistema automático irá tornar possível às aplicações que estão na edge possuírem um *deploy* adaptativo enquanto estão em execução, como resposta às variações dos recursos computacionais disponíveis e suas cargas. Para chegar a este fim, este trabalho será o

primeiro passo na integração de dois projectos já existentes que, juntos poderão suportar um sistema automático. Um deles realiza a gestão automática de micro serviços mas utiliza apenas o Prometheus como monitor na *cloud*, enquanto o segundo projecto é um monitor leve adaptativo. Esta tese integra então um monitor leve com um gestor automático de micro serviços.

**Palavras-chave:** Microservices, cloud, edge, fog, microservice deployment, microservice management, microservice monitoring

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

Continuous evolution and innovation in technology created the necessity for companies to reorganize the management of their IT Systems and facilitate continuous delivery [2]. Such allows an organization to answer quickly to costumers and its market since continuous delivery enables releasing software with lower risk and fewer bugs, making new features available more frequently to quickly respond to pressing market conditions.[1] [3]. Other aspects to take into account to enforce the evolution and re-adaption of the systems are the data volume generated by Internet of Things (IoT) environments or even the smartness in IoT. Smartness in IoT refers to a smart network with 1) Object adressability and multi-functionality, which means the objects have a direct IP address and the network used in one application is available to be used in other purposes, 2) Open communication standards are used at different levels, from layers interfacing the real world (sensors, etc) to communication layers between the internet and computational [4].

Several common architectural solutions are not prepared for this innovation and upgrading pace, despite being easy to build and keeping the business going. For instance, in a monolithic architecture new functionalities or a new version deployment is not efficient. If any kind of problem occurs (e.g. an unexpected peak of client accesses) it will affect the whole system since we cannot modify or re-scale quickly a single component.

The Microservices architecture appeared as solution to these problems taking advantage of service-oriented architecture and computation. A microservice is an independent process with a well specified functionality that communicates with others through APIs in the network. The principle is that a system can be partitioned into small services, even if they are developed by different teams and with different technologies. This architecture allows adding, removing, and upgrading a specified component without the need to modify the others. Each microservice is deployed independently and if there is a problem

[1] Another big advantage is that life is saner for everyone involved [3]

the service can be retrieved the same way. This architecture is commonly used to structure applications in modern and dynamic environments such as the cloud.

Cloud computing is a computation model that allows the ubiquitous access to a set of resources such as network, storage, and computation capacity on demand [5].

The growth of online applications with high communication and computation requirements and applications submitting a high number of requests to the clouds or generating large amounts of data created the need to lessen network congestion, and service latency. The need to approximate computations and data to locations nearer end users and data sources gave origin to the paradigm of edge computing.

Edge computing [6] takes advantage of technologies such as access points, routers, etc. to support computations and data access at the periphery of the network nearer data sources and clients. The usage of additional resources on the path from the edge to cloud contributed to what is called Fog computing [7]. This paradigm provides local computing resources and network connectivity to centralized services to the end devices, besides the minimization of the request and response times from/to the supported applications.

In this work, we use a broad definition of edge computing with common points to fog computing, in terms of the usage of computational nodes on the path from the periphery of the network to the cloud, but without considering mobile end devices.

The combination of these hybrid cloud/edge/fog computational infra-structures with the microservices architecture, where individual or groups of microservices can be dynamically deployed to those nodes, exponentially creates a big complexity and the necessity for monitorization and automatic management of the overall system (including computational nodes and microservices).

## 1.1 Problem

Migrating the services to the edge infrastructure brings many advantages but it is necessary to deal with the existence of a higher number of deployment places and its heterogeneity. Every offload of data or code has to take into account the nodes' resources, their location, the requests' load, and also the possibility of data and service replication (from the cloud to an edge node). A microservices manager is hence necessary to discover the location of the services simplifying service communication and to support a scalable system with better performance and less error rate. On the other hand, the high number of devices in the edge and the execution of microservices with different capabilities make monitorization and orchestration from the cloud to be less efficient or impossible because of the needed scalability. The increase of the response time to cope with the inevitable changes in the edge infrastructure, either at a computational level or in terms of user accesses' variability, will affect the QoS to end-users. So it is noticeable the major problem to deal, which is the monitorization and the management of microservices as a whole in a cloud and edge context.

In cloud microservices management the complexity depends on the number of microservices in an application as well as the number of users and the expected QoS because it is connected with the scalability of the different services. So having services available on the edge to be used by any kind of device and with low latency requirements, allows the communication between the devices and the services to be faster. The services may even be used to filtrate and lessen the data volume that still has to be moved.

As for monitorization of such a dynamic environments, where nodes and network connections may fail and client requests are highly variable (e.g. popular applications prone to peek accesses), it is necessary to monitor such dynamicity to ensure the QoS requirements. The monitorization system must be adaptable to this system elasticity via a scalable architecture that reacts well on resources' and microservices' dynamism, while continuing providing useful information in real-time in face of the overall adaptation actions (e.g. service migration and replication)

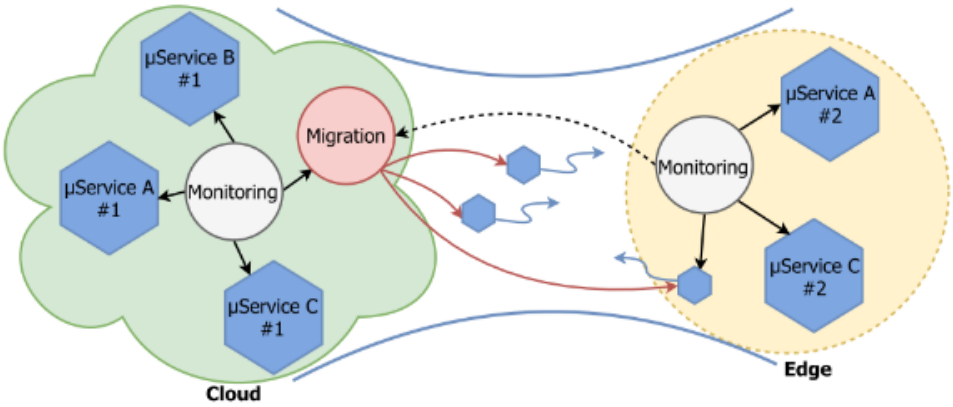


Figure 1.1: Solution proposed to microservices management [8].

Figure 1.1 represents a simplified vision of the management solution [8] that supports the migration and replication of microservices both in cloud and edge nodes, and in order to increase the application's performance. The decision between migrating or not a microservice, or otherwise replicate it and to where, it is based on collected information from both nodes' metrics and microservices' metrics and their dependencies. A migration situation occurs when, for example, a microservice is being heavily accessed from a location far away or this access may generate a high volume of data on transit. The migration helps bring the service closer to clients to avoid network congestion and reduce latency. A replication situation may occur when a replica is overloaded with a high number of accesses and resource usage. The best decision is to balance the load over more replicas that are located at the cloud or edge nearer the highest number of accesses [8].

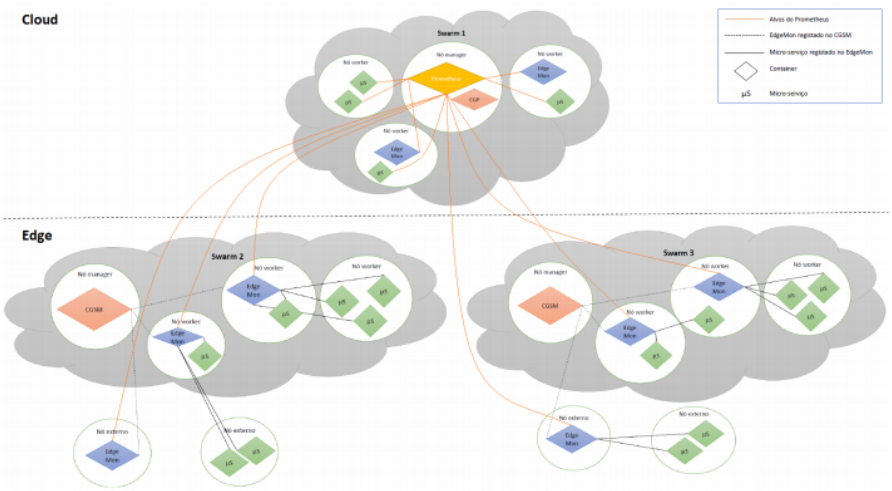


Figure 1.2: Solution proposed to microservices monitorization [9].

Figure, 1.2 illustrates the monitorization system that is composed of a variable number of clusters, each one with a CGSM (System's monitor manager) or CGP (Prometheus' manager) monitoring management components and an undefined number of microservices and EdgeMons. The latter are lighter monitors located at the edge nodes [9]. All components are explained in more detail on chapter 3. The goal is to collect fresh data from microservices and nodes, and to cope with the load upon the monitoring system (the system is able to replicate monitors, whenever necessary).

## 1.2 Objectives

The goal of this thesis is a first step to integrate the monitorization system (Figure 1.2) with the microservices management component (Figure 1.1) to achieve a system with greater autonomy and capable of managing, in an adaptive way, the microservices and the cloud/edge infrastructure nodes.

The management system is guaranteed through metrics collections from the managers to manage the system. Nevertheless it has some down aspects that we aim to minimize. The fact that we cannot have more than one local manager per region to balance the load, change the necessity to specify future actions on the hub manager to a system of alerts, are some of them [10]. One of the most important downside aspects, is the fact that he collects metrics only to manage the system, and it is necessary to start monitor the microservices too. Meaning control the launch, location, metrics collection and if necessary preventive measures (such as migration and replication) to maintain the system. Thinking about the monitorization we realize that with some work the monitorization system [9] can be a good fit at least to help with the monitorization. Of course it also has its downsides, such as the lack of a graphical interface, the necessity of have multiple components that are unnecessary while integrating in this new system. It is needed human intervention to specify where will the monitors and CGSM be deployed, who will they monitor and

the response to overload detection and migration, which explains the urge to have a management component. Besides the last aspect mentioned, one of the biggest aspects was that it has a central monitor component that relies on the communication of every monitor to work well. With these problems this dissertation also aimed to solve this down aspects of this thesis too. Besides the effords to minimize and solve some of the problems of both and integrate both works also aims to reduce the latency on collecting monitoring data from edge nodes and reduce the load of the monitor managers. This integration work connected the light monitor EdgeMon with the local manager in the automatic management system.

## 1.3 Contributions

The main contributions from this work are:

- Extension and adaptation of the already existent architectures to accomplish the integration of the developed tools.
- Functional prototype that implements these functionalities,
- Evaluate the system in several scenarios to show the suitability and performance of its components and mechanisms.

## 1.4 Document organization

The document is organized in the following manner: It starts with an introduction followed by Section 2.1 which refers to Cloud computing, which includes its description, deployment models, virtualization resources, scheduling and cloud resource management, and its downsides. Then 2.2 describes fog computing, its relevance and characteristics (e.g fog node), as well as its challenges. Section 2.3 puts in perspective containers and Virtual machines. In Section 2.4 a view of Managers/Orchestration is given with some examples of platforms. Microservices as well as their purposes risks and companies that implemented it are in Section 2.5.

Chapter 3 is the approach plan, which starts in Section 3.1 where it is delineated the already existing hybrid monitoring system, its components, and functionalities. In Section 3.2 it is done the same as in the previous Section, but relatively to the automatic microservice management system.

Section 3.3 shows the goal of joining the two systems and Section 3.4 shows how to achieve the reunion of these two main system functions. From Section 3.5 to 3.6 it is mentioned the components changed in this thesis. To introduce the way in which the components interact with each other there is Section 3.7.

Chapter 4 represents the implementation and validation chapter. Section 4.1 refers to the technologies, and why they were used. In 4.2 it is broken down the most important aspect in the conception of the thesis, such as code, algorithms, APIs, and some configurations.

The Chapter 5 defines the demonstration validation and experimental evaluation. Divided in two Sections, in Section 5.1 it is done different scenarios with their procedures and results it is also presented a interpretation. In the last part, Section 5.2, tests are made to verify the behavior and use of resources by the monitor.

Conclusion, Chapter 6 puts everything into perspective, the final work, the objective, and the theory behind everything. Finishes with the future work which is to enumerate the next steps and little aspects that can be improved in order to reach a more complete and complex system.

# Chapter 2

# State of Art

## 2.1 Cloud Computing

Network-centric computing refers to the location of computing resources in data centers instead of computation being based on local nodes. Similarly, network-centric content refers to the allocation/storing of data in data centers instead of locally [11]. Advancements in technology support the fact that computations done in server farms become more efficient. [12] A server farm is a set of servers interconnected and stored in the same physical facility, and its main propose is to combine the computing power of multiple servers by executing one or more tasks. [13]
Understandably, the biggest advantage of the network-centric content is the possibility to access and share information, in a local independent manner as long as they are online. By spreading the server farms all over the globe at different geographic places, it provides benefits in terms of availability and redundancy since if there is some failure in one of the facilities the other ones can step in.

There are two computing models: - Utility computing in which hardware and software are in data centers and their users are charged depending on their computation, memory, and communication. -Cloud computing that, according to NIST, is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [5]

Cloud computing is a technical and social reality and an emerging technology [14] because of its attributes such as:

1. Offers the opportunity to dynamically obtain computing resources and support different workloads while giving the user the illusion of unlimited resources. These kind of features is guaranteed through infrastructures maintained by the providers and they do not require human interaction with each service provider [5, 11].

2. Cloud systems have the control and can optimize resource usage for supporting the services making it measurable. By monitoring and controlling the resources usage allows charging users only for the resources they use [5, 11].

3. Computing service capabilities are available globally through the network by heterogeneous client platforms such as tablets, phones or laptops. Providers also assure security and maintenance [5, 11].

4. The economy of scale allows the providers to use specialization and centralization to organize processes becoming more efficient. Most enterprise applications today have been designed for low performance/high-cost hardware and low bandwidth network making necessary on-premise software and hardware resulting in poor hardware utilization and high TCO (Total Cost of Ownership). A reason for high TCO relates to the expertise needed from the providers to the end-user organization. With the high cost of this re-skilling process and the fast rate of change, the software gets out-of-date. Ubiquitous network and scalable computing platforms overcome these enterprise models. Concentrating the resources with the IT providers results in an increased specialization [11, 15].

5. Using resource multiplexing makes cloud computing cost-effective and by reducing the price for the providers we reduce it to the users too. In cloud computing, multiplexing is done through virtualization technology which separates the application from the physical resources that are needed. The concept of virtual machine encapsulates an application and its associated features such as adaptable resource provisioning and migration increase dynamic resource provisioning capabilities and efficiency of resource usage [16].

Cloud computation is Service Model orientated which means the resources in a hardware/software level are available as services that users can request if and when they want [17]. Clouds offers services that are in three different [5, 11, 17]:

- **Software as a Service (SaaS).** It is given to the user the capability to execute applications over the internet through a web browser or a program interface. A big example of a SaaS provider is Google.

- **Platform as a Service (PaaS).** Offers capabilities to the user to create and deploy applications using programming languages, libraries, services and tools supported by the provider. A big example of a PaaS provider is Google.

- **Infrastructure as a Service (IaaS).** It is given to the user the capability to provision infrastructural resources where the consumer can deploy and run the software, refers usually to VM's. A big example of an IaaS provider is AMAZON - AWS, offers infrastructures with servers to realize computation and storage.

### 2.1.1 Deployment Model

There are different types of clouds and when an enterprise is considering changing its applications into a cloud environment they have to see each benefit and drawbacks:

- Private Cloud. Infrastructure is open exclusively for one organization and it is managed by the organization itself or by an external one. The exclusivity increases control over performance, reliability, and security [5, 17].
- Public Cloud. Infrastructure is provisioned for general public usage, it may be managed by any kind of organization but it is located at the cloud provider premises [5, 17].
- Hybrid Cloud. Is a combination of private and public cloud trying to address the limitations of the both of them. Part of the services run in the private cloud and the remaining in the public clouds. It allows a bigger control and security over application data when comparing to public clouds [5, 17].
- Community Cloud. Provisioned for an exclusive community of consumers from organizations that have the same responsibilities, for example, security requirements, policy, mission. It can be maintained or owned by organizations from the community or external ones, and it may be on or off premises [5, 17].

### 2.1.2 Virtualization resources

Cloud computing is directly related to distributed and parallel computation. This is due to the fact that applications are based on a client-server model in which a client runs in a user's machine and the computations are done in the cloud. Because of the data-intensive applications we use a large number of instances that run simultaneously.

Parallel computation is the concept of divide complex problems in smaller ones and concurrently solves them decreasing the time to obtain a solution.

A distributed system corresponds to a set of computers that by being connected thought the network and distribution software (middleware) can coordinate their activities and share system resources giving the user the idea that is only one computer facility.

Concurrency helps to realize multiple activities in simultaneously reducing the data-intensive problem execution time. A lot of time a system that uses concurrency explores virtualization of resources such as processor and memory. The main objective is to:

- Hide latency and boost performance;

- Avoids limitations imposed by physical resources;
- Increases reliability;

With the constant growth of users and application diversity and scale of a system, resource management becomes complex. Adding the fact that they deal with heterogeneous systems and hardware, need to redistribute the load. Because installing a standard OS on systems and rely on conventional OS techniques is challenging for both providers and users the alternative is virtualization. Virtualization simplifies some of the resource management tasks, for example, because of its capability to save the state of a VM, which run under a VM monitor (VMM), it is possible to migrate to another server for balance load proposes. We can distinguish two different kinds of virtualization:

- **Full Virtualization**. It is achievable when the VMM provides a perfect replica of the physical hardware when it comes to hardware abstraction. OS running on the hardware, in this case, will run without any modifications. Examples of this kind of usage are in VMware VMM.
- **Para-virtualization**. This kind of virtualization demands that the guest OS must be modified to use only instructions that can be virtualized and run under VMM. Examples of this kind of virtualization are Xen [18] and Denali [19]. Para-virtualization is used in three cases:
  - Features of the hardware cannot be virtualized;
  - Improve performance;
  - Simplify interface;

### 2.1.3 Scheduling and Cloud Resource Management

#### 2.1.3.1 Resource management

Cloud is a complex system with a lot of shared resources and unpredictability in requests and affection of external uncontrollable events are two events that may occur. Cloud resource management is considered one of the principal functions of man-made systems and requires complex policies and decisions for its optimization besides it affects the three basic system evaluation criteria: functionality, performance, and cost.

For the different delivery models (IaaS, PaaS, and SaaS) there are different strategies used, but in all of them, the service providers are confronted with large loads that challenge cloud elasticity. When a spike is predicted resources are provided but when it is unpredictable we need a mechanism - Auto Scaling. It can be used in these unplanned spikes when there is a set of resources that can be released or allocated. Or when there is a monitoring system allowing a controlled loop to decide as it happens to reallocate resources.

To diminish the human intervention policies are made because of system dimension, service requests, load unpredictability and the number of users. There are five groups of policies [11]:

1. Admission control.
2. Capacity allocation.
3. Load balancing.
4. Energy optimization.
5. QoS guarantees.

(1) have has the goal to prevent the system from accepting workloads that may prevent the conclusion of an already in-execution or contracted work. (2) is used when resource allocation is needed for individual instances. (3) and (4) can be done locally and they correlate and affect the cost of providing the service. A critical objective of cloud computing is to minimize the cost of service providing and the energy consumption that is why load balancing is really important. The goal is to use the smallest number of servers while the others stay in standby which is a state with low energy consumption. (5) is one of the most difficult and critical aspects to address and in the future of cloud computing.

#### 2.1.3.2 Scheduling

Scheduling is responsible for resource multiplexing/sharing at several levels, it is a critical part of cloud management. Multiple threads can run in a single application, multiple applications can run in a single VM and multiple VM's can share a server. The CPU scheduling supports the virtualization of a processor where each thread acts like a virtual processor. A communication link can be multiplexed between multiple virtual channels, one in each flow [11].

The objectives of scheduling algorithms for a batch system are to maximize the transfer rates which is the number of jobs per time unit and minimize the answer time which is the time between the work submission and its conclusion. A preemptive schedule is when it can interrupt a task of less priority and execute one with higher priority. Scheduling algorithms must be efficient, fair and starvation free. Two dimensions of resource management must not be forgotten by a scheduling policy: (1) when are the access to resources granted and (2) the number of resources allocated [11].

### 2.1.4 Downsides of Cloud Computing

There are some ethical issues associated with cloud computing:

1. The fact that it relies on multiple places means that its maintenance is made by different organizations;

2. Control is done by multiple organizations. Giving this control opens the possibility of occurring non-authorized access, failure of the infrastructures, corrupt the data or service unavailability. Because of the multiple organizations-based type management, whenever any of these occurs it becomes difficult to discover the source or entity responsible [11].

3. Multiple services inter-operate in the network. Cloud services have complex structures and it makes difficult to determine who to blame in case something happens. Many entities can be part of an action in a complex chain of events or systems [11].

When referring to a big degree of control it means to provide the option to measure alternatives, arrange priorities and act in the best interest when facing incidents. Overall, when users have control over the processes and equipment involved they tend to feel more comfortable. To choose between internal solution and a cloud-based implementation the level of trust is an important step as the assessment of the risk associated with each model. The level of trust is proportional to the direct control the organization is able to exert on the external service provider [5, 20].

#### 2.1.4.1 Privacy

Privacy is defined in various ways, influenced by culture, some cultures favor privacy, others underline community [11]. The way that information is processed or who is responsible for is a major decision because cloud services complex structure makes it impossible to conclude who is responsible in case something bad happens (Accountability). In cloud computing two terms conflict – privacy versus accountability – thus it is important to know at all times what logs are stored and who has access to them.

#### 2.1.4.2 Cloud Storage and Vendor Lock-in

Also, there is a fear of user dependency on the cloud service also known as vendor lock-in. When organizations or individuals are dependent of only one cloud service provider it adds some risks. It brings the possibility of non-availability for unknown periods that may cause large-scale impacts in organizations such as potential permanent loss of data and substantial increase of prices charged by the provider in terms of memory, computation cycles or bandwidth usage. The alternative would be to change the provider. However, it increases the prices significantly and takes a large period. This limitation is called, lock-in [11].

## 2.2 Motivation and What is Fog computing?

The data volumes generated by IoT (Internet of things) environments is reaching a point of overwhelming storage systems and analytics applications [21]. Cloud computing helps

addressing the storage and processing problems associated with IoT through scalable on-demand solutions. However, cloud data centers are few in number and located in remote locations (e.g. colder regions to reduce energy costs) what results on delays on accessing remote data and processing.

Additionally, it is not efficient to be trading with the cloud a big data volume for storage and processing as it would saturate the network bandwidth besides not being scalable [21].

There are some cases worth mentioning because of their high data volume. Analysis of healthcare-related IoT application flows up to more than 20,000 tuples per second of data and with more than 28 million users [22].

With this continued growth of online applications, communication, and computation [23], some issues need to be addressed. The need to lessen network congestion, accelerate analysis and making decisions. Edge computing is an option as it supports the idea to take advantage of technologies that allow computation to be performed at the edge so that computing can be done nearer of data sources [6]. Basically, edge computing would be, in very broad terms, performing computations outside boundaries of data centers [24], without any notion of cloud services whatsoever [25].

Eventually, it turns out to be inefficient because it can not handle multiple applications and if so, they would start competing for limited resources resulting in an increased processing latency and resource contention. These problems are overcome by the unification of cloud resources with edge devices, calling it fog computing [21].

The distinction of these is done thru the fact that edge computing does computation at the edge of the network without a notion of cloud services. Depending on the source, fog computing can be the same as Edge Computing or considered a blend between cloud, edge, and intermediary nodes, this being a small or medium data center [25].

Fog computing then integrates edge devices and cloud resources with the goal of overcome these limitations. Prevents resource contention at the edge through cloud resources and coordinates the use of geographically distributed edge devices. [21]. Facilitates management and programming of computation, networking, and storage between data centers and end devices. The National Institute of Standards and Technology (NIST) defines fog computing as a "layered model for enabling ubiquitous access to a shared continuum of scalable computing resources. The model facilitates the deployment of distributed, latency-aware applications and services, and consists of fog nodes (physical or virtual), residing between smart end-devices and centralized (cloud) services" [7].

So it is understandable that fog computing besides the minimization of the request and response time from/to supported applications have the capability to provide to the end devices local computing resources and network connectivity to centralized services.

### 2.2.1 Fog Computing Characteristics

#### 2.2.1.1 Fog node

To understand the characteristics that characterize fog computing it needs to comprehend what is fog nodes.

Fog node is the principal component, and they may be physical (Switches, routers, servers, etc.) or virtual (virtualized switches, VM's, etc.). They are coupled with smart end devices or access networks providing computing resources. These kinds of nodes also [7]:

1. Aware of its geographical distribution and logical location in a cluster context;
2. Provide some data management and communication service between the network's edge layer (end-devices) and fog computing service or cloud computing resources (centralized);
3. Operate in a centralized or decentralized manner to deploy a given fog computing capability;
4. Be configured as stand-alone fog nodes communicating among them to complete the service or be configured in a cluster form to provide horizontal scalability over different geolocations, using mirroring or extension mechanisms.

**Fog Node Attributes**

Fog nodes need to support one or more of these capabilities:

1. Nodes operates in an independent form making local decisions - *Autonomy*;
2. Nodes can come in different form factors and deploy in a variety of environments - *Heterogeneity*;
3. Support hierarchical structures, where sometimes have different layers providing different subsets of service functions while working together as a continuum - *Hierarchical clustering*
4. Besides performing most routine operations automatically node are managed and if - *Manageability*
5. Network operators, domain experts, equipment providers or end can program fog node in multiple levels.- *Programmability*

#### 2.2.1.2 Essential Characteristics

Fog computing deals with IoT data in a local manner using clients or edge devices near users to make a large amount of storage, control, communication, configuration, and management. The proximity to sensors by edge devices benefits this approach while

leveraging the on-demand scalability of cloud resources and involving components of data-processing or analytical applications to run in a distributed cloud and edge devices. It helps with the management and programming of computing, storage services, and networking between end devices and data centers [7]. Some of the characteristics that make fog computing different from the other computing paradigms are:

a) *Low latency and contextual location awareness* Offers the lowest-possible latency because the fog node can know their logical location in the context of the entire system and latency cost. Most of the times there are co-located with smart end-devices, analysis and response data by these devices is faster than a centralized cloud service or data center could give.

b) *Geographical distribution* Geographic identifiable distributed deployments are very useful and efficient for instance high-quality streaming service to moving vehicles is delivered through proxies and access points geographically situated along tracks and highways.

c) *Heterogeneity* Process and collects data of different form factors gotten through multiple types of network communication capabilities.

d) *Interoperability and federation* Capability to interoperate and services are federated across domains.

e) *Real-time interactions* Instead of batch processing, they involve real-time interactions.

f) *Scalability and agility of federated, fog-node clusters* Adaptive in nature, at cluster or cluster-of-clusters level, supporting elastic compute, resource pooling, data-load changes, and network condition variations, to list a few of the supported adaptive functions.

#### 2.2.1.3 Architecture

Fog computing can be dismantled in five layers. The first one, in the bottom-most, lays the end devices which are sensors, also the edge devices and gateways. This layer is included applications that can be installed in the sensors to enhance their functionalities. The second layer is the network, used to communicate among these elements and the cloud. For supporting the resource management and process of IoT tasks reaching the cloud is the third layer in which are set up the cloud services and resources. The top of the cloud layer is occupied by the software for management resources. It manages the infrastructure and enables QoS of fog computing applications. The fifth and last layer is saved for the applications that leverage fog computing to distribute intelligent and innovative applications for end-users [21].

As was said the fourth layer is to the resource management software. Through middleware like services implementation, applications optimize the use of cloud and fog

resources. These services' objective is to decrease the cost of cloud usage while reaching acceptable levels of latency by pushing workload to fog nodes [4, 21]. These services are:

- *Performance Prediction:* Knowledge- Base service information, estimates the performance of the available cloud resources and then transmits the information to resource provisioning service.
- *Resource Provisioning:* When performance it is not at is best or the number of tasks and flows are big it decides the number of resources to be provisioned.
- *Raw-Data Management:* With direct access to the data sources, this service provides views from the data, for the other services. These views can be achieved by simply querying through SQL, NoSQL REST APIs or with more complex processing such as MapReduce.
- *Monitoring:* Keeps a performance and status of applications and services track, and supplies this information to other services when required.
- *Profiling:* From the information obtained through knowledge base and monitoring.
- *Resource Provisioning:* For hosting applications Cloud, Fog, and network resources are acquired by this service. Deciding the number of resources is made with the information provided by other services and user requirements on latency, as well as the credentials managed by the security service.
- *Security:* Responsible for authentication, authorization and all of the cryptography. It is possible to build fog stacks and applications without the use of all of these elements, they are reference elements.

### 2.2.2 Challenges

Understanding fog computing's potential it is noticeable its challenges:

#### 2.2.2.1 Programming models and architectures

Offloading is a technique in the mobile computing domain most used when referring to offload workloads to the cloud [26]. Fog environments require the capability to easily add and remove dynamically the resources because of the frequent participation in the network of mobile devices. The most stream and data processing frameworks are based on static configurations so they do not provide scalability and flexibility as much as needed [4, 21]. Since offloading to the cloud is not always possible or the best choice, alternatives where presented, CloudAware an adaptive Mobile Edge Computing (MEC) programming framework [27]. It offloads workload to edge devices, facilitating elastic and scalable edge-based mobile application developments [27].

#### 2.2.2.2 Security, reliability and fault tolerance

Having distributed resources, multiple users and service providers fault tolerance, security, and reliability is a major key challenge. The challenge comes from the designs and implements techniques of authorization and authentication that works with multiple fog nodes with different computing capacities. Potential solutions for this problem are Trust execution environment (TEE) and Public Key infrastructure [28]. It is important for users on fog deployments plan for the failure of networks, service platforms, individual sensors or even applications [4, 21].

#### 2.2.2.3 Energy consumption

Fog environments consist of many nodes, the computation is distributed and can be less energy efficient than the centralized cloud model of computation. The reduction of energy consumption is a major and important challenge [4, 21]. In the study of trade-off between delay in fog computing systems and power consumption both were modeled and the problem of allocating workloads between the fog and cloud was formalized [29].

#### 2.2.2.4 Resource Management

When talking about fog devices they are often network devices that have additional computing power and storage, unfortunately, it is very hard to match a server or the cloud resource capacity. Ergo the importance of having good management of resources for the efficient operation of a fog computing environment. Resource management systems must also dynamically establish which analytic tasks should be pushed to which edge or cloud resource for latency minimization. Was presented a resource management model that performs efficient and fair management of resources for IoT deployments. This resource-management framework has the ability to predict resource usage of users and according to his behavior and the probability of using it in the future, allocates resources. Predicting gives us fairness and efficiency in the consumed resources [30].

## 2.3 Containers and VMs

Nowadays the quantity of virtualized data centers is increasing through applications hosted on one or more virtual machines which are then mapped to physical servers.
This technology provides a versatile physical resource allocation to virtualized applications where the workload is adjusted through the dynamic variations of the number of resources to each application and also, the mapping of virtual to physical resources. Virtualization by: (I) Enabling multi-tenancy allows multiple instances of virtualized applications to share a physical server; (II) Multi-tenancy also allows data centers to pack and consolidate applications within a smaller set of servers reducing the operating costs; (III) Simplifies

replication and scaling of applications [31].
In server virtualization there are two kinds of technologies (demonstrated in figure 2.1[1]):

- Hardware-level virtualization implicates running a hypervisor to virtualize the server's resources throughout multiple virtual machines. Each hardware VM runs its OS and applications.
- Operating system-level virtualization virtualizes the resources at the operating system-level. The containers are generated with this virtualization which has enclosed the OS processes as well as their dependencies. They are controlled by the underlying OS kernel [31].

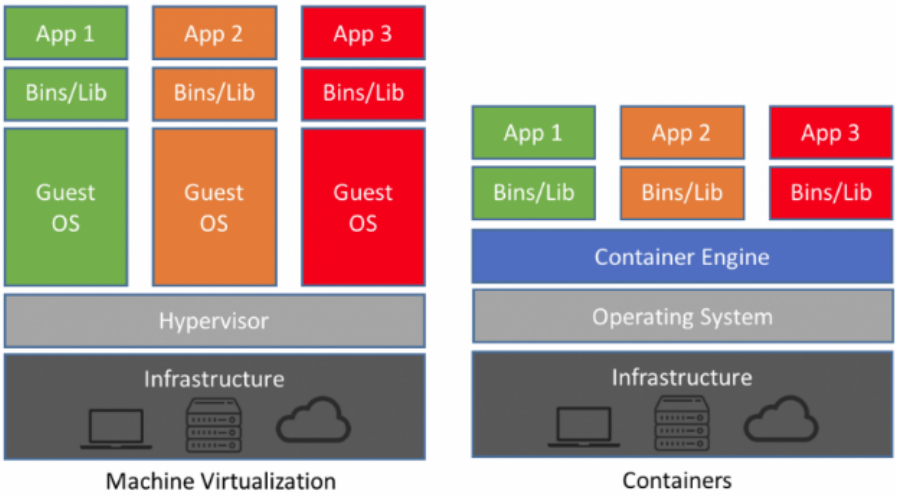


Figure 2.1: Containers and Virtual Machines illustrations.

[1]https://blog.netapp.com/blogs/containers-vs-vms/

Discussing containers and VMs is important to know that they differ within the technology they use in virtualization, notable especially in their performance and manageability.

### 2.3.1 Containers

A container also referred to as operating system and application virtualization [32], is a software environment that packages code and every dependency (libraries dependencies, binaries, and configuration files). Packaging allows the application or application component, microservice, to run quickly and reliably from one computing environment to a different one [33, 34].

This innovative technology does not emulate physical hardware and hence it is lightweight and has less overhead. It grants the execution of multiple containers (applications) in one machine, without having conflicts between them [32, 34, 35].

Containers have the ability not only to function when necessary but to upgrade and dispatch new versions of applications and/of a microservice, granting a higher level of abstraction in terms of the life cycle management [32, 35]. Using containers, an application or microservice can be executed on any platform if it is running a container engine. In other words, it means that regardless of the infrastructure, containerized software will always run the same [33, 35].

Docker [36] , CoreOS Rocket (rkt)[2] and LinuX Containers (LXC)[3] are examples of container implementations. Despite the other container engines, docker and rocket are the foremost used ones as a result of their enhanced features [37].
Executing multiple containers at the same time generated the need for mechanisms and rules for their efficient management. There are already some tools that allow this orchestration of containers but without fully autonomic functionalities throughout the various possible scenarios. Kubernetes, in a 2018 study, was in the lead for the most used orchestrator with 51%[4]. The second position was Docker Swarm with a total of 11%.

### 2.3.2 Virtual Machines

In 1974, Gerald Popek (UCLA) e Robert Goldberg (Harvard) defined a virtual machine: "A virtual machine is seen as a efficient and isolated replica of a real machine. This abstraction is built by a virtual machine monitor, VMM" [38].

Virtual Machines (VM) are guests. They are created within an environment known as host in witch is feasible to own multiple Virtual machines at the same time. Despite that VMs offer identical capabilities as physical computers they are only software computers as they are only computer files running on a physical computer behaving like a separate one. Each VM can run a unique operating system and has its own libraries, binaries, and applications. A hypervisor or VM monitor is software, firmware, or hardware that creates

[2] https://coreos.com/rkt/
[3] https://linuxcontainers.org/lxc/
[4] https://sysdig.com/blog/2018-docker-usage-report/

and runs Virtual Machines. To work in a proper and efficient manner the hypervisor has to attend some basic requirements. Must provide an execution environment identical to the real machine from a logical point of view. Programs that are being executed in a virtual machine can only have light performance degradation. Finally, the hypervisor must have absolute control of the resources of the real system (host) [38].

Virtual machines[5] are specially used for two reasons. (I) To perform risky tasks because they are sandboxed from the rest of the system which means any software within the VM cannot tamper the host. (II) Server Virtualization is called when a physical server is split in multiple independent virtual servers through an application that each one can run its operating system.

### 2.3.3 Containers VS Virtual Machines

**Containers** have a bare metal[6] performance but in multi-tenant scenarios, there is some interference. They share the underlying OS kernel and consequently create a lack of isolation, which implies that if any problem happens it may continue or infect the host.

While **Virtual Machines** have limited resources, the containers allow soft limits that are extremely helpful in scenarios where the limit of memory allocated is exceeded (overcommitted). This happens as a result of the possibility to recycle resources allocated to other containers that are not utilized. To overcome the lack of isolation and to enhance efficiency in resource sharing, running containers inside VMs solves this problem thus making it a viable architecture. Besides containers offer bare-metal performance, low footprint and, integration into the software development process, the isolation properties in VMs make hybrid virtualization approaches more appealing for research [31].

Although container-based services have better performance other than VM-based services, some studies revealed that using Amazon AWS EC2 (EC2 container service) has a worse performance when compared with services deployed in Amazon EC2 VMs [39]. This surprise on the results is due to the fact that container-based services in Amazon cloud are containers on top of EC2 VMs and not on bare-metal physical hosts. The performance in this study was measured in terms of response time, CPU usage, and other performance metrics in various deployment scenarios [39].

## 2.4 Managers/Orchestration

To make possible the execution of multiple containers it is necessary mechanisms to manage efficiently and effectively. Orchestration of containers involves a set of operations arranged by service providers, cloud, and the application's owner to make the deploy, monitoring, and control dynamically the resource configurations to guarantee a certain QoS. In these mechanisms are included the initial deployment of the containers but also

[5] https://www.vmware.com/topics/glossary/content/virtual-machine
[6] https://www.rackspace.com/library/what-is-a-bare-metal-server

their execution time management. [32] The decisions taken by the orchestration must take into account the environment dynamism and complexity, and adjust in real-time. For that its required to exist autonomic mechanisms that make decisions without human intervention so factors like resource management in execution time, containers monitoring, etc, are important factors.

### 2.4.1 Orchestration Platforms

The existent tools to orchestrate containers still does not give autonomic functionalities that adapt to various scenarios.

Kubernetes, one of the most used tools of container orchestration allows the management of a cluster of nodes. Substitution of a failed container, auto-scaling containers based on CPU utilization rates are examples of Kubernetes functionalities. [32]

Another platform used is the Docker Swarm, which is a tool integrated with Docker and allows the creation of a cluster of nodes to execute containers. It is possible to specify the number of replicas of each container, change it in execution time and, if there is a failure in a container a new one is initialized[7].

These are two of the most used tools that offer containers management but they are yet primitive in terms of functionalities in the adaptive mechanisms when the number of accesses varies nor the identification of the best location to execute containers based on the origin of the accesses. Additionally, these tools are not taken in account when it comes to heterogeneous infrastructures or even less in fog/edge cases.

## 2.5 Micro-Services

To better understand microservices it is easier to start with the monolithic style. A monolithic application[8] is built as a single unit, organized in modules that are directly dependent of the application. Imagine we have an enterprise application, most of the time they are composed of three parts: a client-side application (user interface made by HTML and javascript), a database and a server-side application. The server-side will be a monolith application: a single logical executable, which means any kind of change implies building and deploying a whole new version of the server-side application, non scalable. [40]

This is one of the biggest disadvantages of using this architecture style and align with others such as Inflexible (cannot use different technologies), Unreliable (if one feature does not work the whole system fails) or with slow development [41]. The microservices style started to get more interesting, the difference between both architectures is shown in figures 2.2 and 2.3.

[7] https://sysdig.com/blog/2018-docker-usage-report/
[8] https://martinfowler.com/articles/microservices.html

| Category | Monolithic architecture | Microservices architecture |
|---|---|---|
| Code | A single code base for the entire application. | Multiple code bases. Each microservice has its own code base. |
| Understandability | Often confusing and hard to maintain. | Much better readability and much easier to maintain. |
| Deployment | Complex deployments with maintenance windows and scheduled downtimes. | Simple deployment as each microservice can be deployed individually, with minimal if not zero downtime. |
| Language | Typically entirely developed in one programming language. | Each microservice can be developed in a different programming language. |
| Scaling | Requires you to scale the entire application even though bottlenecks are localized. | Enables you to scale bottle-necked services without scaling the entire application. |

Figure 2.2: Comparing monolithic and microservices architectures [42].

In a monolithic system to fight against the large code base often is used abstractions and modules to get more cohesion. Cohesion is described as a "drive to have related code grouped together". It is reinforced by Robert C. Martin's definition of Single Responsibility Principle, that says "Gather together those things that change from the same reason, and separate those things that change for different reasons.", which is an important point a view when referring to microservices as they follow this approach. [43]

The principle is that a software system is decomposed in small services that alone can be developed and deployed. They are most of all loosely coupled which means that besides being able to deploy a single microservice on its own there is no need to coordinate with other microservices [42]. For each service, there is its own code repository. These small services can also be developed by different teams with different technologies and different programming languages. Making microservices loosely coupled allows the system to still work even if some of the services are down thus making the system more resilient [40].

There is more than one dimension when evaluating microservice architecture scalability. One of the first dimension that comes to mind is the load. When referring to the load dimension it means that it is possible to add resources in order to keep the performance whilst increasing the workload. In a functional dimension becomes easy to join new functions and qualities at any stage and that allows this architecture to scale to the best version[9]. There are many ways to design this type of architecture but if it is poorly designed it can have a negative impact on the system performance [44]. When the requirement for service invocation requires communication with a container or host through a network link as a result of data transfer rates and the potential failure of the communication link, there is a reduction in performance [45].

Microservices architecture is used in complex systems when they aim to have high reliability, performance, and scalability. Figure 2.3 ilustrate the diference between the two types of architectures [10].

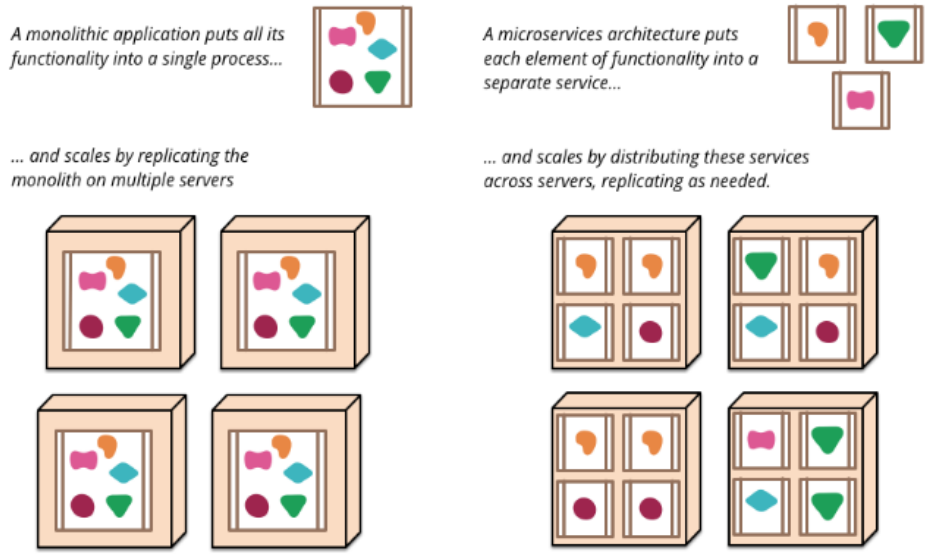


Figure 2.3: Monoliths and Microservices.

[9]https://specify.io/concepts/microservices
[10]https://martinfowler.com/articles/microservices.html

### 2.5.1 Purposes

#### Minimize Costs of Change

Microservices are intended to scale according to new requirements.

The single responsibility principle, explained earlier in section 2.5, allows the minimization of the cost of new or modified requirements, which is one of the principal goals of the microservice architecture style.

There are some rules and recommendations[11] to take into account:

- Different functionalities must be at different services.
- It is beneficial to use small codebases to be easier to extend and modify. Simpler codes also reduce the error risk when introducing new code and make IDE loads faster.
- A service cannot assume that another service is written in a specific language in other words the protocol used in communications should be technology agnostic.
- Services must use its official API while communicating with each other and to do so an inter-service communication must be through a network connection. One service can not access the data that has been written by another service. It is a must that each service has its own logical database schema.
- Centralized orchestration[12] implies the control of all interactions and elements, while in a choreography microservices are observing the environment and act on events. So while dealing with microservices its is preferable to use choreography over orchestration. In orchestration, the interactions occur across the network thus invocations take longer and are dependent on the network conditions.

Following these rules and recommendations, it is prevented that code goes to the wrong place and ensure new modifications are enabled (changeability).

#### Encourage Generalization, Replaceability, and Reuse

Above is advised that different responsibilities should be placed in different services. Similar to this rule and to achieve the best reusability for a single service, it is necessary to build smaller services that do one thing well. Normally when talking about the re-utilization of an already existing service the common argument that is presented is the complexity, because the solution does more than what is needed. Making it tempting to build a new and simpler solution instead of learning about the existing one.

The question about microservice architecture should not be how big is the service but which functionalities should be grouped in the same service[13].

[11] https://specify.io/concepts/microservices
[12] https://solace.com/blog/microservices-choreography-vs-orchestration/
[13] https://specify.io/concepts/microservices

#### Increase Operations Efficiency

In operations efficiency, we can see the advantage of using a microservice architecture, when it comes to scaling. With this kind of architecture, we only scale the part that really has performance issues, and when it is under high load the hardware resources are cheaper. Scaling a monolith application horizontally (adding more machines), it is necessary to install the whole monolith application multiple times. In cases that only a part becomes a bottleneck, it is a big waste in terms of hardware resources.

### 2.5.2 Risks

#### Increase Operations Complexity Risk

In microservice type architecture is necessary to operate more services and without the help of automatization mechanisms, it is very difficult to manage and deploy hundreds of services. It is really hard to debug services that intercommunicate with other services. To mitigate this disadvantage organizations use containers (for example, Docker) and PaaS technologies (cloud application platforms)[14].

#### Distribution

A microservice application can be considered a distributed system because each service is executed in an independent way and be in different machines.

The inconvenience appears when it comes to remote connections, they are slow, and if it has a chain of remote calls the answer time will increase. An alternative to overcome this disadvantage is to do parallel asynchronous calls (independent calls) thus getting influenced only by the slowest call.

### 2.5.3 Companies that implement the microservice architecture

Companies usually start by using a monolithic architecture cause it is faster to build up a monolith and start the business. With the system growth, the increasing complexity of the code and architecture, and the need for more developers, the system loses flexibility agility and speed affecting directly the performance.

Microservices interact with each other through API gateways, and they can be deployed, developed, and scaled separately. API gateway[15] is an entry point for all clients, it handles requests and can expose different API for each client. It means that, for example, if an online store needs any change in payment, the change and re-deploy are done to only a part of the system. This allows lower costs, a quicker release cycle, and motivate innovations. [46]

[14] https://specify.io/concepts/microservices
[15] https://microservices.io/patterns/apigateway.html

#### 2.5.3.1 Case 1 - Amazon

While monolith architecture was still the way of developing IT Systems, Amazon managed to be one of the first where microservices transformed the whole business, achieving a major success.

In a monolith architecture because of the fact that all the services and components were so tightly coupled the developers could not deploy changes in a quick and smooth manner. This translated in weeks before users could use heavy code changes. Through microservices, Amazon[16] simplified the pipeline and break down structures into single applications. This break down allowed the developers to understand and rebuild the bottlenecks, and divide particular services to small dedicated teams. Thus a service-oriented architecture. [46]

#### 2.5.3.2 Case 2 - Netflix

This took a while to happen, one of the earliest adopters of microservices the transition process was made in steps. With more than 500 microservices and API gateways, Netflix application handles more than 2 billion API edge requests per day.

The first step made was to move movie encoding and non-customer facing applications. This was followed by decoupled customer-facing applications such as sign up, device selection and configuration, movie, and series selection. [46, 47]

#### 2.5.3.3 Case 3 - Uber

Back then and while Uber was just in a single city the solution was based on a monolith architecture. Soon it started causing problems in scalability and continuous integration, forcing Uber to transform its global IT system into microservices. The new architecture allowed to introduce an API Gateway and independent services with individual functions and that can be deployed and scaled separately [46, 48].

[16] https://thenewstack.io/led-amazon-microservices-architecture/

# Chapter 3

# Proposed Solution

When using execution environment layers such as cloud, edge, and different devices, it is obvious that we have a complex hybrid environment in our hands with the consequences in terms of its difficult management. Cloud computation allows us to store and process large amounts of data, while the edge nodes are dispersed and heterogeneous, where aspects such as scale and latency are very important. The result may be an overhead on existing cloud monitors, which are fundamental to inspect the system's state. To reduce the monitoring data transferred to the cloud to be processed, in our case, by Prometheus [49] and produce alerts on the presence of important events, an adaptable light monitor was developed to be deployed on edge nodes. This was the work of two previous MSc thesis at DI/FCT/UNL [9, 50]. The already developed prototype allows the monitoring of hybrid environments combining heterogeneous resources of both cloud and edge nodes, with the main goal of supporting an efficient, agile, and adaptable monitorization of microservices applications that may be (partially) deployed on edge nodes. The resulting lighter monitoring system additionally generates notifications with low latency whenever pre-defined urgent situations appear in the system. The monitoring system, nevertheless, still lack some properties that are considered important to its correct and efficient operation. For instance, there is still room to extend the previews works in areas such as security, adequate client interface, aggregation functions, but most importantly, its management and adaptability like they are provided by the microservices management component, also developed in another MSc thesis at DI/FCT/UNL [8]. This work described was further extended as presented in MSc thesis "Gestão Dinâmica de Micro-serviços na Cloud/Edge" [10].

The following sections are organized to first explained the existent projects. The existent hybrid monitoring system section 3.1 and the existent automatic microservice management system, section 3.2. Secondly we proceed to explain inside each one of the

mencioned sections the previous works components which we are going to take advantage of. Section 3.3 refers to the goal of joining these two systems. The architecture of the improved system comes in section 3.4. Following this section, sections 3.5 and 3.6 explain each new improved component. In section 3.7 it is introduced how the components interact between each other.

## 3.1 The Existent Hybrid Monitoring System - Previous Work

The general problem underlying the developed monitoring system was that, at that time, the existing monitoring solutions to support microservices applications were only based on cloud nodes or offered heavy solutions not compatible with restricted resources typical of edge nodes.

The first work [50] developed a light edge monitor named EdgeMon that, together with the Prometheus located at the cloud, allowed 1) collecting metrics from edge (and cloud) nodes and microservices deployed in containers, 2) filtering values to be pulled from Prometheus at a lower rate, but also 3) maintaining the possibility to generate instant notifications upon relevant events.

The second work [9] extended the EdgeMon capabilities and created new components to support the monitors' adaptability, including supporting very restricted computational resources. Figure 1.2 presented a snapshot of a particular instantiation of the hybrid monitoring system.

### 3.1.1 Edgemon

The EdgeMon is deployed at an edge node or at a cluster whenever it is necessary to collect their metrics as well as from the microservices located there. A set of pre-defined metrics are collected both via push and pull methods and the monitor is able to support the dynamic placement of microservices in the node. In the case of nodes with restricted capabilities, the monitoring system deploys even simpler components, namely the cScraper with the Node Exporter to obtain the containers' metrics. The cScraper generates metrics over resource usage and performance characteristics of the containers executed in that same host.

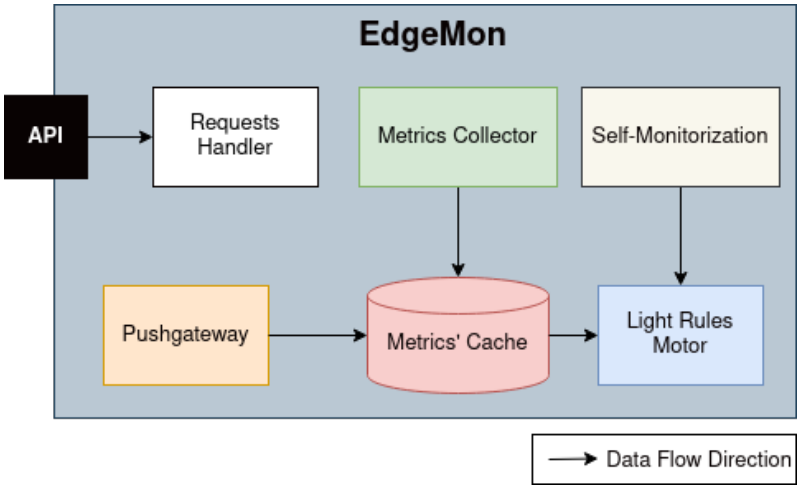


Figure 3.1: Adapted detailed view of the EdgeMon Component [9].

The EdgeMon in Figure 3.1, is responsible for the following actions:

- **Collect Metrics:** With the Node Exporter, it collects the metrics at a host/node level.
- **Store Metrics:** Uses the cache for saving the last collected metrics. It also allows monitoring targets to inject new metrics.
- **Rules Evaluation:** Periodically or upon arrival of new metrics there is an evaluation of a pre-defined set of rules. The time between evaluations is configured by the user. The rules assesment is based on the metrics.
- **Alert Mechanism:** When rules are triggered the correspondent alerts are generated and sent to the Alerts Pushgateway in the cloud. The rules are activated if the metrics are not in the expected range.
- **Self monitorization** Besides the monitorization done at the targets, it is capable do do self monitorization.

Edgemon's goal is to collect metrics of itself and attributed targets. To do so it is in place a periodical cycle. It is responsible for rule evaluation and alerts to the higher layers (CGSM and Alerts PushGateway). Every Edgemon is managed by the CGSM and can also be a target for Prometheus.

#### 3.1.1.1 Type of metrics

There are three levels of metrics: Container, host, and application. When in application-level Edgemon allows performance monitorization and management, and applications' availability. Using these we measure the application performance, optimization possibilities and may detect problems if they happen. For example, the average response time metric of the application to the requests is a good indicator of general performance. Having bigger response times means some part of the application is causing trouble. Another

example is the number of requests per second that helps to understand the current load in the system. Error and success rates, service failures and restarts, and performance and latency of responses are some of the already existent application-level metrics delivered by the Go language library.
To control and optimize the resources given by the instance we have the container level metrics. Throttled CPU Time allows defining correctly the CPU timeshare between the docker containers. A discrepancy in this time means the necessity, or not, of more CPU that the host can or can not give. Some memory metrics allow resources' optimization by limiting the memory used per container, which guarantees the correct execution without wasting memory. Network metrics are also available, indicating the traffic information that can be used to prevent situations such as DoS attacks. Some of them are Connectivity, error rates and packet loss, latency, and bandwidth utilization.
Host level metrics are about infrastructure performance, which is directly connected to the performance of the applications running on containers allocated on it. Examples are consumption of CPU and memory, the number of containers and some others help to keep the consistency of the system and use of resources. Different types of metrics that Node Exporter is capable to collect at a host level are for example: CPU, memory, disk space, processes.

#### 3.1.1.2 Metric Scraping

The time between two scrapes of a target is configurable. This means the interval is adaptable in a way that we can fulfill the request and needs of a client and the capability of the network. For instance, in situations where the variability of the metrics is meaningless for the client, we can have a high scrape interval reducing the weight in the network. On the other hand, in critical situations where we have high variability and want to reduce the alert delivery time, it is convenient to decrease the interval. It is possible to choose per microservice which type of metrics we want to collect (MetricType). The two configurations available are to gather only one kind of metric or any kind. Container-level metrics collect routine is decided whether there is or not an Edgemon in the same node as the microservice. If an Edgemon is not available the same metrics are collected using cScraper and Node Exporter otherwise, the request is redirected to Edgemon. Having an Edgemon it is avoided the deployment of the Node Exporter and cScraper thus not wasting resources.

#### 3.1.1.3 Metric storage and delivery

Data in the Edgemons is stored in a cache, and each time new data arrives the values are changed. There are different modes of scraping: store, node, and full. If the cache is not an option there will be, consequently, a lack of persistence of the metrics it would be necessary for the Edgemon to collect new data when requested from higher layers. This would increase significantly the network traffic, and the Edgemons would be a lot

heavier. In these cases, the client prevents it by changing the scrapeMode. If the network is highly overloaded, the scrapeMode should be "store", because the metrics delivered are the stored ones. In a scenery where the information is crucial the mode "node" would deliver the most recent metrics. The third model "full" gives an evolution vision of the system returning the last stored metrics in cache and the most recent metrics.

#### 3.1.1.4 Edgemon parameters

Edgemon and monitored services are launched in containers as part of a cluster/swarm and orchestrated by Docker. Even though it is possible to use Docker to get information about the containers there is a necessary previous initial configurations for the container when launched:

- Service image.
- Ports to publish the service to make the API accessible to clients.
- To which swarm it should join and so it is necessary to define the overlay network.
- Entry parameters: value that warns if the Edgemon belongs or not to a swarm and the address to where the alerts are sended to.

The behavior of the Edgemon is defined by a set of other parameters:

- Scrapping mode: full, store, or node.
- Scraping interval: time between metrics collected from targets and itself.
- Type of metrics to be collected: all, container, host, or app.

#### 3.1.1.5 Infrastructure limitation and possible versions

In the previous works, two versions of the Edgemon were created (Figure 3.2) one more basic with only the store and metric evaluation (rule and alert mechanism) capability and one more complex doing the same plus target monitorization and self monitorization. There are two versions because since we are talking about the edge, it is important to understand that computational resources are very unpredictable and sometimes very low. So two versions were created to adapt the monitorization activity to the available resources. In this thesis, we always refer to the more complex version.

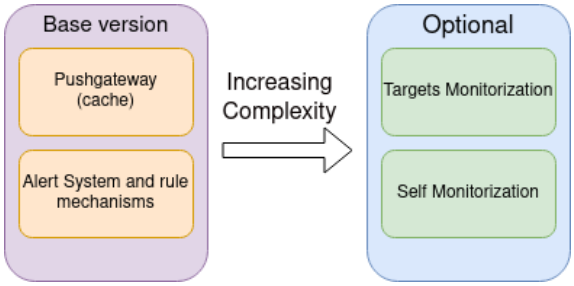


Figure 3.2: Two versions of the Edgemons. Adapted from [9].

#### 3.1.1.6 Edgemon External Exporters

Previous works are based on the fact that an Edgemon is present in each node to be monitorized. When it is not present they use two services to export metrics of host and container level. The two used exporters are:

- cScraper: Service to collect data about container resources and performance. Obtained trough Docker Engine.
- Node Exporter: Prometheus exporter to collect hardware and operative system metrics exposed by NIX kernels.

#### 3.1.1.7 CGSM

The CGSM component represented in Figure 3.3 is responsible to manage the monitoring targets and also the EdgeMons, and in order to cope with the system's changes. It has the following characteristics:

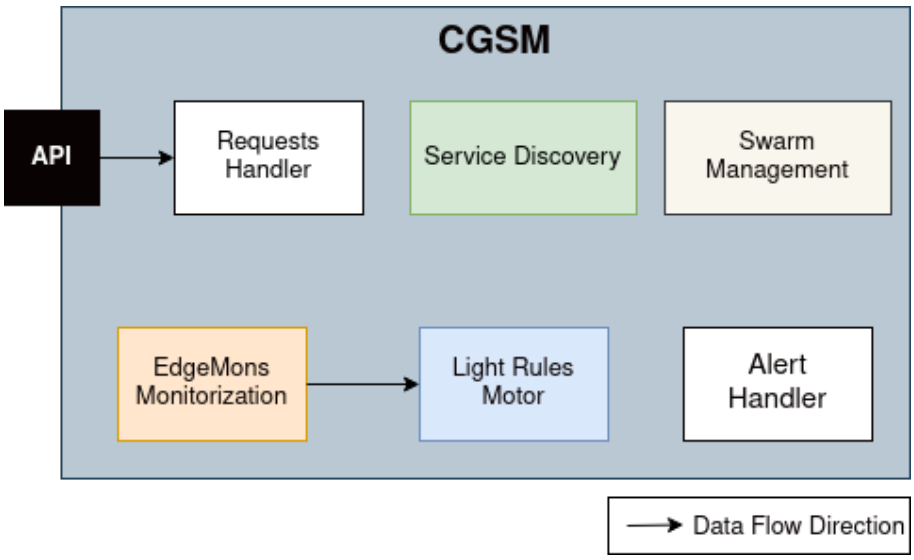


Figure 3.3: Adapted detailed view of the CGSM Component [9].

- Discover relevant services worth monitor inside a cluster and prepare the cluster for the EdgeMon monitor task.
- Optimize the allocation of targets to EdgeMons and monitor existent EdgeMons in the cluster. The definition of this monitoring is made through rules and measurements.
- Manage the EdgeMon life cycle and register new targets for monitoring when they are external to the cluster.
- It is capable of modifying the system architecture by adding, removing, or migrating monitors.

The CGP component (Figure 1.2) was created to manage the Prometheus monitoring targets within its cluster. It has only a service discovery mechanism to inform of each service location and if it is still active to Prometheus. Otherwise, it rewrites the YAML file so that Prometheus' target list can be updated. Its features include:

- Scan for services within the cluster and update the Prometheus target list.
- Add external services to Prometheus target list.

## 3.2 Automatic Microservice Management System - Previous Work

The general goal of the first work that developed the microservices management component [8], was to manage microservices' migration and replication, in an efficient and automatic way, while guaranteeing an adequate QoS.

Figure 3.4 shows a general view of the components needed to make the management component work properly. The infrastructure includes the edge nodes and cloud nodes. Cloud nodes may come from different regions. The nodes form a cluster and its objective is to have a set of resources to execute microservices.

In the Figure 3.4 it is possible to see the cluster master represented as M, and various workers represented as W. The master has knowledge of every node that forms the cluster. The microservices are designated as e-µS. The microservices management component (Services Management) is located in a cloud node and it has the task to initialize the other sub components such as the monitoring component used to monitor the nodes and microservices, as well as to launch the microservices that form the application. In case of a frontend microservice, the management component initializes the load balancer to balance the load between replicas [8].

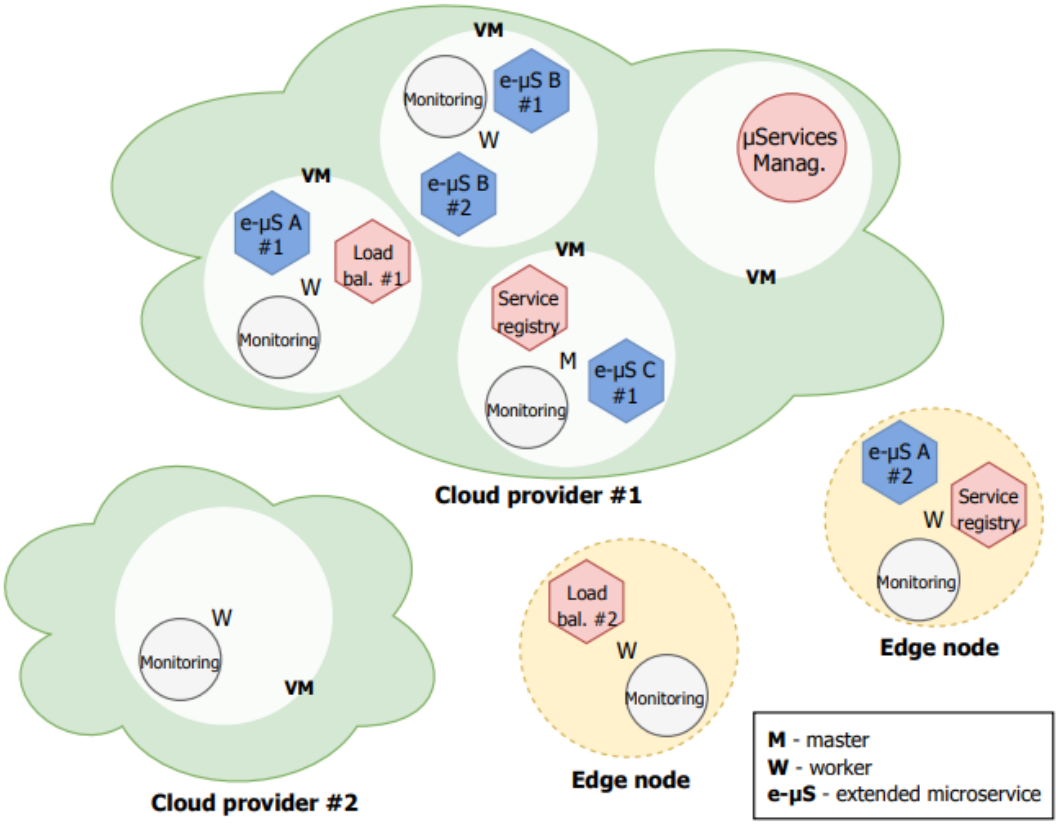


Figure 3.4: Vision of the Microservice Management System and its components [8].

Figure 3.5) shows what needs to be deployed in a node (VM) to support microservices' deployment.

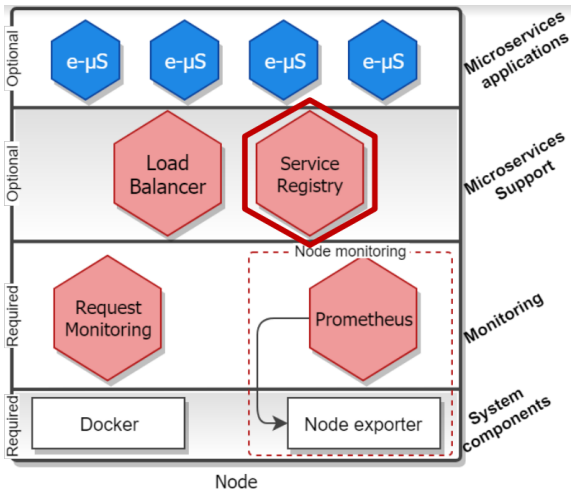


Figure 3.5: Node Architecture [8].

In the lower layer there are the mandatory components:

- A container manager since every microservice is encapsulated in a container.

- A monitorization service in the node, in this case the node exporter, one of the prometheus components, to retrieve scrap nodes' metrics. In this work proposal both the node exporter and Prometheus are to be replaced by the Edgemon.

Next we have a monitorization layer for microservices and nodes. This is necessary to evaluate the microservices and nodes' load, and to help the manager decide the need of any kind of reconfiguration. The following layer is optional because the service registry and a load balancer are not mandatory at every nodes. The service registry keeps the information about the location of a microservice and its replicas. The load balancer distributes the load among the replicas. In the upper level we have the existent microservices in a node.

Figure 3.6 describes the sub-components of the microservice management component and their communication.

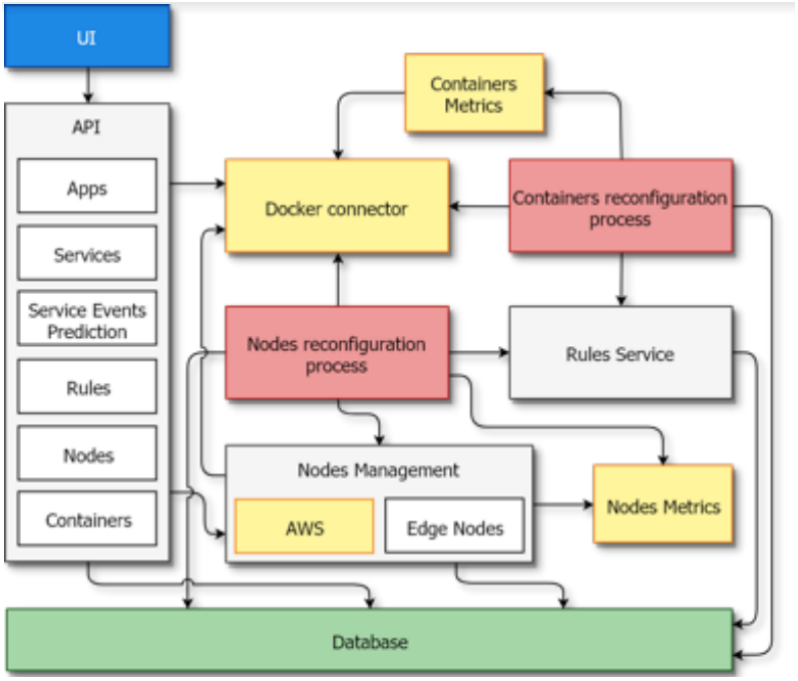


Figure 3.6: Microservices management component [8].

So it is possible to see:

- Client interface (available APIs)
- A database to store the information
- A node management sub-component
- Sub-component to retrieve metrics: Containers metrics and node metrics
- A sub-component to connect with the containers manager
- A rules service that uses a rule engine to specify and fire rules that represent the actions to be performed when a rule is fired.

- A component responsible for node reconfiguration (creates and destroys VMs and decides where they are to be deployed)

- A component responsible for the management of the microservices' containers.

This work was later extended as exemplified in Figure 3.8 and will be described in section ahead.

### 3.2.1 Node Architecture

A node in this system represents a computational node (a VM in the cloud) where it is possible to deploy microservices as we see in Figure 3.5. In a node, it is included a load balancer, registry service, request monitor, Prometheus, and possibly containers with application services. The load balancer based in an NGINX server is responsible for load balancing services' accesses. The allocation is done by country, continent, and region in this order. The service registry associates the name of services to an IP address, allowing them to be discovered. The request monitor then registries the discovery of services requests, upon request. This monitor sends the data to the local manager to decide where to deploy the containers. Prometheus and node exporters are responsible for collecting metrics about the managed nodes, mostly CPU and RAM. Node metrics are collected by Prometheus and node exporter, whilst the containers metrics, like bytes statistics (used bytes), are obtained through docker stats (available in docker, section 2.4.1). Furthermore, components were added to the node, namely, the option of Kafka agents and local managers. Every element is initialized when it joins the swarm. The docker is assumed to be already installed and executing in the machine.

### 3.2.2 Managers

The Automatic Microservice Management System rests in 3 managers (Hub, local, and master managers) from Figure 3.8. In turn in the Monitoring System the most important components are the CGP, CGSM, and the Edgemon module from Figure 3.9. We will only focus on the Edgemon because the other modules were removed and their functionalities reconstructed in other modules in order to evolve the whole system.

### 3.2.3 Hub Manager

The Hub Manager allows the user to interact with a visualization of the current system state. It includes the possibility of triggering actions in the managers, such as to launch containers and control their states, and supports having a view of the current nodes and virtual cloud instances. The Hub Manager launches support components, modifies, adds and removes applications, rules, services, and even simulated metrics. It also includes non-functional characteristics for instance actions feedback, filters, forms, and much more.

### 3.2.4 Master Manager

The Managers components to automatically manage the microservices allow the definition of rules and metrics to use in migration and replication of containers to improve performance and node occupation and, to increase or decrease the number of replicas of service containers. The Master manager represents the management of the whole system, functions include:

- Collect all the data from managers, applications, services, simulated metrics and defined rules.
- Deploy an API to the Hub manager and the user.
- Allocate elastic IP addresses in the AWS (the adress should be known from start).
- Manage the support components (Local managers, registry servers, Kafka agents and load balancers).

### 3.2.5 Local Manager

The utility for local managers is to manage a set of nodes and containers in a region through metric analysis, application of rules, some monitorization, and trigger actions. Local managers do not interact directly with the master manager, instead they use Kafka agents of the same region. Its functionalities are:

- Monitoring a set of nodes and containers to get their metrics.
- Include adaptation cycles of the system through metric analysis, applying rules, and making decisions based on the collected information.
- Deploy an API to the hub manager to access the current state and manually manage it.
- Send periodic Heartbeats to the master manager.

### 3.2.6 Operations Support Components

Operations Support Components are a set of necessary components used in the previous work Figure 3.8 [10] to support the implemented system functionalities. They do the registry, service discovery, collect metrics, data communication, and load-balance of services.

#### 3.2.6.1 Proxy with basic authentication

The authentication proxy was already in place to protect the information kept by the docker in each node, so it is still used.

#### 3.2.6.2 Registry server and client and Service discovery

The scalability achieved, namely to use different regions, implemented a solution to allow the servers to replicate the replicas of the microservices. It was used an approach based on IP elastic addresses in the AWS to associate them to public known IP addresses at the beginning of the execution of the system. The registry server and client service were included to allow the discovery of services by the other services, this was based on the Eureka from Netflix.

On the registry client, the discovery algorithm uses the city, country, and continent values associated with each replica of microservices. Instead of using words, the values are in coordinates. Coordinates permit us to calculate the distance to the replicas and select them based on it. To avoid the selection of the same replica for a lot of dependent services two more algorithms are in place [10] alongside a randomness factor. In the previous works to facilitate the communication of microservices, registry clients, and service discovery APIs were added to support the microservices' languages such as Java, Go, Python, C, C++ e NodeJS.

**Limitations:** Because of the necessity to previously know the registration service public IP address in the launch moment, the registry server can only be initialized in the cloud and never on an edge machine. Nonetheless with the new architecture, it is not necessary for the services to registry themselves.

#### 3.2.6.3 External services requests' monitorization

The external services request was used to allow the association of the localization and the number of service accesses by the microservices. External services requests monitorizes each replica according to coordinates. By collecting the information, the local manager can access it and choose the best location to migrate or replicate a container of a service, normally nearer the dependent services.

#### 3.2.6.4 Prometheus and Node exporter

The previous works included the Prometheus and node exporter executing in every node to collect metrics (real and simulated) from the machines in the system. Despite this thesis's inclusion of monitorization components, Prometheus and Node exporter are still in place to collect node metrics of the monitorization components.

#### 3.2.6.5 Load balancer and configuration API

A load balancer, based on an NGINX server, is used to distribute requests from the frontend. Its configurations allow supporting various types of frontend services simultaneously.

The load balancer component uses version 2 of Maxmind GeoIP [1]. There is one load balancer per region so the requests are rerouted to a random server in the region.

**Limitation:** The load balancers do not have access to the other services' load balancers (in different regions). Because of it, the user may receive a 404 Not Found, saying there is no server sock-shop-frontend running, while it was one replica executing in Europe Fig 3.7.

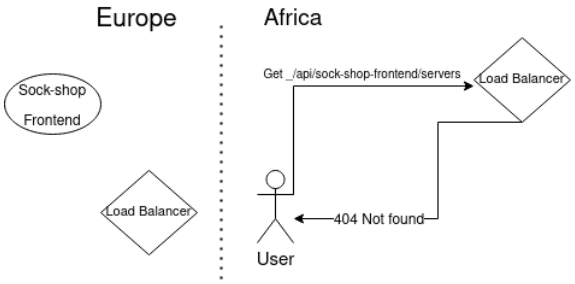


Figure 3.7: Load Balancer limitations [10].

#### 3.2.6.6 Kafka Agent

In the previous work, Kafka agents are used to access data from the master manager. A Kafka agent is a cluster node managed by the Kafka system to where logs of topics can be replicated. In this way, the data synchronization can be made between the master manager and the local manager and vice versa. A Kafka agent is launched in each region. In this thesis, a new topic was created at the Kafka agents to facilitate synchronization between the Edgemon and the master manager.

In the following we describe the extension plan to join both existent works.

## 3.3 Integrating a Microservices Manager with a light Edge Monitor

The main goal of this thesis is the integration of the two previous works mentioned in section 3.1 and 3.2.

The first one is a light monitorization system. The second one is an automatic management system that uses Prometheus as a cloud monitor while deploying the node exporter in the nodes. In fact the second integrated system was an extension of this automatic management system as presented in Figure 3.8, and described in Thesis "Gestão Dinâmica de Micro-serviços na Cloud/Edge" [10].

The first one is an automatic management system that uses Prometheus as a cloud monitor while deploying the node exporter in the nodes. The second one is a light monitorization system.

[1] https://www.maxmind.com/en/geoip-demo

The aim is that the Management System uses the Edgemon as a local monitor at an edge node and the region manager can define:

- The necessary rules for infrastructure nodes and microservices constraints.
- Query the monitoring data and be notified of relevant events.
- Define in what way the monitoring system must adapt.

Thus by extending the two existent works the objective is to get closer to a self-adaptable monitoring system with a set of functionalities that fully support the edge's inherent dynamic context, both with higher efficiency and lower resource usage.

## 3.4 Merge the Hybrid Monitoring System and the Automatic Microservice Management System

The automatic manager system described in section 3.2 was extended to a hierarchical definition, with local managers per region, and a central manager that has the vision of the whole system. This was the system used in the integration and its architecture can be seen in Figure 3.8. When we see this previous work on the architecture of the Cloud/Edge's Microservices dynamic management System (Figure 3.8 ) it is possible to see the use of multiple regions on the globe. This is a hierarchic distribution where each region deploys a Local Manager (Section 3.2.5), a Kafka Agent (Section 3.2.6.6), a Login Server (Section 3.2.6.2) , and a Load Balancer (Section 3.2.6.5). To control the hole system it uses a master manager (Section 3.2.4) in one of the regions while the data synchronization between managers is reached through Kafka.

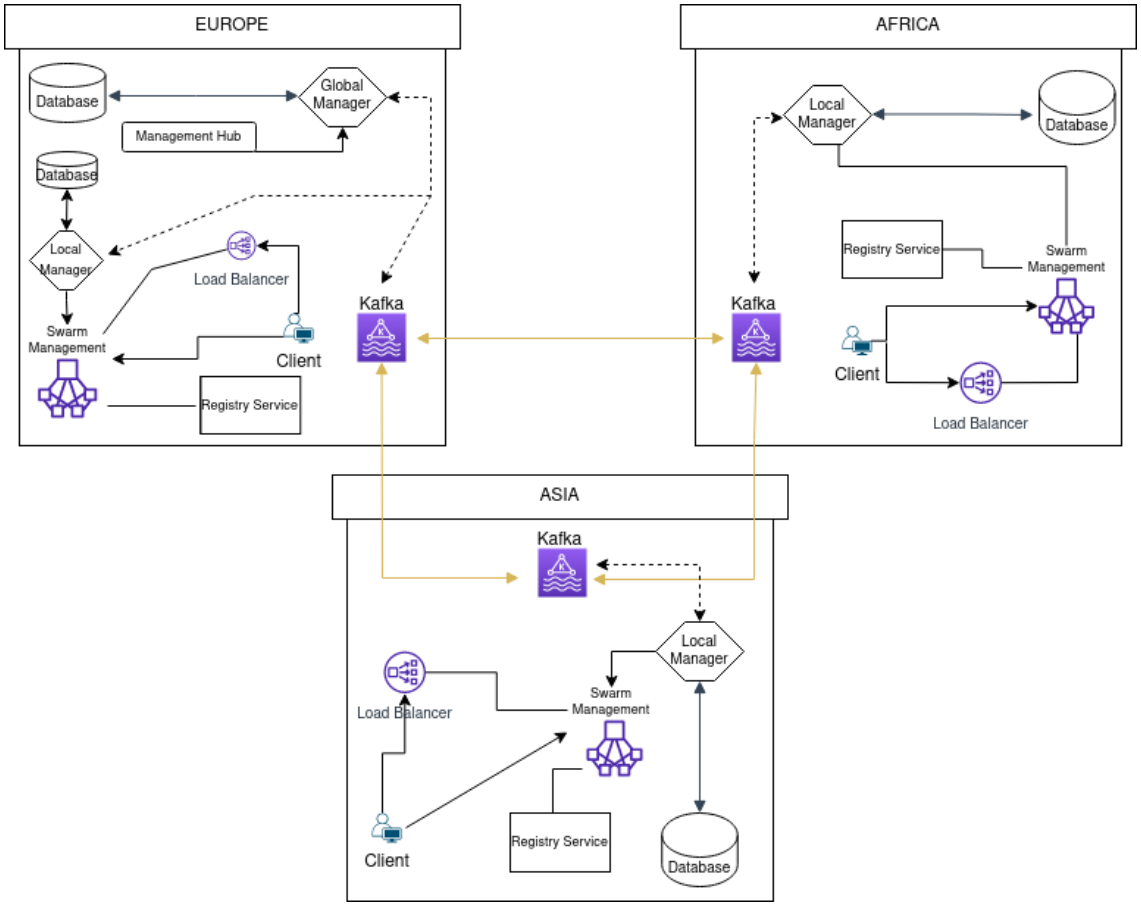


Figure 3.8: Previous image of the simplified Dynamic Management System [10].

From the Hybrid Monitoring System, Figure 3.9, we inherit the Edgemon, Monitor System Manager (CGSM), Prometheus Manager, Pushgateway, Alertmanager, Node Exporter and the Cscraper.

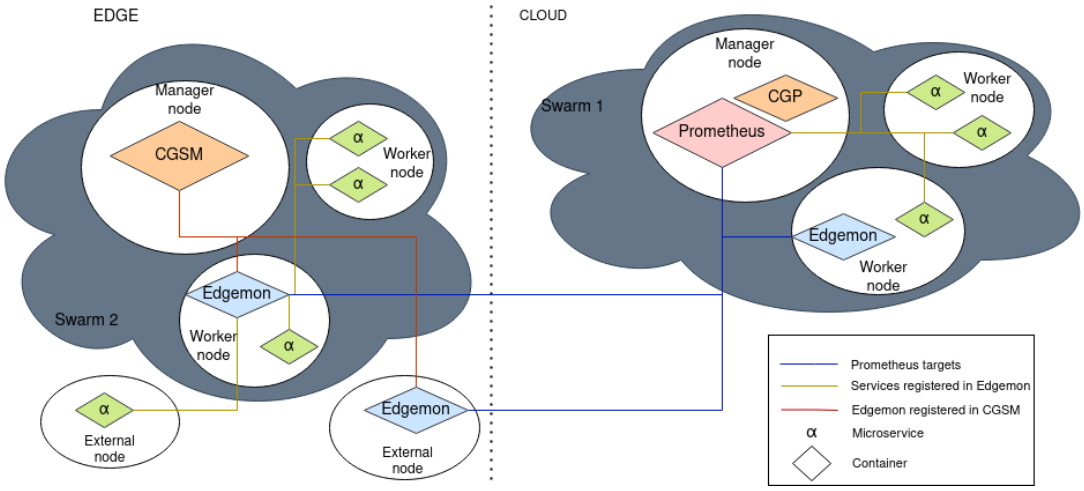


Figure 3.9: Previous image of the simplified Hybrid Monitoring System. Adapted from [9].

It is important to understand that this dissertation comprehends the integration of a Microservices Manager with the Edge Monitor both mentioned above. The final architecture deploys a Master Manager in one of the regions that controls the System throughout all the regions. Each region deploys all the expected components and in the initial moment, one Edgemon is controlled by the Local Manager. The CGSM, component responsible for the System management and service discovery, is built into the region manager. This detachment of the CGSM (centralize module) will guarantee more independence of the Edgemon modules because instead of all the Edgemons reporting to one entity, they will interact with their own local manager, that now provides the functions of the CGSM for example the service discovery and attribution. On the other hand it will increase not only the workload of the Local Manager but also its complexity.

In the following, we describe the components that are relevant for this proposal.

## 3.5 Managers

The base of the Hybrid Monitoring and Automatic Microservice Management System rests in 4 principal components, 3 managers (Hub, local, and global managers) from [10] and the Edgemon module from [9]. The union of these two sets of components are to manage the system in a way of automatically adapt it. When facing different situations it is to always pursue the best decision to make the solution more efficient and ultimately deliver a QoS to the user.

### 3.5.1 Hub Manager

In the Hub Manager it is now possible to verify the state of the monitorization components. We can see the existing Edgemon and its state, labels, logs, ID, location, etc. The most

important aspect is that we can enter directly in the Edgemon component and see not only the scrape tasks and targets with all the information and collected metrics but we have access to all the API of the Edgemon component. When accessing the Edgemon in the graphic interface it is only possible to see GET functions. This limitation is due to security and performance reasons to reduce the outside influence, limit the possible accesses and give more independence to the monitorization.

### 3.5.2 Master Manager

The Master manager represents the management of the whole system, his functions, besides the ones in the previous works, include:

- Manage not only the support components but also edgemons.
- Receive requests and registry all Edgemons created, to be consumed by Prometheus.

### 3.5.3 Local Manager

In this thesis, the Local Manager became more complex because of the increased capability and involvement in the system. Its increased complexity is due to functionalities such as:

- Manage the Edgemons' life cycle to optimize the efficiency of the system
- Manage and allocate microservices to Edgemons.
- Receive the alerts from the Edgemons self monitorization.
- Deploy an API to manage the Edgemons' (send and receive requests).
- Send creation and delete flags of Edgemons to master manager.

### 3.5.4 Edgemon Component

The biggest advantage of using an Edgemon is that it is a light monitor. That means it can be deployed in nodes with low computational resources when compared to Prometheus (verified in test 5.2.4 [9]). The work in the present thesis took advantage of the already existent Edgemon module, and some independence was provided due to decoupling it from the CGSM (Section 3.1.1.7). For example, the heartbeat is not in place (creation and delete flags are in place) and the alerts are not sent to higher layers but to the region manager. Edgemons when created receive a request with its region manager address for future references (figure 3.10).

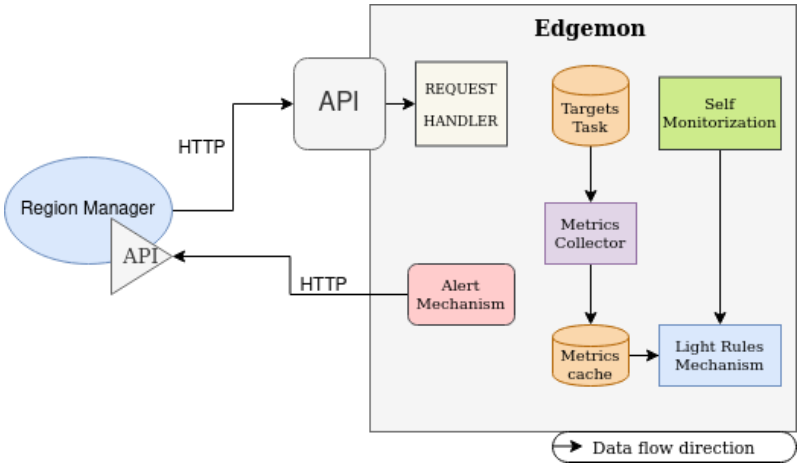


Figure 3.10: Detailed vision of the Edgemon. Adapted from [9].

#### 3.5.4.1 Rules assessment and consequences

From previous works, there is the rule mechanism. The metrics are characterized in rules and then evaluated by this light rule mechanism. Having this mechanism in the edge zone reduces significantly the network traffic that would be caused by the transmission of metrics from the edge to the cloud in order to deliver alerts in real-time. So when the type of the evaluation is simple, it is done in the edge thus reducing the traffic. When it is needed a more complex evaluation, for example involving metrics with time intervals or more than one type of metrics, it is done in the cloud.

Collected metrics are evaluated with rules when they enter the cache or when the rules are created. It is possible to add rules dynamically, introducing the name, type of metrics to evaluate, the supposed interval of the metric value and the type of alert to be thrown (see Figure 3.11). An important factor is that the Edgemon can do self monitorization. By granting the monitor to the possibility to periodically collect information of the containers and infrastructure and filter that information with the rules it is possible to notify the higher layer. When activated, the monitor alerts the higher layer, namely, the region manager to, if necessary, adapt the system (for instance to create a new Edgemon).

In this thesis, the higher layer is now the region manager, and by doing so the system can adapt if necessary (ex. Replicate or migrate microservices) to maintain its balance. The crash of monitors because of the load is an example of situation that is prevented because of the self monitorization and the capability of the higher layer to take decisions. Rule values are definitive and always defined when the rule is created. Situations where rules become active, are:

- Detection that the metric value passed as argument is not in the defined values.
- Detection if the metric does not reach the minimum value.

| Metric Name | Superior Limit | Inferior Limit | Metric Type | Alert |
|---|---|---|---|---|
| container_memory_usage_bytes | 20600000 | 0 | Container | Memory use Exceeded 20MB |

Figure 3.11: Example of a rule [10].

In the previous work the alerts were sent to the CGSM, section 3.1.1.7, now this component was not included in the architecture because the alerts are sent to the regional manager.

#### 3.5.4.2 Edgemon parameters

Edgemon module maintains basically the same parameters described above in section 3.1.1.4 with some slight changes. We only use all the parameters if we want to launch it manually and alone without being part of the system. The only necessary previous configuration upon launch is:

- Service image.
- Ports to publish the service to make the API accessible to clients.

The first Edgemon is launched always in the same port, then when in need to create more, it is in place a mechanism to find free ports to launch the new Edgemon container.

#### 3.5.4.3 Microservices management and its attribution

Every time the local manager initiates a service, it has to be associated with a monitor. To do so it was developed the algorithm: Upon the start of an application, launched containers are stored in the system manager. A routine iterates the containers and for each container pulled its region. Then, using the region, it is obtained the region manager address and a list with the region's Edgemons. In the initialization of the system, in each region, a Edgemon is created. If when we try to pull the region monitors' list we reach a null value, a routine will create one in every region. When we have the list, if the Edgemon is not overloaded it is chosen. Otherwise goes to the next on the list. When chosen, a Scrapetask object is created. Scrapetask is a class created to represent a task to the Edgemon with variables such as shown in Figure 3.12. When built it is saved and sent to the module throughout a request with the task as a JSON object. The overloaded boolean variable is updated accordingly to the overload metrics evaluation based on the monitor.

| ScrapeTask Variables |
| --- |
| Targets - Represent the microservice's address to be monitored; |
| Labels - Labels to the containers; |
| ScrapedBy - Edgemon's name; |
| SwarmNodeId - Swarm ID; |
| ContainerID - Container ID of the target microservice; |
| ServiceID - Service ID of the target microservice; |
| EdgemonAddress - Edgemon's address; |
| EdgemonContainerID - Edgemon's container ID; |

Figure 3.12: ScrapeTask's most relevant variables.

#### 3.5.4.4 Alert receive and management

When the alerts are triggered from Edgemons auto-monitorization, they are sent to the region manager. Facing the alerts and the value of the metric out of the boundaries, the local manager takes one of two decisions:

- Load Partitioning: Half of the scraped targets are released from one Edgemon and attributed to another.
- Migrate: It is created a new replica if there is none in the same region and the whole load is passed to it.

## 3.6 Node Architecture

The node in our work is similar to the previous works. The local manager is responsible for every Edgemon that may exist in its node or in other nodes as long as they are in the same region. If not there will be another local manager. The major difference in the new node architecture is the fact that we have an Edgemon as an optional component (Figure 3.13). Edgemon is an optional component because not all nodes will have an Edgemon but at least one node per region will. The number of Edgemon components changes via adaptable mechanisms depending on the number of microservices to be monitored. The idea is for the local manager to attribute targets to Edgemon to collect metrics about them. These targets may or may not be in the same node. From the previous work [9] the Edgemons can collect its container and node metrics. We do not use it for this purpose because the integration of the CGSM, Section 3.1.1.7, with the local manager allowed us to deploy Edgemons only in some nodes. Thus reducing the weight in some edge nodes. To collect node and container metrics, the work still maintain the light Prometheus and node exporter. Nevertheless, it is possible to remove the Prometheus and node exporter (red box). It is important to notice that the Edgemon is still necessary and crucial for good system performance but we do not need it in every node. In other words, we need at least

one Edgemon per region. It is necessary for every microservice to be monitored but not every node to have a Edgemon.

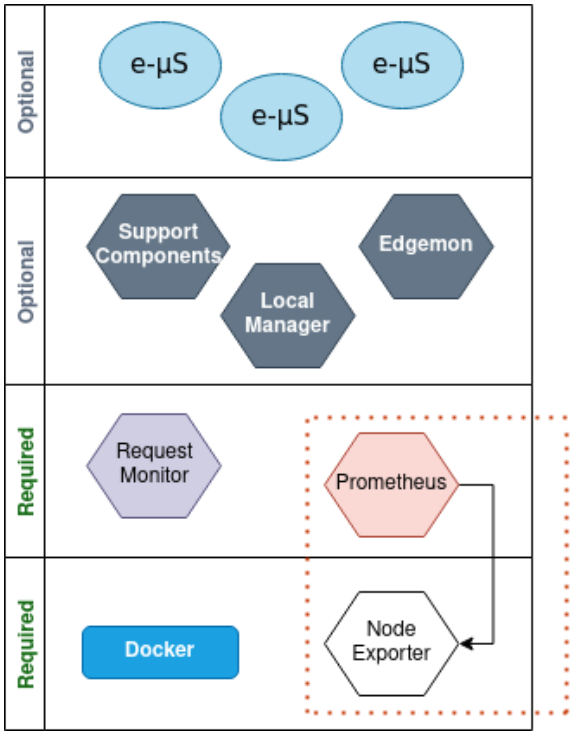


Figure 3.13: Proposed Node Architecture.

## 3.7 Components Interaction - How Everything Connects

When we look at the edge it is noticeable the dispersion and heterogeneity of the nodes and the different types of microservices lodged on them. This new architecture, Figure 3.14, is organized in clusters, which are swarms in the docker environment. Every node can launch a microservice and at the same time it is monitored and it is capable of monitoring the microservice. So we are talking about a system with a different number of clusters, hosts, and nodes. We maintained the concept of the previous works, namely, the hosts are every machine controlled by the system whether they are in a manager cluster or not. In the hosts' category, it is possible to distinguish the cloud virtual instances and the edge machines that the system has access to. One host becomes a node, i.e part of the system, when it is added to a manager docker swarm. When added, the configuration is made with 2 containers to support the operations that are in every node (basic authentication proxy and request monitor defined in the image in the operations support components module). Because of the different capability of resources, two types of cloud instances were defined. One type is to lodge load balancers or microservices and the other type has more capacity where now it lodges not only local managers, Kafka agents or registry servers but also Edgemons to monitor the region services alongside with the local managers. Because of the undefined number of microservices and the need to ensure the their monitorization,

the Edgemons are also undefined regarding their quantity and they use geo-localization mechanisms. This ensures that the targets are monitored by the closest monitor reducing the latency and increasing the performance of the system. In the case of one monitor getting overloaded, there are in place mechanisms of adaptable solutions to generate new and closer monitors. As the microservices, the monitors are also monitored. To make each node more independent, every time an Edgemon is created it sends a request to the global manager showing that it exists. The global managers then build a list of existing monitors. Feeding this list to the Prometheus eliminates the need for Edgemons to talk to each other or require monitoring other Edgemons. The communication with the Edgemons component as in the previous work is done using APIs REST.

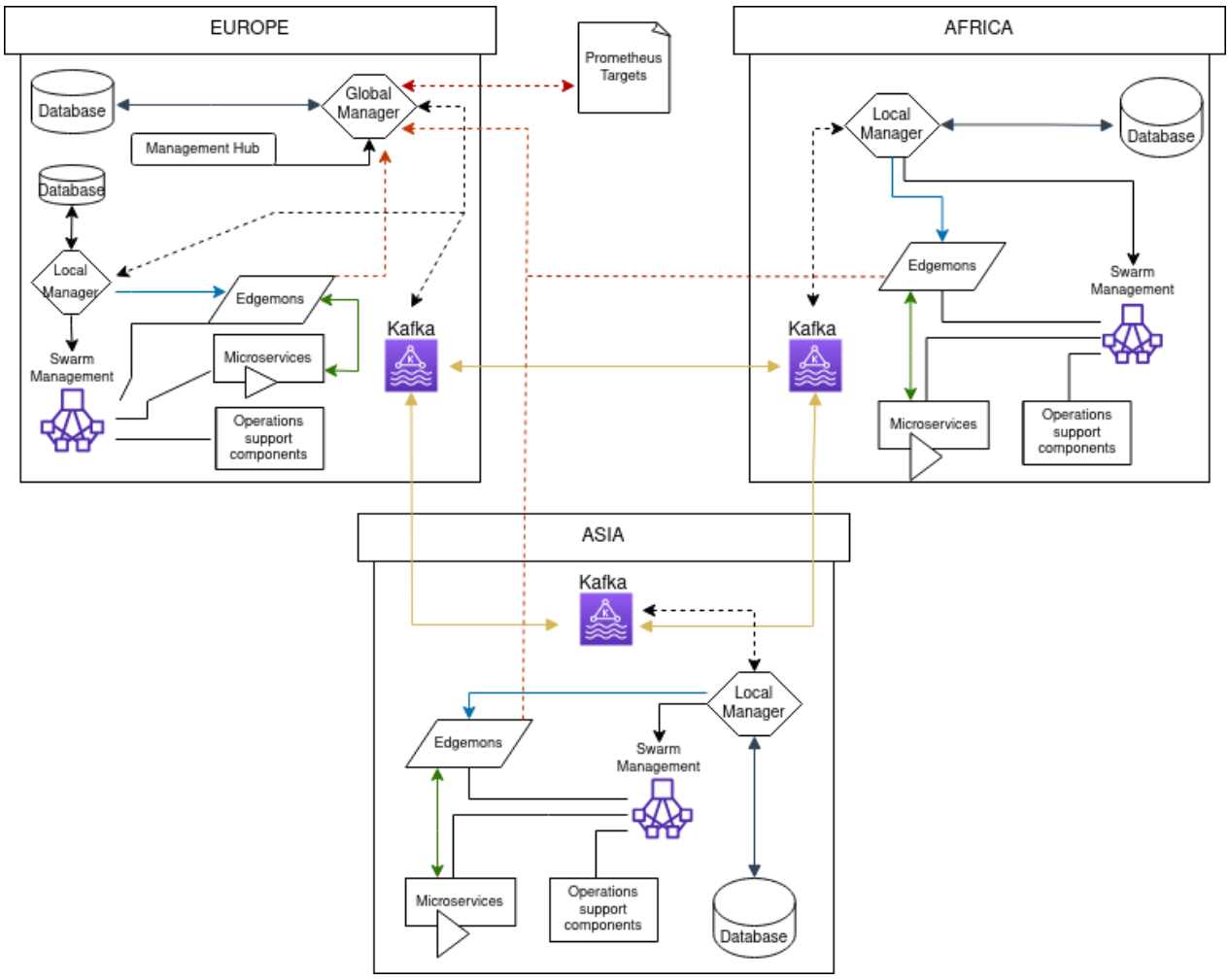


Figure 3.14: Proposed Architecture.

So in the end we have a system that starts with the launch of components by region. In one of them we have the global Manager while in the other regions we have local managers. In every region, we have components to manage and monitor microservices. Every time a node joins a region (manager's cluster), it is configured with the managing components to support operations. When any node decides to launch a microservice it is added to a

monitor if present and capable, otherwise a new one is created and the information goes to the global manager. Then while collecting metrics the system will adapt accordingly with needs using a set of rules, whether is to replicate or migrate service monitors, or to change the Edgemon from one region to another.

### 3.7.1 Interaction between components

Every component of the system deploys an API some added and others from previous works [10] and [9] (Annexes I - VII) to receive requests . To communicate between components, it is in place a HTTP protocol in a REST architecture. So every HTTP request triggers an action to, these operations mostly used by the following pairs manager HUb - Master manager; Master manager - Local manager; Local manager - Load Balancer; Registry service - microservices; Local manager - Edgemon. Some microservices need to communicate with each other, it is used RPC technology between processes that allow remote execution. There are even more complex communications to be done between microservices. RabbitMQ is a system of message transmission with asynchronous transmission and distributed deployment.

### 3.7.2 Prometheus and Master Manager

Prometheus and the Master manager do not have a direct relation, nevertheless, the master manager has the job of receiving a request every time an Edgemon is created from the regional manager. With the requests, the master manager builds a file of every existent Edgemon, this way the Prometheus can consume the file. When consumed, the Prometheus and the Edgemon form a relation of server-client where Prometheus is the client. Edgemon is now a Prometheus' target and through the API of the Edgemon, he can request metrics ($\backslash metrics$).

### 3.7.3 Managers and Edgemon

Previous works CGSM section 3.1.1.7, and CGP section 3.1, are now in the Local and master manager. In other words, it is the region manager that does the management of the region Edgemons. At the beginning of the system, the local manager creates an Edgemon. Every alert or necessity is transmitted from the Edgemon to the manager and he acts accordingly. Edgemons and system necessities include replication, migration, attribute targets, creation, and elimination. The managers attribute targets to Edgemons as a single service or a list of services both in JavaScript Object Notation (JSON); In this format are also sent rules from managers to Edgemons. On the other hand, JSON format requests are also sent by Edgemons when alerts are triggered.

### 3.7.4 Edgemon and Target

Unidirectional or client-server relation. In the first case, targets inject the data directly into the monitor. On the second, through the endpoint \metrics it is requested (HTTP request) the metrics. It is important to notice that the metrics collected by the Edgemon from the targets are only at an applications' level. The other types of metrics are collected from the containers and hosts.

### 3.7.5 Edgemon and Edgemon (Same region)

Edgemons communicate via region manager, they only interact to relinquish targets.

### 3.7.6 Local Manager and Master Manager

Master manager communicates with local manager through HTTP to redirect clients requests, launch applications, and services, launch and synchronize containers, add and remove and synchronize nodes. It also communicates its actions and adds, and removes monitors. Both of the managers deploy APIs.

# Chapter 4

# Implementation

The implementation chapter has the objective to clarify the used technologies in section 4.1 and in section 4.2 it is explained aspects of the implementation across the system.

## 4.1 Used Technologies

### 4.1.1 Languages

As coding languages, for manager master (Section 3.5.2 ) and local managers (Section 3.5.3 ) including database and service libraries it was used Java 11. The system also includes applications of microservices implemented in Java, Go, Python. Go or Go lang was developed by Google in 2007, and it is a compiled language with garbage collection and runtime reflection. It is efficient and fast with concurrency mechanisms that maximize the use of multi-core machines or network connections. Go language was used for components such as the monitorization components. Go is a lightweight, flexible, and modular code with asynchronous execution is an advantage because these components are frequently used in the program with the possibility of being present in each node. It was not necessary to implement a microservice from scratch but because of compatibility and testing purposes, scripts and microservices were changed such as YAML and Dockerfiles.

### 4.1.2 Databases

The master manager uses the H2[1] database motor to change the data in a disc file. On the other hand, the local manager also uses H2 but saves the data in memory because using Kafka agents allows them to retrieve system information. To retrieve class objects is

[1] H2 database: https://www.h2database.com/html/main.html

used queries to the memory or the file database. Some services use Redis[2] or MongoDB[3] because of data persistency. Each microservice database is launched in a container with the service container. Memcached[4] is also used in some microservices to increase the data access speed through memory caching.

### 4.1.3 Frameworks and Libraries

This section is destined to enumerate the principal frameworks and libraries used in the development and improvement of the components:

 Spring Boot allows creating Spring applications pre-configured making easy the development of some parts of the application. Spring is a framework to develop Java applications in this case used to implement the managers.

 Project Lombok is a library that through annotations automatically manages code that is repetitive in Java language. In this case, it generates getters and setters to the objects, constructors, or even methods like toString, hashCode, or equals.

 MapStruct is another library that uses annotations and classes properties detection to generate code. Allows to route Java classes and is used to route database JPA and DTOs transferred between managers.

 Spotify's Docker Client is used to enable the Docker functionalities in a Java application.

 React Simple Maps brings the interactive maps and the notion of selecting coordinates. Uses d3-geo[5] and TopoJson[6] to create the maps.

[2]Redis: https://redis.io/
[3]Mongo: https://www.mongodb.com/
[4]https://www.memcached.org/
[5]https://github.com/d3/d3-geo
[6]https://github.com/topojson/topojson

 Gorilla Mux is a Go package that implements the request router by pairing the endpoints to the method that processes the request. Used in the registry client and request monitor to enable the REST APIs.

### 4.1.4 Tools

To help develop the system a set of tools was used:

 IntelliJ IDEA was one of the Integrated development environments (IDE) used to write the code of some components.

 VSCODE was the most used IDE, used to write and manipulate all the components of the system. Additional extensions were added: (ms-python.python, vscjava.vscode-java-dependency, golang.go).

 Kafkacat is a tool capable of producing and consuming Kafka topics. Used to debug the trade of data between managers.

 Golang playground is a web service that allows to execute Go code through a browser.

npm npm installs and manage the hub manager library packages.

Maven Maven installs and manages dependencies de Java written components as well as build the respective file - Java Archive (JAR).

### 4.1.5 Products

The main products included in the system were:


Docker is a set of products that uses an operational level of virtualization to deliver software in modules called containers. The containers are isolated from each other with their software, libraries, and configuration files. It was used Dockerfiles[7] to define the construction of docker images stored in Docker hub repositories used to initiate the containers. Each manager starts and manages a swarm through docker swarm[8] to manage the nodes.

Kafka defined by a platform of event streaming, propagates the data between managers.

Drools provides the templates to implement the hosts, services, and containers rules defined in the system.

## 4.2 Management and Monitorization system

As already mentioned the system is built in two essential parts. The management is done with managers (master, local, and hub) and the monitorization (Edgemons). In the previous thesis, we had the notion of a CGSM (3.3) that, despite being discarded, its functionalities are now assured by the new managers referred in section 3.5.2 and 3.5.3. The Edgemon and the monitorization part of the manager use the same technique to manipulate data. They store data in memory and resort to disc storage to recover from failures. Now we also have a file that stores the existent Edgemons.

Fast and frequent access to data and compatibility are advantages valued in using memory. So writing and reading RAM is prioritized with the help of data structures. The alternative could be to use an in-memory database (H2 like the managers for example) that offers functionalities that help manipulate big data blocks. But it would be more expensive in terms of CPU and memory. It is important because it is possible to have a scenery where the execution would need an Edgemon in every node. The disk storage was made through JSON and YML files.

[7] https://docs.docker.com/engine/reference/builder/
[8] https://docs.docker.com/engine/swarm/

The number of writings in the disk is considerably less, happens only when an Edgemon is launched, a new target is registered or when a rule is added. The readings only occur at the beginning of execution and in the Prometheus routine to read his targets.

To allow multiple and parallel requests there is a goroutine service to each connection. This goroutine service is due to listenAndServe functions that receive an address and handler from net/http[9] library. Sharing memory access with goroutines needs to be controlled to guarantee consistency and integrity. Took advantage of Map type from sync[10] library, which has synchronized mechanisms making possible safe accesses. Concurrent maps were used since they were already offered by the library with advantages over Mutex and RWMutex separately. They would lock every map bucket while in the concurrent map each bucket has its Mutex.

The new architecture is divided in:

- Four main components: Master manager, Local manager, Hub manager, and Edgemon;
- There is one library to define the database used by the managers and another one with code that implements common services used by the managers;
- Seven support components: Authentication Proxy, Registry Server, Registry Client, Request location monitor, Load-balancer, Load balancer manager, Prometheus and Kafka agents.

### 4.2.1 Managers Library

The libraries have been increased with Java code with the definition of entities and database repositories, as well as services from the managers. The libraries allow sharing of code between the master manager and the local manager. This is mainly because there are several functionalities that are present in both. Before the compilation, the libraries are compressed in JAR files allowing them to get imported as a dependency for example in a pom.xml used by maven[11] in the compilation.

Now the database library defines more than forty-nine JPA entities which are translated into more than sixty-two tables after being processed by Spring[12] framework. The tables store essential data to allow the execution of managers through applications, containers, services, cloud and edge machines, nodes, conditions, rules, metrics, support components, heartbeats, and the entities' relations data. Service library now includes more than forty-one services implementing common functionalities of managers, from docker swarm, manage and configure nodes and hosts, execute SSH commands, start containers, and list, add or remove entities.

[9] https://pkg.go.dev/net/http
[10] https://pkg.go.dev/sync
[11] https://maven.apache.org/
[12] https://spring.io/

### 4.2.2 Master Manager

The master manager, is implemented in Java with the use of the framework Spring Boot from previous works. It does the principal management of the system, being the central point where applications, Edgemons, services, rules, metrics configurations are stored, manage the components launched in each region. At the beginning of execution, it configures the initial system. With the addition of Edgemon's concept, the master manager at the beginning of the execution launches the Edgemon service alongside the other services and creates the first Edgemon in the node where the program was launched, the master node. Every time an Edgemon is created it registries it on the Prometheus target file. To interact with the other components and receive requests, the master manager also starts an HTTP server that provides an API (annex: I).

There are two modes to run the program, LOCAL and GLOBAL. The local mode only launches the master manager, which means it will only have one node associated with the edge and no cloud instances. While the global, at the start of the program allocates IP addresses from the cloud and launches local managers on them. In this thesis, it was developed a third mode, NETWORK. This new mode tries to get a little of both worlds while being in the same network using more than one node. The advantage of this mode is that now the program runs in a more controlled system, because it is smaller, while at the same time using the REST APIs, address requests, and the resources of one or more machines. It is possible to select whether the other nodes would be just usable hosts to host microservices, or worker nodes, witch means being controlled by a local manager in spite of being in the same region.

### 4.2.3 Local Manager

The local manager implemented in Java using Spring Boot framework and it's launched in a container by the master manager, at most of one per region. The component runs inside a docker container. As the architecture evolved, the need to use previous modules like CGSM (section 3.1.1.7) was eliminated but the concept of its functionalities was preserved. In this new architecture, the local manager gained more complexity and responsibility in the system by being, the manager of the local monitoring modules. The local manager initiates a server HTTP where he exposes its API (Annex II and III) that now is a union between the WorkermanagersService.java (class inherited and changed from previous works) and the EdgemonsService.java (new class that represents the link connection between managers and Edgemons). For further understanding, relevant functionalities will be described later.

#### 4.2.3.1 Service Discovery

When a local manager is launched an Edgemon is created and it will monitor any service that is launched by the local manager and it feeds the local manager with metrics. The big difference between adaptive systems and static systems relies on the fact that to evolve the second needs human interference. An adaptive system responds to events it is programmed for. In previous works, when launching an application, the best suitable place where to launch it is already implemented. So in this thesis, we took advantage of it. With at least one Edgemon per region, it is possible to monitorize the running application. As well as when the application service gets offline and the local manager eliminates it, the associated Edgemon does the same. To launch an application, the local manager needs to receive the order sent by the hub manager, then collects the services that are associated with that application, seeks the available hosts, and distributes the services. At that time, the local manager searches the list of possible Edgemons to whom it can give the monitoring task, and assigns them.

#### 4.2.3.2 Edgemons task attribution

To maximize the efficiency of the monitorization system, two major factors are taken into account:

- Edgemon load. Besides the routines to evaluate the metrics and the rules, when attributing cargo to the Edgemons, we have to see if the Edgemon is available, in other words if it is not overloaded. If it is, a new Edgemon has to be created.
- The positions of the Edgemon and the launched microservice. It is already known that in this architecture one of the main factors to the attribution has to be the region in which the service is launched. So when a new microservice is launched, we consider its region. From the region's Edgemons, it contacts the closest. If it is available it is attributed, if not it searches for the next one. In the end, if none of the Edgemons are available, a new one is launched to monitor the microservice.

When the Edgemon is selected a class scrapeTask is created, it is turned into a JSON object through the endpoint /app. The request will follow through the Edgemon module, which is going to start to monitor the service after adding it to the targets list. The code can be seen in listings 4.1 and 4.2 :

Listing 4.1: Java code for Edgemon task attribution METHOD: addTarget

```
public Set<String> addTarget(String appName, Container container) {
```

```
            Set<String> edgemonTarget = new HashSet<String>();
            edgemonTarget.add(container.getAddress());

      //Header for the JSON object
            Map <String, String> labels = new HashMap<String, String>();
            labels.put("__meta_docker_task_name", "");
            labels.put("__meta_docker_task_desired_state","running");
            labels.put("job", appName);


            RegionEnum region = RegionEnum.getClosestRegion(container.getCoordinates());
            Edgemon edgemon = getRegionEdgemon(region);

            //Get region Edgemons
            List<Edgemon> edgeList = getEdgemons(region);
            for(Edgemon candidate: edgeList){

         //Check if Edgemon candidate is overloaded
                  if(!candidate.getOverload()){
                         edgemon = candidate;
                        break;
                  }
            }
            String swarmNodeId = hostsService.getManagerNode(managerId);
            String containerId = container.getId();
            String edgemonAddr = edgemon.getPublicIpAddress()+":8080";
            String edgemonContainer = edgemon.getContainerId();

      //Creating scrape task for the Edgemon and update Edgemon state
            ScrapeTask scrapeTask = new ScrapeTask(edgemonTarget, labels, swarmNodeId,
                ↪ containerId, edgemonAddr, edgemonContainer);
            edgemon.addTarget(scrapeTask);
            edgemons.save(edgemon);

            Gson gson = new Gson();
            String jsonObject = gson.toJson(scrapeTask);
            log.info("json created");
            byte[] scrapeJsonObject = jsonObject.getBytes();

                  //REST request
                  atributeEdgemonTarget(edgemon, scrapeJsonObject);

            return edgemonTarget;
      }
```

Listing 4.2: Java code for Edgemon task attribution METHOD: atributeEdgemonTarget.

```
public Set<String> atributeEdgemonTarget(Edgemon edgemon, byte[] scrapeJsonObject) {
```

```
            String url = String.format("http://%s:%d/app", edgemon.getPublicIpAddress(),
                ↪ edgemon.getPort());
            try {
                log.info("Sending request {} {} specific edgemon {}", url,
                    ↪ scrapeJsonObject, edgemon.getPublicIpAddress());
                return restTemplate.postForObject(url, scrapeJsonObject, Set.class);
            }
            catch (HttpClientErrorException e) {
                throw new ManagerException(e.getMessage());
            }
        }
```

#### 4.2.3.3 Migration and Replication/Partitioning

In the previous works, the migration process for a monitor was to move the monitor from one node to another in the same swarm, and replication was to create a new replica based on the state and configuration of another monitor. This was to avoid the reinsertion of the same rules and microservices in the new instance. But this view was changed, because now we have a manager that controls where the microservice will be launched, and because of it a node hardly gets fully overloaded. After an alert is triggered the monitor becomes not usable (ensured by a variable) and when looking for capable monitors it will not be available. They are distributed across the region nodes at the launch. Nevertheless, the two notions of migration and replication are maintained, but at a microservice level. Migration in this work is the process of moving all the tasks from one Edgemon to another, leaving one empty and one operational. Most of the time is done to eliminate the first and if there is none to receive the tasks, one is created. Replication maintains its notion, it is when an Edgemon gets or is starting to get overloaded, the mechanism divides the scrape task list and removes the second half. Then initiates a cycle to attribute the microservices to another Edgemon and again, if there is none, it creates one. The alert is sent from the Edgemon module itself, then the handler at the Edgemon service will read the alert (listing 4.3 and 4.4) and react based on it. These mechanisms are ensured using the endpoints available in the Edgemon API (Annex IV).

Listing 4.3: Java code for Edgemon Mechanism handler METHOD: loadMechanisms

```
public void loadMechanisms(String alertType, String containerId){

            //Target edgemon, it will be partitioned or its services migrated
            Edgemon edgemon = getEdgemon(containerId);
            if(edgemon != null){
                log.info("nao e null: ");
            }
```

```
Set<ScrapeTask> scraped = new HashSet<>();
Set<ScrapeTask> firsthalf = new HashSet<>();
Set<ScrapeTask> secondhalf = new HashSet<>();
scraped = edgemon.getTargets();

if(scraped.size() > 1){
	if(alertType.equals("migrate")){
		firsthalf = scraped;
	}
	else {
		int i = 0;
		int count = scraped.size()/2;
			for(ScrapeTask task : scraped){

				log.info("task is: {}", task);
				if(i < count){
				    firsthalf.size());
					firsthalf.add(task);
				}
				else {
					secondhalf.add(task);
				}
				i++;
			}
		log.info("size of firsthalf : {} ",firsthalf.size());
		log.info("size of secondhalf : {} ",secondhalf.size());
	}

	Gson gson = new Gson();
	String jsonObject = gson.toJson(firsthalf);
	byte[] scrapeJsonObject = jsonObject.getBytes();

	//Delete the list of scraped targets from the edgemon
	String url = String.format("http://%s:%d/delete/scrapedtargets",
	    ↪ edgemon.getPublicIpAddress(), edgemon.getPort());
	try {
		log.info("{} targets deleted from specific edgemon {}:{}",
		    ↪ alertType, edgemon.getPublicIpAddress(), edgemon.
		    ↪ getPort());
		restTemplate.postForObject(url, scrapeJsonObject, Set.class);
		log.info("Firsthalf removed from edgemon scrape list : {} ",
		    ↪ edgemon.getOverload());
		edgemon.changeOverloaded();
		log.info("new overload is: {}", edgemon.getOverload());
		log.info("Overload changed");
		edgemon.setTarget(secondhalf);
		edgemons.save(edgemon);
	}
	catch (HttpClientErrorException e) {
		throw new ManagerException(e.getMessage());
```

```
                }
            //Assign the firsthalf os scrape tasks to new Edgemon
            assignNonScraped(firsthalf, edgemon);
        }
    }
```

Listing 4.4: Java code for Edgemon Mechanism handler METHOD: assignNonScraped

```
public void assignNonScraped(Set<ScrapeTask> nonScrapedTargets, Edgemon edgemon){

        //Calculate the region from the nonscrapped service
        RegionEnum region = edgemon.getRegion();
        List<Edgemon> edgemonsFromRegion;
        boolean attributed;
        Gson gson = new Gson();

        for(ScrapeTask nonAssigned : nonScrapedTargets){

            edgemonsFromRegion = getEdgemons(region);
            attributed = false;

            for(Edgemon e : edgemonsFromRegion){

                log.info("Edgemon␣{}:{}␣under␣evaluation:␣", e.
                    ↪ getPublicIpAddress(), e.getPort());
                if(!e.getOverload()){

                    nonAssigned.setEdgemonContainerId(e.getContainerId());
                    nonAssigned.setEdgemonAddr(e.getPublicIpAddress());
                    e.addTarget(nonAssigned);
                    edgemons.save(e);
                    attributed = true;
                    log.info("Non␣Scraped␣Target␣{}␣in␣region␣{}␣was␣
                        ↪ atributted␣to␣edgemon␣{}:{}", nonAssigned.
                        ↪ getTargets(), region, e.getPublicIpAddress(), e.
                        ↪ getPort());
                    break;
                }
            }
            if(!attributed){


                Container container = containersService.getContainer(
                    ↪ nonAssigned.getEdgemonContainerId());
                HostAddress hostAddress = container.getHostAddress();

                //Creates new edgemon in this region
                int availablePort = hostsService.findAvailableExternalPort(
                    ↪ hostAddress, port);
```

```
                    log.info("Port available found: {}", availablePort);
                    Edgemon newEdgemon = launchEdgemon(hostAddress, availablePort);
                        ↪
                    String containerId = newEdgemon.getContainerId();
                    containersService.addContainer(containerId);

                    nonAssigned.setEdgemonContainerId(containerId);
                    nonAssigned.setEdgemonAddr(newEdgemon.getPublicIpAddress());

                    String jsonObject = gson.toJson(nonAssigned);
                    byte[] scrapeJsonObject = jsonObject.getBytes();
                    log.info("json created in assignNonScraped");


                    newEdgemon.addTarget(nonAssigned);
                    atributeEdgemonTarget(newEdgemon, scrapeJsonObject);
                    edgemons.save(newEdgemon);
                    saveToFile(newEdgemon);
                    log.info("New edgemon created as {}:{}, and service {}
                        ↪ assigned", newEdgemon.getPublicIpAddress(), newEdgemon.
                        ↪ getPort(), nonAssigned.getServiceId());
                }
            }
            log.info("All targets have been attributed");
        }
```

### 4.2.4 Edgemon Component

Edgemon is one of the most important pieces of the architecture, by being responsible for the monitorization of the microservices. It collects and temporally stores different kinds of metrics where some of them are used to Edgemons' adaption mechanisms, and others to transmit to managers so they can adapt as well. It uses Docker to obtain metrics about the microservices containers, nodes, and itself and can use the node exporter to obtain host type metrics. The endpoints to communicate with it are shown in annex IV. Accordingly, to the metrics collected from itself and the other targets, Edgemon sends alerts when rules are triggered to the local manager who reconfigures the system. For example, when the resources exceed a certain value it may mean that the node is overloaded and then activates the replication or the partitioning mechanism alert handler in the managers using the API. To communicate with its manager, not only can it use its API but also has stored the address and port of the manager to facilitate communications.

### 4.2.5 Prometheus

CGSM (3.3) also had the work of launching the Edgemons of the system. With the changes, Edgemons are more independent than they used to be. In every region, the local manager has control of what Edgemons exist, and provides to the Prometheus the capability to

maintain the control and scrape tasks over the Edgemons. A signal is sent to the master manager from the local manager whenever an Edgemon is created. This happens to keep the Prometheus target file updated, as now, it is the master manager who has the job to keep this file updated. It is used a yml file to define the yml file where the Edgemons are stored as we can see in listing 4.5 and 4.6.

Listing 4.5: YML file to define the scrape file

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'edgemons'
    scrape_interval: 5s
    file_sd_configs:
      - files:
        - 'swarm-endpoints_new.yml'
```

Listing 4.6: YML file swarm-endpoints_new saving Edgemons

```
[ {
"targets" : [ "XXX.XXX.X.XXX:PPPP" ],
  "labels" : {
    "job" : "XXX.XXX.X.XXX"
  }
} ]
```

The method fileRegistry (on listing 4.7) is called with an argument which is the Edgemon to be registered. After reading the existing target file, the program then writes everything again, this time adding the new Edgemon, to certify that there are no repetitions. To do so the file content is represented by JSON objects.

Listing 4.7: Java code for Edgemon task attribution METHOD: fileRegistry

```
public void fileRegistry(String newEdgemon) {

 JSONParser jsonParser = new JSONParser();
 List<JSONObject> delivery = new ArrayList<JSONObject>();
 JSONObject jsonObject = new JSONObject();
 ObjectMapper mapper = new ObjectMapper();
 List<String> h = new ArrayList<String>();
 JSONObject p = new JSONObject();
 String path ="/home/pedro/Documents/manager/prometheus/swarm-endpoints_new.yml";
    try {
     //Parsing the contents of the JSON file
     JSONArray jsonArray = (JSONArray) jsonParser.parse(new FileReader(path));
        File oldFile = new File(path);
        oldFile.delete();
```

```
      String t = newEdgemon;
      String[] var = newEdgemon.split(":");

        for(Object object : jsonArray) {

            log.info("jsonArray␣{}␣",jsonArray);

     JSONObject record = (JSONObject) object;
            JSONObject labels = (JSONObject) record.get("labels");
            String job = (String) labels.get("job");
     JSONArray targets = (JSONArray) record.get("targets");

            if(!(newEdgemon.equals(targets.get(0)))){
                  delivery.add(record);
            }
   }

      JSONObject jObject = new JSONObject();

      p.put("job", var[0]);
      jObject.put("labels", p);

      h.add(newEdgemon);
      jObject.put("targets", h);

      log.info("edgemon␣a␣ser␣adicionado:␣␣{}", jObject);
      delivery.add(jObject);

      //When in network mode counter of the number of machines
      networkCounter = delivery.size();

      String json = mapper.writerWithDefaultPrettyPrinter().writeValueAsString(delivery
          ↪ );
      FileWriter file = new FileWriter(path, true);
      file.write(json);
      file.close();

      } catch (FileNotFoundException e) {
     e.printStackTrace();
   } catch (IOException e) {
     e.printStackTrace();
   } catch (ParseException e) {
     e.printStackTrace();
   }
 }
```

# Chapter 5

# Demonstration, validation and experimental evaluation

## 5.1 Functional Tests

This section describes the realized functional tests where it is specified the objectives, the scenery and procedures and the results and their interpretation. All the tests are made in an automatic way except when changing the alert to migration or replication. To do this tests, two physical machines in the same network were used as nodes with characteristics as in Figure 5.1. In every scenery it is used one application (Sock-Shop), starting with tree services.

Sock shop is a microservice demo application. The goal of this application is to simulate user-facing part of an website that allegedly sells socks. Its intention is to aid the demonstration and testing of microservices as well as cloud native technologies. In our thesis we use it as the microservices in all the project. It is available a big set of microservices that we defined in the program database.

### 5.1.1 System Baseline - Test 1

#### 5.1.1.1 Objective

The objective of this test is to confirm and understand the mechanisms of creation and setting up both monitorization and management systems, as the manager and worker node. Evaluate the time to prepare the manager and worker nodes with all the components including the Edgemon, the setup of the cluster, the Edgemon launching time in the remote machine, as well as the time to register both machine's monitorization components in the Prometheus target file.

| Hardware | | | | |
|---|---|---|---|---|
| | Architecture | CPU op-mode(S) | CPU(s) | Model name |
| Machine A | x86_64 | 32-bit, 64-bit | 4 | Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz |
| Machine B | x86_64 | 32-bit, 64-bit | 12 | Intel(R) Core(TM) i7-8750H CPU @ 2.50GHz |

Figure 5.1: Detailed hardware from the used machines.

#### 5.1.1.2 Scenery

It is created a swarm with two nodes (manager and worker). Manager node is set up first with all the components and registries itself, then prepares the cluster to add the second node. After launching the second node and its components are initialized a request is sent to the manager node to be registered.

#### 5.1.1.3 Procedure

1. Start to Setup manager node.
2. Launch Edgemon in manager node.
3. Launch support containers.
4. Registry the Edgemon in the manager node.
5. Setup Cluster.
6. Setup worker node.
7. Launch Edgemon in worker node.
8. Registry the Edgemon in the worker node.

#### 5.1.1.4 Results

Looking at the Table 5.1 it is possible to see two important aspects, the difference between the local node (1) and remote node Edgemon launch (2), and writing to the targets' file from a remote node (5) and a local node (6). Writing to a file is made through an HTTP request, and because of it, remote nodes will take more time because of the distance to the master node nevertheless in the table its the opposite due to the fact that when launching the master node (local node) the target file has to be created. Because of the complexity of the master node, it is expected that it will take more time. At the launch of all the

| Parameter | First | Second | Third | AVG(s) |
|---|---|---|---|---|
| (1) Launch Edgemon(Local node) | 12.184 | 14.089 | 13.264 | 13.179 |
| (2) Cluster nodes setup | 1.494 | 1.325 | 1.371 | 1.397 |
| (3) Launch Edgemon(Remote node) | 3.09 | 3.319 | 3.211 | 3.207 |
| (4) Launch all components | 28.154 | 26.7 | 37.126 | 30.66 |
| (5) Write Targets to file (From Remote node) | 0.155 | 0.147 | 0.185 | 0.162 |
| (6) Create Targets file (From Local node) | 1.319 | 0.811 | 2.617 | 1.582 |

Table 5.1: Table to test the initial state of the new architecture.

components (4), besides setting up the cluster (2), it is presented the worst case which means some of the components of the sock shop application have to be downloaded from the online repository. In the system database it is defined which microservices will be downloaded. The creation of the file also takes a little bit more time due to the fact that before writing every Edgemon is passed into a data structure, as well as the existent file if any, so we can eliminate the doubles.

### 5.1.2 Application Attribution - Test 2

#### 5.1.2.1 Objective

This test is to understand the time of launching an application and see their services labeled as scrape tasks. Assigned to an Edgemon, that start collecting their metrics.

#### 5.1.2.2 Scenery

The application is launched in a system where there are two hosts in two different machines, the master node treated by the local and an edge node representing the Remote. When the application is launched the microservices are divided through the two hosts. After launching, all the microservices are attributed to an Edgemon, which can be from the same or from a different machine. Metrics collections start as soon as the services are added to the Edgemons as new tasks. In this test the launch of an application is done in the master manager node using the browser graphical interface. All the procedure are steps that automatically happen after the launch of the application. The Figure 5.2 show the scenery in which this test was made.

#### 5.1.2.3 Procedure

1. Launch the microservices in the hosts.
2. Select the Edgemon to be used.
3. Attribute the microservice using HTTP requests (It can be a remote or local node).
4. Collect metrics of the attributed service.

#### 5.1.2.4 Diagram

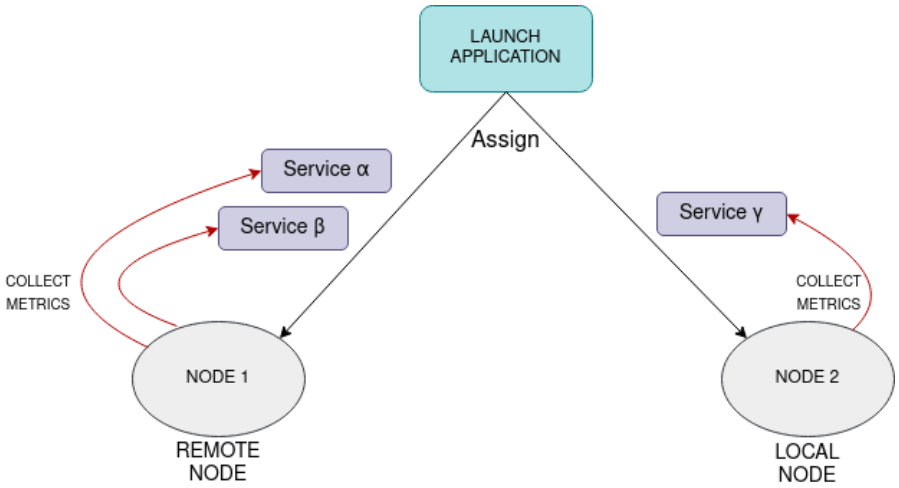


Figure 5.2: Detailed view of the Assign test.

#### 5.1.2.5 Results

| Parameter | First | Second | Third | AVG(s) |
|---|---|---|---|---|
| (1) Launch a Microservice | 4.319 | 4.759 | 4.381 | 4.486 |
| (2) Assign Scrape Target to Local node | 0.051 | 0.082 | 0.071 | 0.068 |
| (3) Assign Scrape Target to Remote node | 0.709 | 0.556 | 0.289 | 0.518 |
| (4) Total Assign Time (After download) | 0.342 | 0.182 | 0.228 | 0.251 |
| (5) Total Assign Time (Before download) | 22.232 | 22.412 | 22.003 | 22.216 |
| (6) Collect Local metrics | 0.038 | 0.029 | 0.028 | 0.032 |
| (7) Collect Remote metrics | 0.383 | 0.419 | 0.428 | 0.41 |

Table 5.2: Table to test Application Attribution.

The most important aspect looking to table 5.2 to be taken into account is the time to assign scrape targets from a local node (2) to a remote node (3). Here, local node time means that the target is running in the same machine as the monitor, while "Assign Scrape Target to Remote node" means that the target is running on another machine, different from the monitor that is being associated with. Thus justifying why to assign to remote nodes are significantly longer than local. This also happens looking at the metrics collecting times, collecting from the same machine (6) consumes a lot less time than from a remote one (7).

### 5.1.3 System Mechanisms (Replication) - Test 3

#### 5.1.3.1 Objective

Aiming to test the replication, this test is to see how the system mechanisms work when triggered. The replication handler and the re-attribution of the microservices are included

while taking the times. Replication as a mechanism on this project is to take some load over a monitor, so it divides the workload (targets) from one monitor to the rest of them.

#### 5.1.3.2 Scenery

For this scenery (Figure 5.3 ), the application is launched with a number of services that can overload a monitor, as defined by an predefined alert rule. So we start in a position where monitors (Edgemons) A and B control microservices. Then we start attributing services to one monitor so the overload alert is raised. After the alert, half of the microservices of A will go to monitor B. This is one of the mechanisms used to manage the monitor system overload.

#### 5.1.3.3 Procedure

1. Attribute to a monitor one more microservice using HTTP requests (It can be a remote or local node).
2. One of the requests raises a replicate alert.
3. Replicate handler triggered.
4. Monitor defined as overloaded.
5. Choosing new monitor to be responsible for half the services.
6. Attribute the microservice using HTTP requests (It can be a remote or local node).
7. Start collecting metrics from the services by the new monitor.

#### 5.1.3.4 Diagram

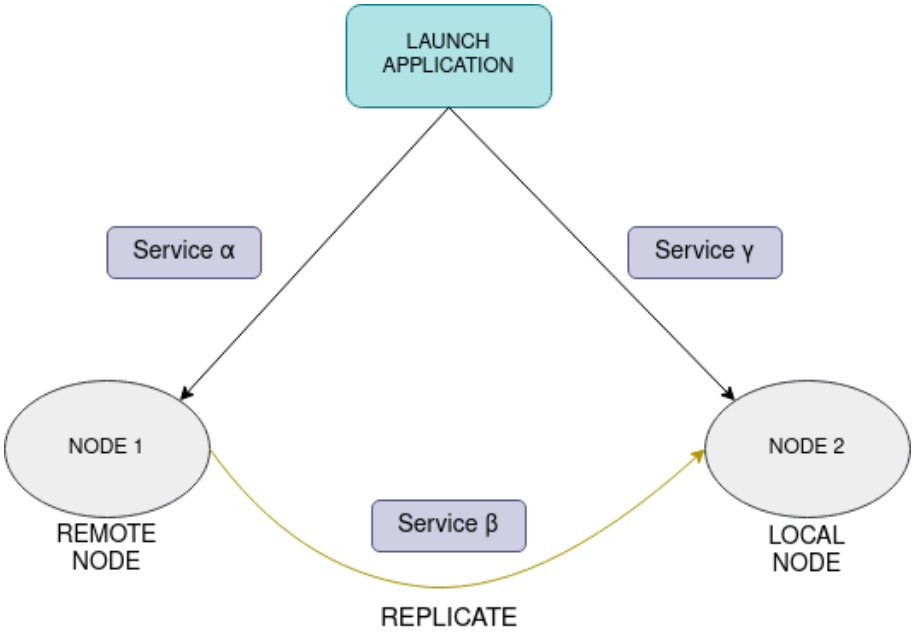


Figure 5.3: Detailed view of the Replication test.

#### 5.1.3.5 Results

| Parameter | First | Second | Third | AVG(s) |
|---|---|---|---|---|
| (1) Attribute the microservice | 0.051 | 0.082 | 0.071 | 0.068 |
| (2) Replicate handler | 0.202 | 0.096 | 0.041 | 0.113 |
| (3) Re-Attribute the microservice to new monitor | 0.084 | 0.061 | 0.021 | 0.055 |
| (4) Replication process | 0.483 | 0.418 | 0.289 | 0.397 |
| (5) Total attribution time of replicated microservice | 0.709 | 0.556 | 0.289 | 0.518 |
| (6) Total program time | 1.251 | 1.067 | 0.66 | 0.993 |

Table 5.3: Table to test System Mechanisms (Replication).

So the Table 5.3 starts with the time to attribute a microservice that does not fulfill the requirements to trigger the replication mechanism (1). Then the replication comes in, the replication handler (2) is only the process of choosing the replication as the mechanism to be used. The replication process (4) is the time to choose the new monitor and attribute the service (3). For the test we talk in one service only because the main feature is to choose the new monitor, the services will be passed as list. Point (5) refers to the time to attribute a service that raised a alert, and was attributed to a new monitor. Point (6) represents the time of the program to attribute services including all the replication processes, which means after launching the microservice, raising the alert, going through the handler, and finally re-attribute to a new monitor.

### 5.1.4 System Mechanisms (Migration) - Test 4

#### 5.1.4.1 Objective

This test is used to test the migration. The migrations on this test will take all the load of a monitor to another one, the closest one. If it is not found it will create one and attribute them to it. As described in section 4.2.3.3, both migration and replication are, in this thesis, at a microservice level.

#### 5.1.4.2 Scenery

In this test (Figure 5.4 ) the used mechanism is migration. After launching a program, one of the microservice will raise an alert. This mechanism will choose a new monitor, ideally the closest one, and will pass all the services to it. Thus taking all the load from monitor A to monitor B.

#### 5.1.4.3 Procedure

1. Attribute the microservice using HTTP requests (It can be a remote or local node).
2. One of the requests raises an migration alert.
3. Migration handler triggered.
4. Monitor defined as overloaded.
5. Choosing new monitor to be attributed, if there is none, a new one is created.
6. Attribute the microservices using HTTP requests (It can be a remote or local node).
7. Collect metrics of the attributed service.

#### 5.1.4.4 Diagram

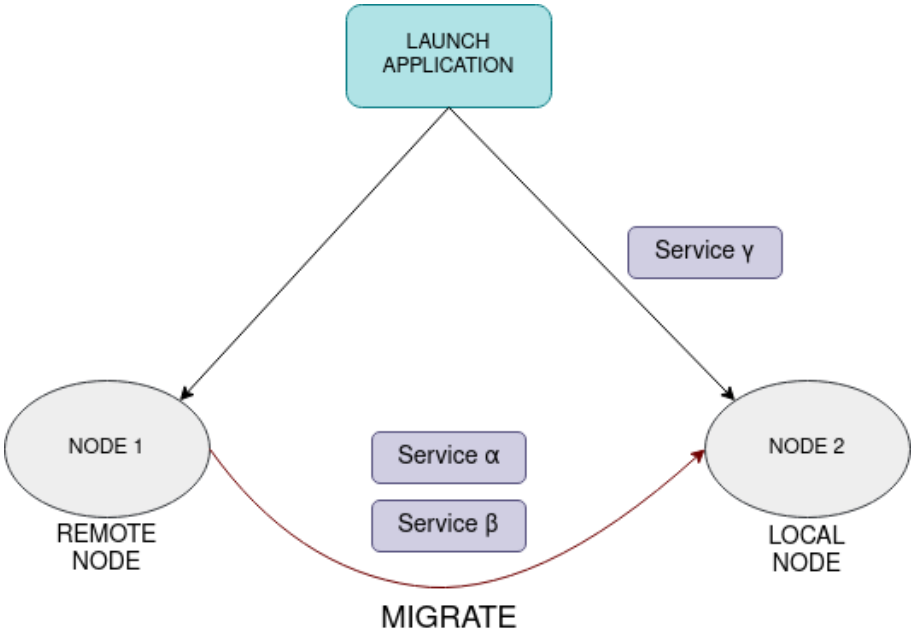


Figure 5.4: Detailed view of the Migration test.

#### 5.1.4.5 Results

| Parameter | First | Second | Third | AVG(s) |
|---|---|---|---|---|
| (1) Attribute the microservice to remote monitor | 0.434 | 0.59 | 0.656 | 0.56 |
| (2) Migration handler | 0.135 | 0.202 | 0.091 | 0.143 |
| (3) Migration process | 0.354 | 0.472 | 0.335 | 0.387 |
| (4) Re-Attribute the microservice to new monitor | 0.105 | 0.17 | 0.077 | 0.117 |
| (5) Total program attribution time with Migration | 0.929 | 1.239 | 1.108 | 1.092 |

Table 5.4: Table to test System Mechanisms (Migration).

The Table 5.4 begins with the time to attribute a microservice to a remote monitor not fulfilling the requirements to trigger the migration mechanism (1). Then the migration is launched, and the handler chooses migration as the mechanism to be used (2). Process (3) is the time to select the new monitor and give away all the services. Point (4) refers the time to attribute services, when the trigger is activated, to a new monitor. Point (5) represents the total program time, including all the migration processes.

It is important to understand that the migration mechanism has a special feature because when the migration alert is triggered and none of the existing monitors is non-overloaded the system creates one.

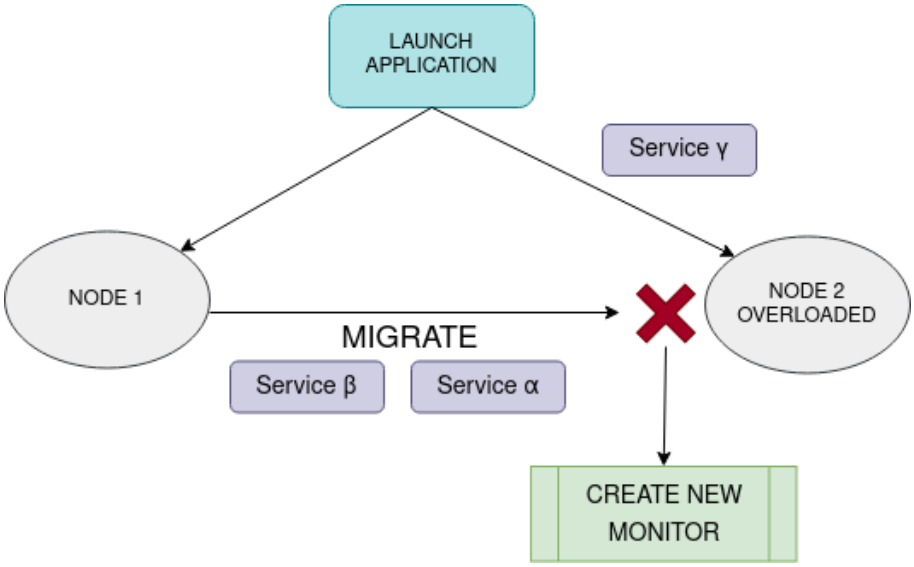


Figure 5.5: Detailed view of the Migration test with creation of a new monitor.

| Parameter | First | Second | Third | AVG(s) |
|---|---|---|---|---|
| (6) After alert, Creation and Attribution | 4.632 | 3.243 | 3.719 | 3.775 |

Table 5.5: Table to test System Mechanisms- Migration (Create new monitor).

The Table 5.5 time (6) represents the time to create a new Edgemon and attribute the services. It is expected to take a longer time, nevertheless, we can not forget that it is not going to happen every time, with every monitor, and when it does happen it will give away all the services that it is currently monitoring.

In conclusion, besides the merge made between the two systems, important aspects were ensured and maintained. It was reduced the number of targets of the Prometheus reducing its resource usage. Makes sense to use Edgemons as the Prometheus targets because they already have in place mechanisms and alerts to monitor the services launched. The master node takes more time in some operations as expected to control the rest of the system.

## 5.2 Performance Tests

In this section, the tests are to verify the behavior and use of resources by the monitor. It is not necessary to test in different scenarios because they were already tested [9] only in a medium or high-stress environment and different location nodes. So the tests were done using an application launched on a remote and local machine. The two aspects tested were the CPU and the RAM because in edge environments these two are the ones that vary the most throughout edge resources. An edge context is where the resources and the latency matter and the solution aims to be lightweight with fewer necessities in every moment.

| Component | Local node | Remote node |
|---|---|---|
| Edgemon (monitor) | 0.14 | 0.06 |
| Prometheus | 0.37 | 0.16 |
| Request Location | 0.03 | 0.01 |
| Registration server | —- | 6.45 |
| Nginx-basic | 0.06 | 0.06 |

Table 5.6: Table to test Memory in the node in percentage.

As expected, showed in table 5.6, the memory consumption will grow as more services are added and monitored and more metrics are collected. The number of objects in the memory will also increase because of that. The most important to see is that the remote node can have a lot less space capacity than the master node, favoring the fact that on the edge we can use less capable nodes.

| Component | First | Second | Third | Fourth | AVG(s) |
|---|---|---|---|---|---|
| Edgemon (monitor) | 2.94 | 9.68 | 13.55 | 7.23 | 8.35 |

Table 5.7: Table to test CPU in the Local node (Master node) in percentage.

| Component | First | Second | Third | Fourth | AVG(s) |
|---|---|---|---|---|---|
| Edgemon (monitor) | 0.61 | 0.35 | 0.02 | 0.39 | 0.343 |

Table 5.8: Table to test CPU in the Remote node (worker node) in percentage.

Regarding the CPU (Tables 5.7, and 5.8) it is important to notice that there are some spikes of CPU usage while executing and with more services the percentage is higher. The spikes are mainly because of the instructions for collecting and storing metrics and the mechanism of adaptation.

The aspects that matter in these tests are the fact that we can see a difference when talking about the remote (table 5.8) and local nodes (table 5.7). Almost in every aspect, the remote node consumes fewer resources. This is very important because we want to spend the less possible resources from the edge nodes because they are not only unknown, less capable, and less controllable but also unpredictable in the sense that we can not predict that they will all be the same.

Chapter 6

# Conclusions and future work

When we have an execution environment such as cloud, edge with different devices, it is obvious that we are facing a complex hybrid environment, with not only difficult management but also difficult monitorization. Namely, we have the Cloud that allows us to store and process large amounts of data, while edge nodes are heterogeneous and geographically dispersed and where aspects such as latency are crucial.

Numerous problems may arise in applications in these contexts, specifically applications with a large number of microservices that may fail and that require enough nodes for their deployment. When talking about nodes, they can all be different in system or resources and they are widely geographically dispersed. More data is also being transferred from the periphery to the cloud, which increases latency and reduces the service quality. Last but not least cloud environments have costs and the network bandwith and latency are limited, so it is a lot better if, in some way, the data transferred to it is minimized. For these reasons, it was necessary to build a system that could take advantage of the edge and cloud environments, manage it in the best way and may alert the end-user on certain actions. Not only manage but also monitor the system is fundamental for the management itself, and in a way that could read the necessities and try, with the management components, to adapt the system, which has different builds. All to continuously keep the services going.

This was a Thesis that considered three previous works to give the first step to achieve a fully adaptable hybrid management and monitorization system.

The first work studied is focused on the monitorization components, which use the edge nodes' computational resources, closer to the end-users, to harbor light monitors. It is interesting work because not only reduces the network traffic but contributes to reduce the computational and energetic costs of the cloud. The light monitors may filter the data on the edge reducing the latency and by having a faster alert mechanism. By being in the edge, the light monitors decrease the time to deliver metrics in comparison with the

cloud.

The second work comprehends the automatic manager system. It was also a well-achieved work that takes advantage of the edge machines to harbor services through the management of nodes and containers. This management system uses metrics analyses, rules, and decisions to either start, stop, migrate or replicate containers in the nodes. It was created with a web application accessible through the browser to control and see the system state. The work has a central component named the master manager that has the vision of the whole system to take advantage of the nodes' computing resources and localization.

The integration of both works, as presented in this proposal, resulted in a prototype that takes advantage of some of the components created in the previous works. Now the local managers from the automatic management system are more complex because they have associated a new monitor service. Every time a manager is launched, which is one per region, a monitor is launched with it. In case it is required more than one monitor per region, the system selects a capable node in the region. Because the master manager continues to have the vision of the whole system, it receives a request to register the new monitors. This registration is made so that the Prometheus file-based discovery can collect metrics from the monitors instead of going to all the application services. As before, reducing central Prometheus load and distributing the detection of the alert situations through the edgemons.

By having the monitorization service as part of a manager from the automatic management system, it is not necessary to have the discovery module created in the previous works, since we can associate a monitor to a manager at launch time. Managers (management and monitor components) can use the mechanisms of replication and migration not only in nodes but also in monitors to improve the efficiency of metrics' collection. Edgemons, not only collect metrics about the microservices but they share them with the managers and send alerts.

Therefore the developed prototype presented in this work allows for the monitoring of hybrid environments combining heteregeneous resources of both cloud and edge nodes, supporting an efficient, agile, and adaptable monitorization of microservices applications deployed in both regions. Also it allows to manage nodes and monitors by reading metrics, using mechanisms to improve not only the efficiency of the system but reduce latency, and data transfer.

In conclusion, by including the monitor component into the management system in a way that it can adapt and monitor the nodes, applications, and containers, we can conclude that the initial objectives were reached. Nevertheless, the text below describes some features and functionalities that are not present or can/need to be more developed.

## 6.1 Future Work

It is possible to identify some aspects that can and need to be improved for the system to be completely and fully capable:

- Add aggregation functions in the Edgemon: This was already mentioned in previous work. Using functions, it is possible to create metrics that sum up or filter sets of collected metrics. This would reduce the network traffic, reduce the computational load, and reduce the occupied space in the monitors' storage.
- Authentication Mechanism: To protect APIs, it is necessary to improve the security, especially if talking about the cloud environment. There is a basic authentication in place, but some APIs like the Edgemon one does not have it. So the resources can be manipulated or even erased. A mechanism of user authentication would make the system benefit.
- Make more use of Edgemon collected metrics: With the union of the two projects, we have an Edgemon component capable of collecting not only application-level metrics but also container and node metrics. It is possible to take advantage of all these collected metrics and improve the system's adaptability. Better sharing with the manager, Prometheus, or even with other managers could help define where is the best place to launch services for example.
- Migrate and replicate to other regions: Another problem that still exists is that the regions are isolated, so the microservices have to be in the same region to work properly. Now not only microservices can not cross regions but the monitors also can not do it.
- A multi regional and cloud evaluation: On this dissertation it was not evaluated the project while running in a multi regional and cloud environment. It is important to take this perspective because, while it was tested in multiple machines, the final goal of the whole system is to run across both worlds doing the management and the monitorization of microservices through the collection of metrics and cross examination with the defined rules.

# Bibliography


[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[2] L. Chen. "Microservices: Architecting for Continuous Delivery and DevOps". In: *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*. 2018. ISBN: 9781538663981. DOI: 10.1109/ICSA.2018.00013 (cit. on p. 1).

[3] T. B. of Continuous Delivery: An Overview. URL: https://puppet.com/blog/top-benefits-of-continuous-delivery-an-overview/. (accessed: 21.07.2020) (cit. on p. 1).

[4] A. V. Dastjerdi et al. "Fog Computing: Principles, architectures, and applications". In: *Internet of Things: Principles and Paradigms* (2016), pp. 61–75. DOI: 10.1016/B978-0-12-805395-9.00004-6. arXiv: 1601.02752 (cit. on pp. 1, 16, 17).

[5] P. M. Mell and T Grance. *The NIST definition of cloud computing*. Tech. rep. Gaithersburg, MD: National Institute of Standards and Technology, 2011. DOI: 10.6028/NIST.SP.800-145. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (cit. on pp. 2, 7–9, 12).

[6] W. Shi and S. Dustdar. "The Promise of Edge Computing". In: *Computer* (2016). ISSN: 00189162. DOI: 10.1109/MC.2016.145 (cit. on pp. 2, 13).

[7] M Iorga et al. "Fog Computing Conceptual Model, Recommendations of the National Institute of Standards and Technology". In: *NIST Special Publication* (2018) (cit. on pp. 2, 13–15).

[8] A. V. Carrusca. "Gestão de micro-serviços na Cloud e Edge". In: *FCT: DI - Dissertações de Mestrado* (2018) (cit. on pp. 3, 27, 33–35).

[9] P. Correia. "Monitorização para suporte de auto-gestão em aplicações de micro-serviços". In: *FCT: DI - Dissertações de Mestrado* (2019) (cit. on pp. 4, 27–29, 32, 42–44, 46, 49, 73).

[10] D. F. S. Pimenta. "Gestão Dinâmica de Micro-serviços na Cloud/Edge". In: *FCT: DI - Dissertações de Mestrado* (Feb-2021) (cit. on pp. 4, 27, 37–39, 41, 42, 45, 49).

[11] D. C. Marinescu. *Cloud Computing: Theory and Practice*. 2013. ISBN: 9780124046276. DOI: 10.1016/C2012-0-02212-0 (cit. on pp. 7, 8, 11, 12).

[12] R. Stair and G. Reynolds. *Principles of information systems*. Cengage Learning, 2015 (cit. on p. 7).

[13] I. Mitrani. "Managing performance and power consumption in a server farm". In: *Annals of Operations Research* (2013). ISSN: 02545330. DOI: 10.1007/s10479-011-0932-1 (cit. on p. 7).

[14] J. Timmermans et al. "The ethics of cloud computing: A conceptual review". In: *Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010*. 2010. ISBN: 9780769543024. DOI: 10.1109/CloudCom.2010.59 (cit. on p. 7).

[15] G. Feuerlicht and S. Govardhan. "Impact of cloud computing: Beyond a technology trend". In: *Systems Integration* (2010) (cit. on p. 8).

[16] A. Choudhary, S. Rana, and K. J. Matahai. "A Critical Analysis of Energy Efficient Virtual Machine Placement Techniques and its Optimization in a Cloud Computing Environment". In: *Physics Procedia* 78.December 2015 (2016), pp. 132–138. ISSN: 18753892. DOI: 10.1016/j.procs.2016.02.022. URL: http://dx.doi.org/10.1016/j.procs.2016.02.022 (cit. on p. 8).

[17] Q. Zhang, L. Cheng, and R. Boutaba. "Cloud computing: State-of-the-art and research challenges". In: *Journal of Internet Services and Applications* (2010). ISSN: 18674828. DOI: 10.1007/s13174-010-0007-6 (cit. on pp. 8, 9).

[18] P. Barham et al. "Xen and the Art of Virtualization Categories and Subject Descriptors". In: *19th ACM Symposium on Operating Systems Principles*. 2003. ISBN: 1581137575 (cit. on p. 10).

[19] A. Whitaker, M. Shaw, and S. S. D. Gribble. "Denali: Lightweight Virtual Machines for Distributed and Networked Applications". In: *In Proceedings of the USENIX Annual Technical Conference* (2002). DOI: 10.1.1.11.6912 (cit. on p. 10).

[20] W. A. Jansen. "Cloud hooks: Security and privacy issues in cloud computing". In: *Proceedings of the Annual Hawaii International Conference on System Sciences* (2011), pp. 1–10. ISSN: 15301605. DOI: 10.1109/HICSS.2011.103 (cit. on p. 12).

[21] A. V. Dastjerdi and R. Buyya. "Fog Computing: Helping the Internet of Things Realize Its Potential". In: *Computer* 49.8 (2016), pp. 112–116. ISSN: 00189162. DOI: 10.1109/MC.2016.245 (cit. on pp. 12, 13, 15–17).

[22] R. Cortés et al. "Stream processing of healthcare sensor data: Studying user traces to identify challenges from a big data perspective". In: *Procedia Computer Science*. 2015. DOI: 10.1016/j.procs.2015.05.093 (cit. on p. 13).

[23] M. Aazam, S. Zeadally, and K. A. Harras. "Offloading in fog computing for IoT: Review, enabling technologies, and research opportunities". In: *Future Generation Computer Systems* (2018). ISSN: 0167739X. DOI: 10.1016/j.future.2018.04.057 (cit. on p. 13).

[24] J. Leitão et al. "Towards Enabling Novel Edge-Enabled Applications". In: 732505 (2018). arXiv: 1805.06989. URL: http://arxiv.org/abs/1805.06989 (cit. on p. 13).

[25] D. Bermbach et al. "A Research Perspective on Fog Computing". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10797 LNCS (2018), pp. 198–210. ISSN: 16113349. DOI: 10.1007/978-3-319-91764-1_16 (cit. on p. 13).

[26] S. Kosta et al. "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading". In: *Proceedings - IEEE INFOCOM*. 2012. ISBN: 9781467307758. DOI: 10.1109/INFCOM.2012.6195845 (cit. on p. 16).

[27] G. Orsini, D. Bade, and W. Lamersdorf. "Computing at the Mobile Edge: Designing Elastic Android Applications for Computation Offloading". In: *Proceedings - 2015 8th IFIP Wireless and Mobile Networking Conference, WMNC 2015*. 2016. ISBN: 9781509003518. DOI: 10.1109/WMNC.2015.10 (cit. on p. 16).

[28] I. Stojmenovic and S. Wen. "The Fog computing paradigm: Scenarios and security issues". In: *2014 Federated Conference on Computer Science and Information Systems, FedCSIS 2014*. 2014. ISBN: 9788360810583. DOI: 10.15439/2014F503 (cit. on p. 17).

[29] P. Misra, Y. Simmhan, and J. Warrior. "Towards a Practical Architecture for the Next Generation Internet of Things". In: (2015). arXiv: 1502.00797. URL: http://arxiv.org/abs/1502.00797 (cit. on p. 17).

[30] M. Aazam and E. N. Huh. "Dynamic resource provisioning through Fog micro datacenter". In: *2015 IEEE International Conference on Pervasive Computing and Communication Workshops, PerCom Workshops 2015*. 2015. ISBN: 9781479984251. DOI: 10.1109/PERCOMW.2015.7134002 (cit. on p. 17).

[31] P. Sharma et al. "Containers and virtual machines at scale: A comparative study". In: *Proceedings of the 17th International Middleware Conference, Middleware 2016*. 2016. ISBN: 9781450343008. DOI: 10.1145/2988336.2988337 (cit. on pp. 18, 20).

[32] E. Casalicchio. "Autonomic orchestration of containers: Problem definition and research challenges". In: *ValueTools 2016 - 10th EAI International Conference on Performance Evaluation Methodologies and Tools*. 2017. ISBN: 9781631901416. DOI: 10.4108/eai.25-10-2016.2266649 (cit. on pp. 19, 21).

[33] Docker Inc. *What is a Container | Docker*. 2017 (cit. on p. 19).

[34] M. Amaral et al. "Performance evaluation of microservices architectures using containers". In: *Proceedings - 2015 IEEE 14th International Symposium on Network Computing and Applications, NCA 2015*. 2016. ISBN: 0769556817. DOI: 10.1109/NCA.2015.49. arXiv: 1511.02043 (cit. on p. 19).

[35] E. Casalicchio and V. Perciballi. "Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics". In: *Proceedings - 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems, FAS*W 2017*. 2017. ISBN: 9781509065585. DOI: 10.1109/FAS-W.2017.149 (cit. on p. 19).

[36] D. Merkel. *Docker: lightweight Linux containers for consistent development and deployment*. 2014. DOI: 10.1097/01.NND.0000320699.47006.a3 (cit. on p. 19).

[37] Y. Al-Dhuraibi et al. "Elasticity in Cloud Computing: State of the Art and Research Challenges". In: *IEEE Transactions on Services Computing* (2018). ISSN: 19391374. DOI: 10.1109/TSC.2017.2711009 (cit. on p. 19).

[38] C. A. Maziero. "Sistemas Operacionais: Conceitos e Mecanismos". In: (2013), p. 356. URL: http://187.7.106.14/emmonks/sos2/so-livro.pdf (cit. on pp. 19, 20).

[39] T. Salah et al. "Performance comparison between container-based and VM-based services". In: *Proceedings of the 2017 20th Conference on Innovations in Clouds, Internet and Networks, ICIN 2017*. 2017. ISBN: 9781509036721. DOI: 10.1109/ICIN.2017.7899408 (cit. on p. 20).

[40] N. Dragoni et al. "Microservices: Yesterday, today, and tomorrow". In: *Present and Ulterior Software Engineering*. 2017. ISBN: 9783319674254. DOI: 10.1007/978-3-319-67425-4_12. arXiv: 1606.04036 (cit. on pp. 21, 23).

[41] W. I. M. I. T. M. Architecture. *https://www.edureka.co/blog/what-is-microservices/*. URL: https://www.edureka.co/blog/what-is-microservices/. (accessed: 28.07.2020) (cit. on p. 21).

[42] S. Daya et al. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks, 2016. ISBN: 9780738440811. URL: https://books.google.pt/books?id=eOZyCgAAQBAJ (cit. on pp. 22, 23).

[43] S. Newman. *Building Microservices*. 2015. ISBN: 978-1-491-95035-7 (cit. on p. 22).

[44] T. D. Stojanovic et al. "Identifying microservices using structured system analysis". In: *2020 24th International Conference on Information Technology, IT 2020*. 2020. ISBN: 9781728151366. DOI: 10.1109/IT48810.2020.9070652 (cit. on p. 23).

[45] D. Shadija, M. Rezai, and R. Hill. "Microservices: Granularity vs. Performance". In: *UCC 2017 Companion - Companion Proceedings of the 10th International Conference on Utility and Cloud Computing*. 2017. ISBN: 9781450351959. DOI: 10.1145/3147234.3148093. arXiv: 1709.09242 (cit. on p. 23).

[46] *10 companies that implemented the microservice architecture and paved the way for others*. URL: https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-others/. (accessed: 21.07.2020) (cit. on pp. 25, 26).

[47] T. Microservices Architectures: Become a Unicorn like Netflix and Hailo. URL: https://www.slideshare.net/gjuljo/microservices-architectures-become-a-unicorn-like-netflix-twitter-and-hailo. (accessed: 21.07.2020) (cit. on p. 26).

[48] Microservice Architecture — Learn, Build, and Deploy Applications. URL: https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a. (accessed: 21.07.2020) (cit. on p. 26).

[49] *Prometheus: From metrics to insight*. URL: https://prometheus.io/ (cit. on p. 27).

[50] A. Lameirinhas. "Monitoring in Hybrid Cloud-Edge Environments". In: *FCT: DI - Dissertações de Mestrado* (2019) (cit. on pp. 27, 28).

ANNEX I

# MASTER MANAGER'S API

Annex 1 represents the API of a master manager. All the endpoints succeed /api.

| Request | Endpoint | Information |
|---|---|---|
| GET | /basicauth | Used for authentication. |
| GET | /local-managers | Returns a list of executing local managers. |
| GET | /local-managers/{id} | Returns the local manager with id: {id}. |
| POST | /local-managers | Initiates a local manager in a region or address emitted through the request body. |
| DELETE | /local-managers/{id} | Stops the local manager with id: {id}. |
| GET | /registration-servers | Returns a list of executing registry servers. |
| GET | /registration-servers/{id} | Returns the registry server with id: {id}. |
| POST | /registration-servers | Initiates a registry server in a region or address emitted through the request body. |
| DELETE | /registration-servers/{id} | Stops the registry server with id: {id}. |
| GET | /load-balancers | Returns a list of executing load balancers. |
| GET | /load-balancers/{id} | Returns the load balancer with id: {id}. |
| POST | /load-balancers | Initiates a load balancer in a region or address emitted through the request body. |
| DELETE | /load-balancers/{id} | Stops the load balancer with id: {id}. |
| GET | /kafka | Returns a list of executing kafka agents. |
| GET | /kafka/{id} | Returns the kafka agent with id: {id}. |

| | | |
|---|---|---|
| POST | /kafka | Initiates a kafka agent in a region or address emitted through the request body. |
| DELETE | /kafka/{id} | Stops the kafka agent with id: {id}. |
| GET | /apps | Returns a list with the applications. |
| GET | /apps/{name} | Returns the application with name: {name}. |
| POST | /apps | Add an application emitted through the request body. |
| PUT | /apps/{name} | Updates the values emitted in the request on the application with name: {name}. |
| DELETE | /apps/{name} | Delete the application with name: {name}. |
| GET | /apps/{name}/services | Returns the list of services of the application with name: {name}. |
| POST | /apps/{name}/services | Associates the services emitted in the request body to the application with name: {name}. |
| DELETE | /apps/{name}/services | Dissociates the services emitted in the requests body from the application with name: {name}. |
| POST | /apps/{name}/launch | Launch every service of the application with name: {name}, in the coordinates emitted in the requests body. |
| GET | /services | Returns a list with the services. |
| GET | /services/{name} | Returns the service with name: {name}. |
| POST | /services | Add a service emitted through the request body. |
| PUT | /services/{name} | Updates the values emitted in the request on the service with name: {name}. |
| DELETE | /services/{name} | Delete the service with name: {name}. |
| GET | /services/{name}/apps | Returns the list of applications associated with the service with name: {name}. |
| POST | /services/{name}/apps | Associates the applications emitted in the request body to the service with name: {name}. |
| DELETE | /services/{serviceName}/apps/{appName} | Dissociates the application with name: {appName} from service with name: {serviceName}. |

| | | |
|---|---|---|
| GET | /services/{name}/dependencies | Returns the list of dependencies of the service with name: {name}. |
| POST | /services/{name}/dependencies | Add the dependencies emitted through the requests body to the service with name: {name}. |
| DELETE | /services/{serviceName}/dependencies | Removes the dependencies emitted through the requests body from the service with name: {name} to the service with name: {name}. |
| GET | /services/{name}/dependents | Returns a list of the dependents of the service with name: {name}. |
| GET | /services/{name}/predictions | Returns the predictions done to the service with name: {name}. |
| POST | /services/{name}/predictions | Add a prediction emitted through requests body to the service with name: {name}. |
| DELETE | /services/{name}/predictions | Remove all the predictions, emitted through the requests body, done to the service with name: {name}. |
| DELETE | /services/{serviceName}/predictions/{predictionName} | Removes the prevision with name: {predictionName} from the service with name: {serviceName}. |
| GET | /containers | Return the list of all the containers of the system. |
| GET | /containers/{id} | Return the container with id: {id}. |
| POST | /containers/sync | Sync the stored containers data with the information from all the managers docker swarms. |
| GET | /containers/{id}/logs | Return the logs from container with id: {id}. |
| GET | /hosts/edge | Return a list with the edge hosts. |
| GET | /hosts/edge/{publicIp}/{privateIp} | Return the edge host with public IP: {publicIp} and private IP: {privateIp}. |
| POST | /hosts/edge | Add the host emitted in the requests body. |
| PUT | /hosts/edge/{publicIp}/{privateIp} | Changes the values emitted through the requests body from the host with public IP: {publicIp} and private IP: {privateIP}. |
| DELETE | /hosts/edge/{publicIp}/{privateIp} | Erases the host with public IP: {publicIp} and private IP: {privateIp}. |
| GET | /hosts/cloud | Returns a list of instances running in the cloud. |

| | | |
|---|---|---|
| GET | /hosts/cloud/{instanceId} | Returns the instance in the cloud with id: {instanceId}. |
| POST | /hosts/cloud | Initiate and configure an instance near the coordinates emitted through the request body.. |
| PUT | /hosts/cloud/{instanceId}/start | Initiate an instance with id: {instanceId}. |
| PUT | /hosts/cloud/{instanceId}/stop | Stops an instance with id: {instanceId}. |
| DELETE | /hosts/cloud/{instanceId}/terminate | Delete an instance with id: {instanceId}. |
| POST | /hosts/cloud/sync | Synchronization of the known cloud instances with the information in the AWS. |
| GET | /hosts/cloud/regions | Returns a list with the supported AWS regions. |
| GET | /nodes | Return a list of every nodes in the system. |
| GET | /nodes/{id} | Return the node with id: {id}. |
| POST | /nodes | Initiate a node in the master manager, or local manager, near a coordinate or in a specific node emitted through the request body. |
| PUT | /nodes/{id} | Updates the state of the node with id: {id} with values emitted through the requests body. |
| DELETE | /nodes/{id} | Remove the node with id : {id}. |
| PUT | /nodes/{publicIp}/{privateIp} | Remove the node in the host with the public IP: {publicIp} and private IP: {privateIp}. |
| POST | /nodes/{id}/join | Join to the swarm the node with id: {id}. |
| POST | /nodes/sync | Synchronize stored node data with all the managers docker swarms information. |
| GET | /fields | Return a list with fields used to formulate conditions. |
| GET | /fields/{name} | Return the field with name: {name}. |
| GET | /component-types | Return a list of component types used in rules and simulated metrics. |
| GET | /component-types/{name} | Return the component type with name: {name}. |
| GET | /rules/conditions | Return a list of the defined conditions. |
| GET | /rules/conditions/{name} | Return the condition with name: {name}. |

| | | |
|---|---|---|
| POST | /rules/conditions | Add a condition emitted through the requests body. |
| PUT | /rules/conditions/{name} | Changes the values of the condition with name: {name}, with values emitted in the request body. |
| DELETE | /rules/conditions/{name} | Erases the condition with name:{name}. |
| GET | /rules/conditions/hosts | Return a list containing the association between conditions and hosts rules. |
| GET | /rules/conditions/{name}/hosts | Return a list of hosts rules associated with the condition with name: {name}. |
| GET | /rules/conditions/apps | Return a list of all the associations between conditions and application rules. |
| GET | /rules/conditions/{name}/apps | Return a list of application rules associated to a condition with name: {name}. |
| GET | /rules/conditions/services | Return a list of associations between conditions and service rules. |
| GET | /rules/conditions/{name}/services | Return a list of service rules associated with the condition with name: {name}. |
| GET | /rules/conditions/containers | Return a list of all the associations between conditions and container rules. |
| GET | /rules/conditions/{name}/containers | Return a list of container rules associated with a condition with name:{name}. |
| GET | /rules/hosts | Return the list of hosts rules. |
| GET | /rules/hosts/{name} | Return the host rule with name: {name}. |
| POST | /rules/hosts | Add host rule contained in the requests body. |
| PUT | /rules/hosts/{name} | Updates the host rule with name: {name} using values emitted in the requests body. |
| DELETE | /rules/hosts/{name} | Erase host rule with name: {name}. |
| GET | /rules/hosts/{name}/conditions | Return a list of hosts rules conditions. |
| POST | /rules/hosts/{name}/conditions | Associate conditions emitted through the requests body to the host rule with name: {name}. |
| DELETE | /rules/hosts/{name}/conditions | Dissociates the conditions emitted in the requests body from the host rule with name: {name}. |
| GET | /rules/hosts/{name}/cloud-hosts | Return a list of the cloud hosts associated with host rule with name: {name}. |

| | | |
|---|---|---|
| POST | /rules/hosts/{name}/cloud-hosts | Associate the cloud hosts emitted through the requests body to the host rule with name: {name}. |
| DELETE | /rules/hosts/{name}/cloud-hosts | Dissociates the cloud hosts emitted through the requests body from the host rule with name: {name}. |
| GET | /rules/hosts/{name}/edge-hosts | Return a list of edge hosts associated to the rule with name: {name}. |
| POST | /rules/hosts/{name}/edge-hosts | Associates the edge hosts emitted through the request body to the host rule with name: {name}. |
| DELETE | /rules/hosts/{name}/edge-hosts | Dissociates the edge hosts emitted through the request body from the host rule with name: {name}. |
| GET | /rules/apps | Return a list with application rules. |
| GET | /rules/apps/{name} | Return the application rule with name: {name}. |
| POST | /rules/apps | Add the application rule emitted through the request body. |
| PUT | /rules/apps/{name} | Updates the application rule with name: {name} with values emitted through the request body. |
| DELETE | /rules/apps/{name} | Erases the application rule with name: {name}. |
| GET | /rules/apps/{name}/conditions | Return a list with the conditions form the application rule with name: {name}. |
| POST | /rules/apps/{name}/conditions | Associates the conditions emitted through the request body to the application rule with name: {name}. |
| DELETE | /rules/apps/{name}/conditions | Dissociates the conditions emitted through the request body from the application rule with name: {name}. |
| GET | /rules/apps/{name}/apps | Return a list of the applications associated to the application rule with name: {name}. |
| POST | /rules/apps/{name}/apps | Associates the applications emitted through the request body to the application rule with name: {name}. |

| | | |
|---|---|---|
| DELETE | /rules/apps/{name}/apps | Dissociates the applications emitted through the request body from the application rule with name: {name}. |
| GET | /rules/services | Return a list of the services rules. |
| GET | /rules/services/{name} | Return the rules of a service with name: {name}. |
| POST | /rules/services | Add a service rule emitted through the request body. |
| PUT | /rules/services/{name} | Updates the service rule with name: {name} with values emitted through the request body. |
| DELETE | /rules/services/{name} | Erases the rule from service with name: {name}. |
| GET | /rules/services/{name}/conditions | Returns the rules conditions list from the service with name: {name}. |
| POST | /rules/services/{name}/conditions | Associates the conditions emitted through the request body to the service rule with name: {name}. |
| DELETE | /rules/services/{name}/conditions | Dissociates the conditions emitted through the request body from the service rule with name: {name}. |
| GET | /rules/services/{name}/services | Return a list of the services associated to the service rule with name: {name}. |
| POST | /rules/services/{name}/services | Associates the services emitted through the request body to the service rule with name: {name}. |
| DELETE | /rules/services/{name}/services | Dissociates the services emitted through the request body from the service rule with name: {name}. |
| GET | /rules/containers | Return a list of the containers rules. |
| GET | /rules/containers/{name} | Return the container rule with name: {name}. |
| POST | /rules/containers | Add the container rule emitted through the request body. |
| PUT | /rules/containers/{name} | Updates the container rule with name: {name} with values emitted through the request body. |
| DELETE | /rules/containers/{name} | Erases the container rule with name: {name}. |

| | | |
|---|---|---|
| GET | /rules/containers/{name}/conditions | Return a list of the container rule conditions with name: {name}. |
| POST | /rules/containers/{name}/conditions | Associates the conditions emitted through the request body to the container rule with name: {name}. |
| DELETE | /rules/containers/{name}/conditions | Dissociates the conditions emitted through the request body from the container rule with name: {name}. |
| GET | /rules/containers/{name}/containers | Return a list of containers associated to the container rule with name: {name}. |
| POST | /rules/containers/{name}/containers | Associates the containers emitted through the request body to the container rule with name: {name}. |
| DELETE | /rules/containers/{name}/containers | Dissociates the containers emitted through the request body from the container rule with name: {name}. |
| GET | /decisions | Return the possible decisions. |
| GET | /decisions/services | Return the services possible decisions. |
| GET | /decisions/hosts | Return the hosts possible decisions. |
| GET | /operators | Return the operators that can be used to formulate conditions. |
| GET | /operators/{name} | Return the operator with name: {name}. |
| GET | /operators/{name}/conditions | Return the list of conditions that use the operator with name: {name}. |
| GET | /simulated-metrics/hosts | Return the defined hosts simulated metrics list. |
| GET | /simulated-metrics/hosts/{name} | Return the hosts simulated metric with name: {name}. |
| POST | /simulated-metrics/hosts | Add a hosts simulated metric emitted through the request body. |
| PUT | /simulated-metrics/hosts/{name} | Updates the hosts simulated metric with name: {name} with values emitted through the request body. |
| DELETE | /simulated-metrics/hosts/{name} | Erase the hosts simulated metric with name: {name}. |
| GET | /simulated-metrics/hosts/{name}/cloud-hosts | Return the list of cloud hosts that use the simulated metric with name: {name}. |
| POST | /simulated-metrics/hosts/{name}/cloud-hosts | Associates the simulated metric with name: {name} to the cloud hosts emitted through the request body. |

| | | |
|---|---|---|
| DELETE | /simulated-metrics/hosts/{name}/cloud-hosts | Dissociates the simulated metric with name: {name} from the cloud hosts emitted through the request body. |
| GET | /simulated-metrics/hosts/{name}/edge-hosts | Return the edge hosts that use the simulated metric with name: {name}. |
| POST | /simulated-metrics/hosts/{name}/edge-hosts | Associates the simulated metric with name: {name} to the edge hosts emitted through the request body. |
| DELETE | /simulated-metrics/hosts/{name}/edge-hosts | Dissociates the simulated metric with name: {name} from the edge hosts emitted through the request body. |
| GET | /simulated-metrics/apps | Return the list of the defined applications simulated metrics. |
| GET | /simulated-metrics/apps/{name} | Return the applications simulated metrics with the name: {name}. |
| POST | /simulated-metrics/apps | Add the applications simulated metric emitted through the request body. |
| PUT | /simulated-metrics/apps/{name} | Updates the application simulated metric with name: {name} with the values emitted through the request body. |
| DELETE | /simulated-metrics/apps/{name} | Erases the applications simulated metric with the name: {name}. |
| GET | /simulated-metrics/apps/{name}/apps | Return the applications that use the simulated metric with name: {name}. |
| POST | /simulated-metrics/apps/{name}/apps | Associates the simulated metric with name: to the applications emitted through the request body. |
| DELETE | /simulated-metrics/apps/{name}/apps | Dissociates the simulated metric with name: from the applications emitted through the request body. |
| GET | /simulated-metrics/services | Return the list of defined services simulated metrics. |
| GET | /simulated-metrics/services/{name} | Return the simulated metrics for service with name: {name}. |
| POST | /simulated-metrics/services | Add a service simulated metric emitted through the request body. |
| PUT | /simulated-metrics/services/{name} | Updates the service simulated metric with name: {name} with the values emitted through the request body. |

| | | |
|---|---|---|
| DELETE | /simulated-metrics/services/{name} | Erases the services simulated metric with name: {name}. |
| GET | /simulated-metrics/services/{name}/services | Return the services list that use the simulated metric with name: {name}. |
| POST | /simulated-metrics/services/{name}/services | Associates the simulated metric with name: {name} to the services emitted through the request body. |
| DELETE | /simulated-metrics/services/{name}/services | Dissociates the simulated metric with name: {name} to the services emitted through the request body. |
| GET | /simulated-metrics/containers | Return a list with the containers defined simulated metrics. |
| GET | /simulated-metrics/containers/{name} | Return the containers simulated metrics with name: {name}. |
| POST | /simulated-metrics/containers | Add a container simulated metric emitted through the request body. |
| PUT | /simulated-metrics/containers/{name} | Updates the container simulated metric with name: {name} with values emitted through the request body. |
| DELETE | /simulated-metrics/containers/{name} | Erase the container simulated metric with the name: {name}. |
| GET | /simulated-metrics/containers/{name}/containers | Return a list of containers that use the simulated metric with name: {name}. |
| POST | /simulated-metrics/containers/{name}/containers | Associates the simulated metric with name: {name} to the containers emitted through the request body. |
| DELETE | /simulated-metrics/containers/{name}/containers | Dissociates the simulated metric with name: {name} from the containers emitted through the request body. |
| GET | /regions | Return a list of the regions. |
| GET | /region/{name} | Return the region with name: {name}. |
| POST | /ssh/execute | Execute the SSH command emitted through the request body. |
| GET | /ssh/scripts | Return a list of supported scripts that may be used. |
| POST | /ssh/upload | Uploads the file emitted through the request body. |
| GET | /logs | Return a list with the logs. |

ANNEX II

# LOCAL MANAGER'S API

Annex 2 represents the API of a local manager. All the endpoints succeed /api.

| Request | Endpoint | Information |
|---|---|---|
| POST | /apps/app/launch | Initiates all the service of the application {app} using containers. |
| GET | /containers | Return a list of the containers. |
| GET | /containers/{id} | Return the container with id: {id}. |
| POST | /containers | Launch a container in an address, or coordinate emitted through the requests body. |
| DELETE | /containers/{id} | Stops the container with id: {id}. |
| POST | /containers/{id}/replicate | Replicate the container with id: {id} to the address emitted through the requests body. |
| POST | /containers/{id}/migrate | Migrate the container with id: {id} to the address emitted through the requests body. |
| GET | /containers/{id}/logs | Return the container: {id} logs. |
| POST | /containers/sync | Synchronize the database containers with the docker swarm. |
| POST | /hosts/cloud | Initiates and configures an instance in the cloud. |
| GET | /nodes | Return a list with the nodes. |
| GET | /nodes/{id} | Return the node with id: {id}. |
| POST | /nodes | Add a node at the address or coordinate emitted through the requests body. |

| | | |
|---|---|---|
| PUT | /nodes/{id} | Updates the node {id} properties with the values emitted through the requests body. |
| DELETE | /nodes/{id} | Removes the node with id: {id} from the docker swarm. |
| PUT | /nodes/{id}/join | The node with id: {id} rejoins the docker swarm. |
| DELETE | /nodes/publicIp/privateIp | Removes the node form the host with public IP: {publicIp} and private IP {privateIp} from the docker swarm. |
| POST | /nodes/sync | Synchronize the database nodes with the docker swarm. |

ANNEX III

# MANAGER'S API

Annex 3 represents the complement API of a manager referent to the monitorization management. All the endpoints succeed /api/edgemons.

| Request | Endpoint | Information |
|---|---|---|
| GET | /enum | Return all the Edgemons with detailed information |
| POST | /app/{microServName} | |
| POST | /alert/migrate/{orgContainerId}/{destContainerId} | Migrate the services monitorized by the Edgemon with container: {orgContainerId} to the Edgemon with container: {destContainerId}. |
| POST | /alert/migrate/{orgContainerId} | Migrate the services monitorized by the Edgemon with container: {orgContainerId}. |
| POST | /alert/replicate/{containerId} | Replicate the Edgemon with container: {containerId} and gives in half of the monitorization targets. |
| POST | /alert/{alertType} | Depending of the alertype: {alertType}, replicates or migrates. |
| POST | /alert/overload/{containerId} | Change the value of the Edgemon's overload. |
| POST | /create/{region} | Creates a new region to be used in the program. |

| | | |
|---|---|---|
| POST | /delete/{containerId} | Delete the Edgemon with container: {containerId} |
| POST | /stopMonitoring/{containerId} | Stops the monitorization activity from Edgemon: {containerId}. |
| POST | /startMonitoring/{targetContainerId} /{edgeContainerId} | Starts the monitorization of target microservice: {targetContainerId} by the Edgemon with container: {edgeContainerId}. |
| POST | /{containerId}/rulesp/{fname}/{origin} /{ltreshold}/{htreshold}/{metricName} /{metricHelp}/{ruleType} | Add rule to Edgemon with container: {containerId}, values are emitted through path variables. |
| GET | /{containerId}/rulesp | Return a list with the rules of Edgemon with container: {containerId}. |
| POST | /{edgemonName}/launch/ {workerManagerId} | Creates an Edgemon with name: {edgemonName} at the worker manager: {workerManagerId} |

ANNEX IV

# EDGEMON'S API

Annex 4 represents the API of an Edgemon module. All the endpoints succeed the Edgemon IP address.

| Request | Endpoint | Information |
|---|---|---|
| PUT | /scrape/{mode} | Change the scrape mode. Receives as path variable the new mode (node, store or all). |
| POST | /register | Register external microservices in the swarm. Localization (URL) emitted through the requests body. |
| POST | /scrapetarget | Register a swarm microservice. |
| POST | /scrapedtargets | Register a list of swarm microservices. Receives in the request body (JSON format) the microservices. |
| GET | /scrapedtargets | Return every microservices monitorized by this EdgeMon. |
| DELETE | /delete/scrapedtargets | Remove a list of microservices. |
| POST | /rulesp/{name}/{min}/{max}/{type}/{alert} | Create a new rule. Receive as path variables the name: {name}, max and min value: {min}, {max}, type of metric: {type} and the type of alert that must be triggered: {alert}. |
| POST | /rules | Register a list of rules. Rules are receive in JSON format emitted through the requests body. |

| | | |
|---|---|---|
| GET | /rules | Return the rules registered in the EdgeMon. |
| GET | /metrics | Return cache stored metrics. |
| GET | /containermetrics/{containerID} | Return container type metrics from container: {IP}. |
| GET | /nodemetrics | Return the host type metrics of the local node. |
| DELETE | /activetargets/{containerIP} | Pause the monitorization activity of the container: {IP}. |
| POST | /activetargets/{containerIP} | Restart the monitorization activity of the container: {IP}. |
| PUT | /activetargets/{containerIP}/ {metricType}/{scrapeInterval} | Updates the parameters of the monitorization. The values are emitted through path variables. |

ANNEX V

# REGISTRY CLIENT'S AND SERVICE DISCOVERY'S API

Annex 5 represents the API of a registry client and service discover. All the endpoints succeed /api.

| Request | Endpoint | Information |
|---|---|---|
| POST | /register | Registry the service in the registry server. |
| GET | /services/{service}/endpoints | Return the closest instance from the service {service}. |
| GET | /services/{service}/endpoints?among=x | Return a random instance to the service {service} chosen from the closest x instances. |
| GET | /services/{service}/endpoints?range=d | Return a random instance to the service {service} starting the search at a distance of d kilometres doubling in each iteration. |
| GET | /services/{service}/endpoints | Return all the instances registered in as {service}. |
| POST | /metrics | Add a new monitorization of this instance. Request body: {service, latitude, longitude, count}. |

ANNEX VI

# REQUESTS MONITOR'S API

Annex 6 represents the API of a requests monitor. All the endpoints succeed /api.

| Request | Endpoint | Information |
|---|---|---|
| GET | /location/requests | Return a list of all the monitorization registered. |
| GET | /location/requests?aggregation | Return a list of all the monitorization registered aggregated by service in the last 60 seconds. |
| GET | /location/requests?aggregation&interval={ms} | Return a list of the monitorization aggregated by service in the last {ms} milliseconds. |
| POST | /location/requests | Add a new monitorization task. Request body: {service, latitude, longitude, count}. |

ANNEX VII

# LOAD BALANCER'S API

Annex 7 represents the API to configure registered servers in the load balancer. All the endpoints succeed /api.

| Request | Endpoint | Information |
|---|---|---|
| GET | /servers | Return the servers of all the services registered on this load balancer. |
| GET | /{service}/servers | Return a list of all the servers of service: {service} registered on this load balancer. |
| POST | /{service}/servers | Add new servers to the service: {service}. Request body: {server, latitude, longitude, region}. |
| DELETE | /{service}/servers/{server} | Remove the server: {server} from service {service}. |