André Lameirinhas

Degree in Computer Science and Engineering

Monitoring in Hybrid Cloud-Edge Environments

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser:	Maria Cecília Gomes, Assistant Professor, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa
Co-advisers:	Vítor Duarte, Assistant Professor, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa
	João Leitão, Assistant Professor, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

March, 2019

Monitoring in Hybrid Cloud-Edge Environments

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

To humanity.

Acknowledgements

First of all I would like to thank this institution for providing me the means to do this dissertation. Second, and most importantly, I would like to thank my advisors for their tireless help and availability.

Abstract

The increasing number of mobile and IoT(Internet of Things) devices accessing cloud services contributes to a surge of requests towards the Cloud and consequently, higher latencies. This is aggravated by the possible congestion of the communication networks connecting the end devices and remote cloud datacenters, due to the large data volume generated at the Edge (e.g. in the domains of smart cities, smart cars, etc.). One solution for this problem is the creation of hybrid Cloud/Edge execution platforms composed of computational nodes located in the periphery of the system, near data producers and consumers, as a way to complement the cloud resources. These edge nodes offer computation and data storage resources to accommodate local services in order to ensure rapid responses to clients (enhancing the perceived quality of service) and to filter data, reducing the traffic volume towards the Cloud. Usually these nodes (e.g. ISP access points and on-premises servers) are heterogeneous, geographically distributed, and resource-restricted (including in communication networks), which increase their management's complexity. At the application level, the microservices paradigm, represented by applications composed of small, loosely coupled services, offers an adequate and flexible solution to design applications that may explore the limited computational resources in the Edge.

Nevertheless, the inherent difficult management of microservices within such complex infrastructure demands an agile and lightweight monitoring system that takes into account the Edge's limitations, which goes behind traditional monitoring solutions at the Cloud. Monitoring in these new domains is not a simple process since it requires supporting the elasticity of the monitored system, the dynamic deployment of services and, moreover, doing so without overloading the infrastructure's resources with its own computational requirements and generated data. Towards this goal, this dissertation presents an hybrid monitoring architecture where the heavier (resource-wise) components reside in the Cloud while the lighter (computationally less demanding) components reside in the Edge. The architecture provides relevant monitoring functionalities such as metrics' acquisition, their analysis and mechanisms for real-time alerting. The objective is the

efficient use of computational resources in the infrastructure while guaranteeing an agile delivery of monitoring data where and when it is needed.

Resumo

Tem-se vindo a verificar um aumento significativo de dispositivos móveis e do domínio IoT(Internet of Things) em áreas emergentes como Smart Cities, Smart Cars, etc., que fazem pedidos a serviços localizados normalmente na Cloud, muitas vezes a partir de locais remotos. Como consequência, prevê-se um aumento da latência no processamento destes pedidos, que poderá ser agravado pelo congestionamento dos canais de comunicação, da periferia até aos centros de dados. Uma forma de solucionar este problema passa pela criação de sistemas híbridos Cloud/Edge, compostos por nós computacionais que estão localizados na periferia do sistema, perto dos produtores e consumidores de dados, complementando assim os recursos computacionais da Cloud. Os nós da Edge permitem não só alojar dados e computações, garantindo uma resposta mais rápida aos clientes e uma melhor qualidade do serviço, como também permitem filtrar alguns dos dados, evitando deste modo transferências de dados desnecessárias para o núcleo do sistema. Contudo, muitos destes nós (e.g. pontos de acesso, servidores proprietários) têm uma capacidade limitada, são bastante heterogéneos e/ou encontram-se espalhados geograficamente, o que dificulta a gestão dos recursos. O paradigma de micro-serviços, representado por aplicações compostas por serviços de reduzida dimensão, desacoplados na sua funcionalidade e que comunicam por mensagens, fornece uma solução adequada para explorar os recursos computacionais na periferia.

No entanto, o mapeamento adequado dos micro-serviços na infra-estrutura, além de ser complexo, é difícil de gerir e requer um sistema de monitorização ligeiro e ágil, que considere as capacidades limitadas da infra-estrutura de suporte na periferia. A monitorização não é um processo simples pois deve possibilitar a elasticidade do sistema, tendo em conta as adaptações de "deployment", e sem sobrecarregar os recursos computacionais ou de rede. Este trabalho apresenta uma arquitectura de monitorização híbrida, com componentes de maior complexidade na Cloud e componentes mais simples na Edge. A arquitectura fornece funcionalidades importantes de monitorização, como a recolha de métricas variadas, a sua análise e alertas em tempo real. O objetivo é rentabilizar os recursos computacionais garantindo a entrega dos dados mais relevantes quando necessário.

Contents

1 Introduction **1**

1.1 Problem 2

1.2 Objectives 3

1.3 Proposed Approach 3

1.4 Contributions 3

1.5 Document Outline 4

2 State of the Art **5**

2.1 Cloud Computing 5

2.1.1 Cloud Service Providers 8

2.1.2 Virtualization 9

2.1.3 Containerization 11

2.2 Edge Computing 12

2.3 Microservices 15

2.3.1 Service-oriented Architecture 15

2.3.2 RESTful Service Composition 16

2.3.3 Microservices Approach 16

2.4 Monitoring 20

2.4.1 Cloud Monitoring 20

2.4.2 Edge/Microservices Monitoring 23

2.5 Monitoring Systems 24

2.5.1 Prometheus 25

2.5.2 Elastic 26

2.5.3 Osmotic Monitoring 28

2.5.4 Nonintrusive Monitoring 29

2.5.5 Conclusion 29

3 Proposed Solution **31**

3.1 Architecture Overview 31

3.2 Description of Cloud and Edge Components 35

3.3 Components' Responsability in System Capabilities 37

3.3.1 Metrics Acquisition and Storage 37

3.3.2 Metrics Processing . . . 38
3.3.3 Metrics Filtering . . . 38
3.3.4 Rule Checking and Alerting . . . 38
3.3.5 Visualization . . . 38

4 Implementation **39**
4.1 Architecture Implementation . . . 39
4.1.1 Components . . . 40
4.1.2 Correlation of Components . . . 40
4.1.3 Docker . . . 41
4.2 New Component: EdgeMon . . . 43
4.2.1 Metrics Scraping . . . 43
4.2.2 Metrics Filtering . . . 44
4.2.3 Metrics Storing . . . 45
4.2.4 Rule Checking and Alerting . . . 46

5 Validation **49**
5.1 Technologies Used . . . 49
5.1.1 Sockshop . . . 49
5.1.2 Load-test . . . 50
5.2 Scenarios' Description . . . 51
5.2.1 Stand-alone Sockshop . . . 51
5.2.2 Prometheus scraping Sockshop . . . 52
5.2.3 Prometheus scraping NodeExporter scraping Sockshop . . . 52
5.2.4 Prometheus scraping EdgeMon scraping Sockshop . . . 52
5.2.5 Infrastructure . . . 53
5.3 Results Analysis . . . 53
5.3.1 Stand-alone Sockshop . . . 55
5.3.2 Prometheus scraping Sockshop . . . 56
5.3.3 Prometheus scraping a Metric Scraper scraping Sockshop . . . 58

6 Conclusion **63**
6.1 Limitations and Future Work . . . 64

Bibliography **65**

Chapter 1

Introduction

Cloud systems [11, 12] are continuing to grow in importance as they provide everyday digital services like storage and computation in a convenient way and with a pay-as-you-go business model. As the worldwide demand and the number of mobile and IoT devices increase, so does the strain on the connection links which, consequently, increases latency and therefore degrades QoS. The data volumes produced by those devices is also expected to be so large that transferring and processing such volumes will be difficult if not impossible with current standard cloud solutions.

To overcome these limitations, edge computing [9, 19, 20] has become increasingly popular for the fact that it solves, in part, some of these issues. Edge computing refers to the existence of computational nodes geographically closer to the users, in contrast to remote datacenters available from the cloud platforms, that are used to offload cloud services, preserving their resources. It provides services near the costumer, which reduces latency and enables the ability of storing and processing data at those edge nodes. This reduces the volume of the transferred data and the amount of cloud processing, leading to less energy consumption and costs.

The microservices architecture [1] [13, 23] is very useful in this heterogeneous computational context, since it inherits the characteristics of Service Oriented Architecture (SOA) and Computing (SOC) [14], defining applications as sets of independent light/small services. Their communication is done via message exchange, usually using a communication medium, which allows building more complex services and in an incremental way. Each microservice represents a single functionality with its own (non-shared) database,

[1]https://martinfowler.com/articles/microservices.html

and usually it is developed and deployed independently (e.g. in containers). This separation and their small size, make microservices ideal to be executed on edge nodes that usually have limited storage and computational capabilities. For the same reasons, microservices also allow a faster and cheaper up-scaling of applications in the Cloud, due to smaller resource occupancy, reducing the cost of creating small-scale services' instances. However, a fine-grained modularity leads to increased service management and coordination, whose complexity requires automated solutions for their deployment, life-cycle control, communication and subsequent data dependencies, and overall application adaptation in time.

Monitoring [7] is therefore paramount in the context of microservices architectures to keep track of microservices coordination, resource usage and elasticity, liveness, or contracted Quality of Service (QoS), e.g. in terms of latency/response time and cost. Cloud monitoring [1] is an extensively studied subject, existing diverse tools allowing to determine with accuracy what each costumer is using and billing them accordingly. It also provides system metrics to improve the management and efficient optimization of the services, which is crucial for edge computing and microservices alike. The monitoring of microservices on the edge will therefore need to capitalize on metrics and methodologies of monitoring tools in the Cloud, in order to adapt to the smaller computational and storage capabilities of edge nodes, and to the possibility that microservices may either be running on cloud datacenters and/or on edge nodes.

1.1 Problem

The monitoring services currently available in cloud platforms support a comprehensive observation and control of running applications [18]. The cloud applications based on the microservices architecture, as well as their orchestration and control systems, already exploit those monitoring services to inspect microservices' execution and to support their response to events. This allows granting those applications and systems with the necessary elasticity properties, for instance in the face of failure events or significant variations of workloads.

The availability of very heterogeneous computational nodes at the Edge, to host migrating microservices, introduces the additional burden of how to continue monitoring those services' metrics and how to react to changes (e.g. in terms of the number of computational nodes and the volume and location of users' accesses). Orchestrating and monitoring an application composed of several microservices running with diverse resources and at different locations is therefore a very complex task. This suggests that current solutions based on a centralized monitor and orchestrator running on the Cloud pose scaling problems and are not adequate nor efficient.

1.2 Objectives

The objective of this dissertation is the development of a monitoring architecture and system in the context of microservices' architectures, with instances running on Edge and Cloud layers' nodes. It should allow the efficient collection of data coming from different sources and the dispatch of alerts in real-time from the Edge layer to the Cloud layer without the need of constantly congesting the network. Therefore, we intend to create a system that can be effective while sending considerably less data through the network when compared to more traditional monitoring systems' approaches employed in this environment.

1.3 Proposed Approach

A hierarchical monitoring architecture was envisioned to use the Edge layer as an integral part of the system. Collecting metrics, storing them temporarily, providing evaluation methods and alerting the Cloud layer are its main functionalities while the Cloud layer is used to permanently store potentially all metrics gathered and provide an API for exterior elements to visualize and process the existing data.

Overall, the monitoring system should be able to provide useful real-time information, be distributed over the cloud and edge environments, and be able to manage the large amount of monitoring data it will generate in a timely manner. An implementation was developed based on the aforementioned architecture, allowing the evaluation and validation of its objectives on an experimental environment.

1.4 Contributions

To fulfill these objectives, the contributions presented in this work are:

1. A monitoring architecture with focus on reducing the computational load of the Cloud layer and decreasing network congestion when monitoring distributed applications hosted in hybrid environments;

2. A new monitoring component, created to help fulfill the goals of the previous point;

3. An implementation of a system following the developed architecture and consequent deployment in a cluster environment;

4. The testing of some relevant scenarios to provide validation for the work done;

1.5 Document Outline

The remainder of this document includes: chapter 2 presents an overview on the state-of-the-art of the technologies important for the subject; chapter 3 presents the envisioned solution in a conceptual manner; chapter 4 shows how it was implemented and deployed; chapter 5 explains the performed testing scenarios along with an overview of the results collected; and chapter 6 derives conclusions based on the tests' results and overall objectives along with possible future work.

Chapter 2

State of the Art

This chapter addresses the state of the art related with the technologies that are relevant for this dissertation. We start with an overview of cloud computing, which supports the backbone of the system and is one of the environments where the applications being monitored reside. The second section discusses edge computing, a new paradigm that takes advantage of the ever-increasing number of devices in the periphery of systems to offload cloud nodes and which is also the new environment where components of both monitored and monitoring systems are deployed in our solution. Next, we delve into the microservices architecture, which we use as inspiration for constructing our monitoring system. Finally, the focal subject of monitoring itself is addressed, divided in its cloud and edge counterparts, followed by a section that describes some monitoring systems (conceptual and products) and why they are not sufficient when addressing the problem of network congestion and real-time alerting in hybrid environments.

2.1 Cloud Computing

Cloud computing [11, 12] is a technology that aggregates large quantities of distributed general-purpose hardware to provide easy access to many resources for the masses. It was born out of the necessity of making data storage and computation less expensive and liberating users/companies from the concerns of investment and system maintenance. With time, the facilities that stored these systems (data centers) only tended to get bigger because of the "economy of scale", which led to reduced operating costs making this kind of service a public utility.

There are different types of clouds depending on the requirements. Private clouds are only used by a single organization whereas community clouds are used by multiple users with shared interests. Public clouds are accessible to the general public and usually owned by a company that sells its services. Finally, hybrid clouds are a combination of the ones mentioned above.

Today there are three delivery models for cloud computing (illustrated in figure 2.1). They are Software-as-a-Service (SaaS), which lets users access specific applications that reside in the cloud; Platform-as-a-Service (PaaS) that provides a platform for developing, running and managing user-owned applications without the infrastructural complexity; and Infrastructure-as-a-Service (IaaS) which lets the client have control over some infrastructural decisions such as location, data partitioning, security, scaling, backup, etc.

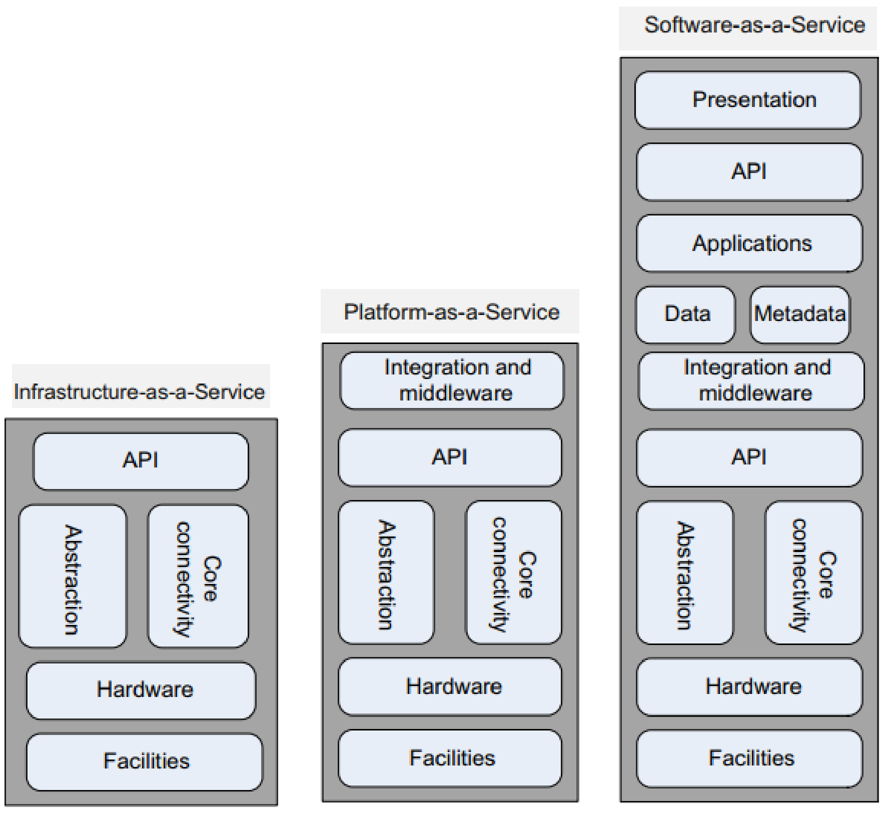

Figure 2.1: The structure of the three delivery models [11]

Some advantages of this technology are:

- Since most cloud systems are owned by companies, there is the will to keep improving this technology to reduce costs;

- The homogeneous nature of its hardware and software (similar machines with similar software installed and communicating with similar interconnects) makes the infrastructure easier to manage in terms of security, resources, fault tolerance and QoS;

- The cloud elasticity frees developers from the concern of working with reduced resources;
- Upcoming companies do not need upfront investment if they use a cloud infrastructure to host their system;
- Having data centers dispersed geographically can prevent catastrophic failures (if applications are deployed in multiple data centers).

There are also some disadvantages:

- The migration of big data from one cloud provider to another incurs in great costs and takes a considerable amount of time;
- Vendor lock-in: when a big cloud user is forced to keep relying on the same cloud provider because of the fact enunciated in the previous point;
- Security and confidentiality of the data are major challenges because there can be failures in data isolation (sometimes data from other users becomes erroneously accessible), and an attack on the cloud security could compromise the data from all users.
- When something goes wrong like unauthorized access, data corruption, infrastructure failure or unavailability, it is very hard to determine who is accountable because there are many independent actors in play like the provider, consumer, carrier, auditor, broker, etc (figure 2.2 illustrates some of these actors).

Regarding the possible applications that can be hosted in the Cloud, three major categories can be defined: processing pipelines, which are data-intensive and compute-intensive applications such as indexing (Googlebot[1], Google's web crawler is a good example), data mining and image processing; batch processing are data-intensive and have deadlines to fulfill their role like daily activity reports of organizations, processing transaction summaries, billing and inventory management; lastly there are web applications that are used to provide clients with web content in a fast way.

[1]https://varvy.com/googlebot.html

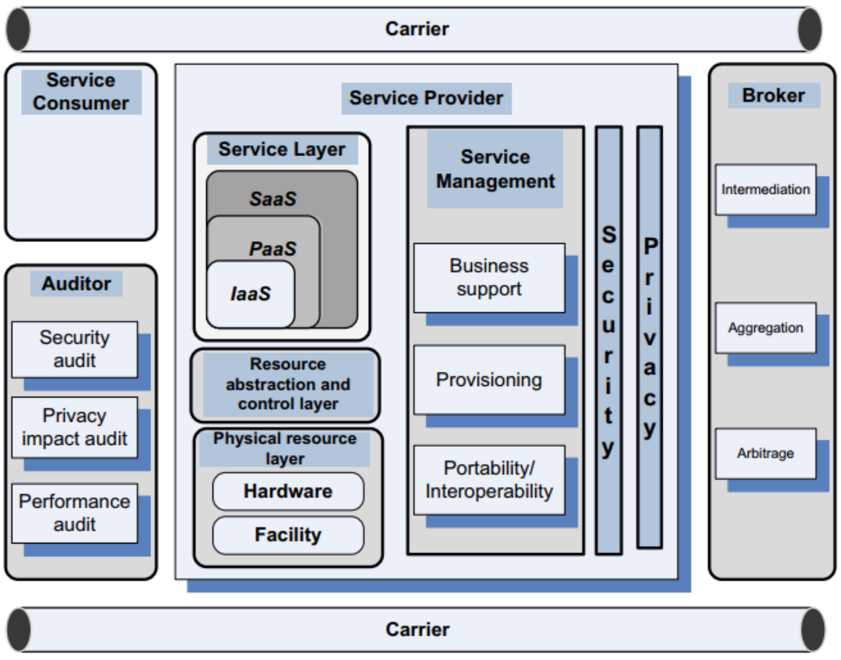

Figure 2.2: The entities involved in Cloud Computing [11]

2.1.1 Cloud Service Providers

Some of the main Cloud Service Providers (CSPs) are:

- Amazon, which provides a plethora of products with its AWS (Amazon Web Services) service. Users can also rely on cloud applications provided by Amazon to facilitate their systems' management such as: Elastic Compute Cloud (EC2) for virtual machines' management, Simple Storage System (S3) for storage and Cloud Watch for monitoring applications which enables alerting, automated responses and insights' discovery to optimize applications via analyzes of logs and collected metrics.

- Google is best known for its SaaS model. It has brought us multiple applications such as Gmail, Google Drive, Google Calendar, Picasa, Google Groups, etc. But it also has AppEngine, which is a PaaS service that supports Python and Java development and Compute Engine, their IaaS platform, which supports scaling and load balancing.

- Microsoft provides both PaaS and SaaS via Microsoft Azure, which includes services that provide a computational environment to deploy applications, a scalable storage and a monitoring service, Azure Monitor, that provides advanced analytics for performance monitoring and machine learning to proactively identify issues and automatically respond to alerts. There is also the Content Delivery Network (CDN), which is a distributed network of proxy servers that maintains cache copies of user data to speed up access and computations.

Besides the main companies, there are also some open-source cloud platforms such as Eucalyptus, OpenNebula, and Nimbus that are used for specific reasons such as scientific research [2].

To better establish the relationship of mutual understanding between cloud providers and clients, a service-level agreement (SLA) can be defined, a negotiated contract that settles what the client wants and what the provider can provide, being it resources, security or QoS. It is also used to specify costs, responsibilities of each party and to provide a framework for communication.

2.1.2 Virtualization

Cloud systems rely heavily on virtualization to provide services to individual users and companies, and as such it is an essential technique for cloud providers today.

Virtualization [10] is a method to simulate virtual objects from physical ones and it can take various forms, such as:

- multiplexing - where multiple virtual objects are created from a single physical one, a good example is a CPU that multiplexes itself into processes;
- aggregation, which is the opposite of multiplexing and a good example is a distributed file system;
- emulation, where an object is created using one or more different physical objects, such as when a hard disk emulates random access memory;
- a combination of the types above is also possible.

Virtualization is now a technique widely adopted because it provides relevant advantages such as system security (granted by service isolation), reliability and elasticity (achieved from the ability to easily migrate applications) and performance isolation. The penalty for these advantages are the added overhead that, in turn, mitigates performance and the increased hardware costs to support this technology. A virtual machine (VM) is an isolate environment that is used to emulate a whole computer but in fact it only has access to a portion of the physical computer resources. There are two types of VMs,

[2]www.eucalyptus.com, www.opennebula.org, www.nimbusproject.org

process VMs that only exist for the duration of the process they were created for, and system VMs that emulate an operating system and can run indefinitely.

To manage these VMs inside a system, the virtual machine monitor (VMM) also called hypervisor was invented. Its main responsibilities are to partition the available resources to the existing VMs while providing isolation and security properties. These VMs instantiate a guest OS that resides inside the VMs sandbox instead of being directly connected to the hardware, and they only have user permissions while the VMM have root permissions. This allows the existence of several guest OSs inside a single system, which in turn enables multiple services to share the same platform and supports the ability to change an application location at will.

There are three possibilities when deploying systems that use VMs (illustrated in figure 2.3):

- Traditional: the VMM runs directly on the hardware (called the bare metal approach), made for performance;
- Hosted: the VMM runs inside the native OS and therefore can be easily installed. It has the advantage of being able to use some of the native OS's components such as the scheduler, pager, I/O drives, etc.
- Hybrid: the VMM runs alongside the native OS.

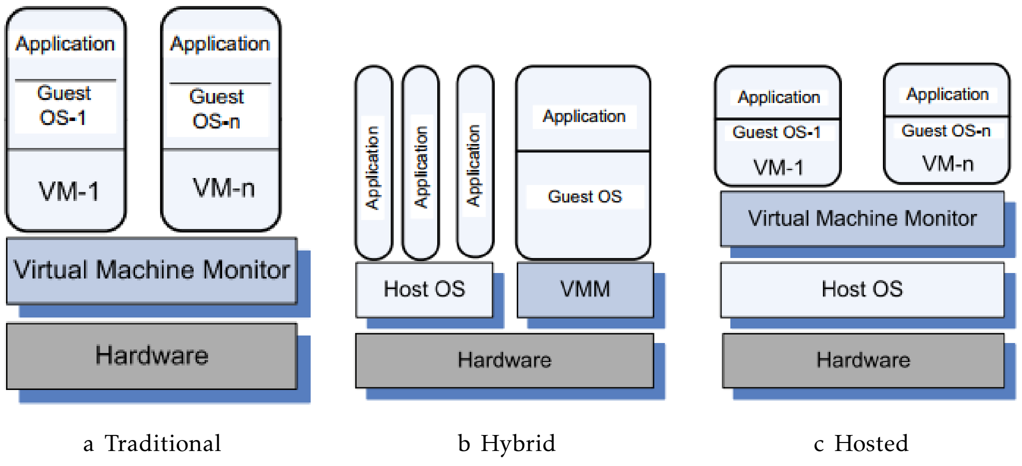

Figure 2.3: Classes of Virtual Machines [11]

For a computer architecture to support virtualization, while having an efficient VMM, some conditions are necessary: the VMM must have total control of the system resources, to be able to allocate them when needed; any program running on a VM must behave exactly the same as when executed on a native OS; and the majority of the machine instructions must be executed without the intervention of the VMM to preserve performance. If all the machine instructions can be virtualized, then we face full virtualization. When that is not possible, we are forced to use paravirtualization, which is the recompilation of parts of the guest OS to support non-virtualizable machine instructions. This is also done to improve performance and to simplify the architecture.

2.1.3 Containerization

In terms of virtualization, there's also the approach to use containers[6], popularized by Docker[3]. A Docker container is a virtual environment used to run an application that will always behave the same way, regardless of the infrastructure (minimum resource requirements must be met). It is lightweight, much smaller than a conventional virtual machine and utilizes key Linux Kernel features, such as Namespaces (for process isolation) and CGroups (for resource isolation), to provide an overall isolation and security to the application inside and its dependencies. An application running inside a container is also very portable as all of its dependencies are packaged together and separated from the machine's operating system. Figure 2.4 shows the architectural difference of hosting applications using containers and virtual machines.

In the context of microservices' architecture, containers are used to facilitate the deployment of applications composed of many components, using one container for each component, which subsequently can be deployed and managed independently. This dissertation uses Docker containers to deploy the implemented monitoring system and also the application being monitored.

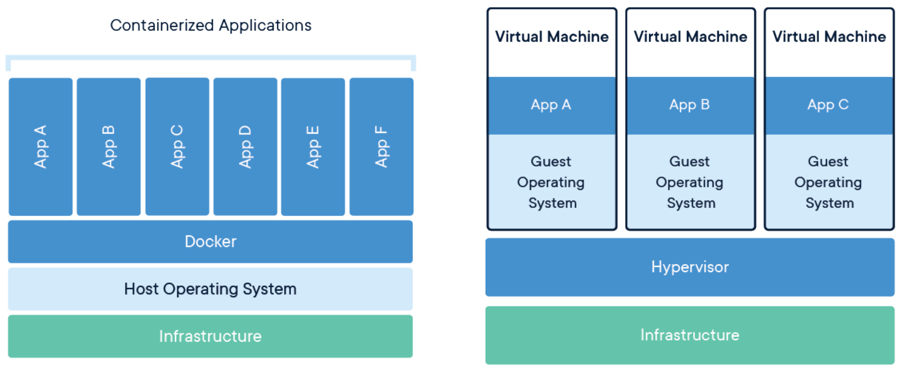

Figure 2.4: Comparison between the architectures of containers and virtual machines

[3]https://www.docker.com

2.2 Edge Computing

With the constant increase in end devices such as smartphones, tablets, and IoT devices, the data sent to the Cloud is also increasing, which makes the idea of making computations in the edge nodes more appealing. Edge computing [9, 19, 20] is an optimization to the cloud computing architecture so that, by moving computations, storage and other resources to the edge nodes of the system (on-premise data centers, ISP's points of presence(POP), Cloudlets, etc.) are alleviating the strain in the communication network caused by data transmission and the cloud servers' workload.

This architecture has many applications like aggregation of big data or sensor data, offloading the Cloud from frequently used transcoding and criptography protocols, caching, etc [3].

Generally, the architecture used for edge computing consists of three levels (depicted in figure 2.5):

- **device level**, where all the user devices that produce data or run applications that use edge/cloud resources reside;
- **edge level**, where the nodes dedicated to edge computing are;
- **cloud level**, the cloud infrastructure itself.

The heterogeneity of the hardware in both device and edge levels and the lack of standards are the principal difficulties in creating edge computing systems.

The implementation of edge computing in real life systems is motivated by several factors. Some of these factors are: the decentralization of the cloud system that, in turn, provides low latency for the users; to overcome the resource limitations of end devices by using edge nodes or other more resourceful end devices for computations; to create a more energy sustainable system (data centers need large quantities of energy to operate); dealing with the ever-increasing network traffic directed at the cloud; and offloading some of the computations done by the cloud closer to the source of the data.

The complex nature of resource management in this architecture leads to the creation of a taxonomy of concepts to help achieve this endeavor (shown in figure 2.6). It divides matters into resource type, objective and resource use (note that these matters are intrinsically related as the type of resources available and their objective shape how they are going to be used). As for resource types, they can be divided into physical resources such

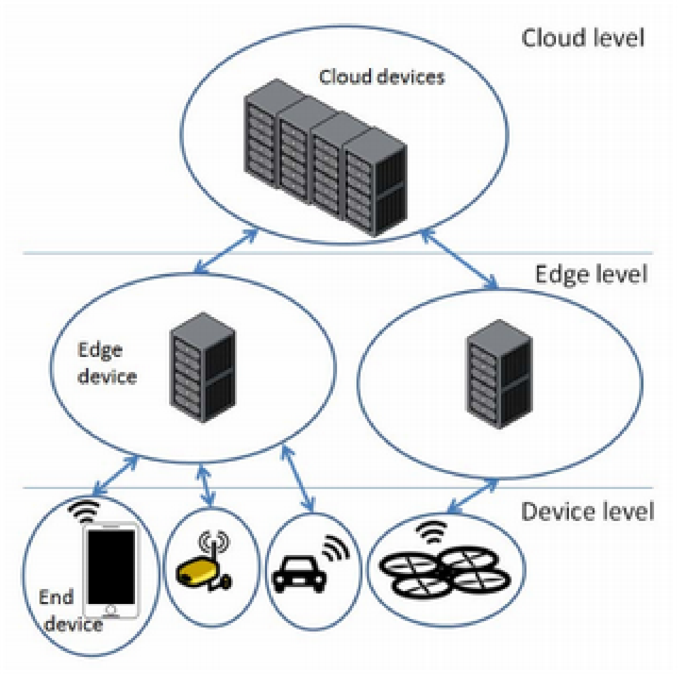

Figure 2.5: Three-level architecture used in edge computing [19]

as computation, communication, storage and energy (precious for end devices), and virtual resources that are virtual abstractions of physical resources, such as virtual machines or containers residing in the edge nodes. The main objectives are the identification of how many resources are available (resource estimation), where and to whom are those resources allocated (resource allocation), how they can be shared (resource sharing) and how to optimize all these objectives (resource optimization). Lastly, resource use defines the purpose of these resources. They can be of a functional nature, representing the resources that are effectively used to provide users with cloud/edge services, and non-functional, which are resources used to guarantee specific qualities a system needs to fully operate. These qualities can be divided into execution qualities like safety and security and evolution qualities like maintainability and scalability.

To be able to build such systems, some challenges must first be overcome, such as: transforming edge nodes such as routers in general purpose computing devices, which will incur in high costs; developing mechanisms for edge node discovery [16]; being able to partition tasks in an efficient and automatic way [24], while dealing with the heterogeneity of the hardware; maintaining the same quality of service (QoS) and user experience (QoE) that was provided by the cloud [2], and dealing with privacy and security of user data [4] when using other end devices and edge nodes for computations [20].

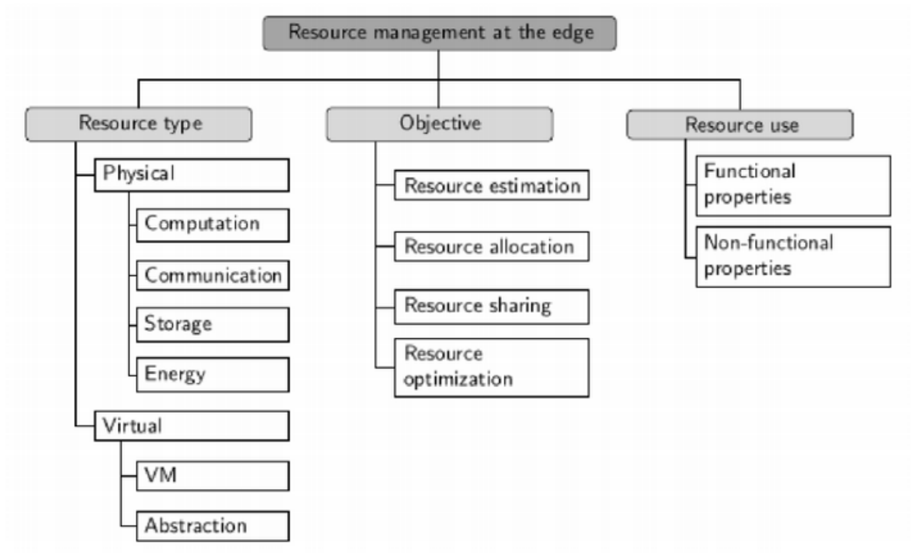

Figure 2.6: A taxonomy of resource management in the edge [19]

This new technology opens space for several opportunities of advancement in computer science. Some examples are: the creation of new standards for edge computing (communication, incentives, resource management, etc.) that facilitate the transition to this technology and help avoid major system problems created by the lack of knowledge in the matter; benchmarking techniques that represent more accurately the edge computing paradigm; an edge computing marketplace with a pay-as-you-go basis (that allows precise billing for the resources used); new frameworks and languages with built-in functionalities to handle the complexity of this technology (like Go [4] handles concurrency); new lightweight libraries [8] , algorithms and micro OS's (or kernels) to be used in edge devices [22]; and a partnership between industry and academic entities with the objective of alleviating the assumptions required when researching large-scale technologies, providing useful data on the matter and real infrastructures for testing, to further advance edge computing [20].

Figure 2.7 shows these motivations, challenges and opportunities in a more concise manner.

[4]https://golang.org/

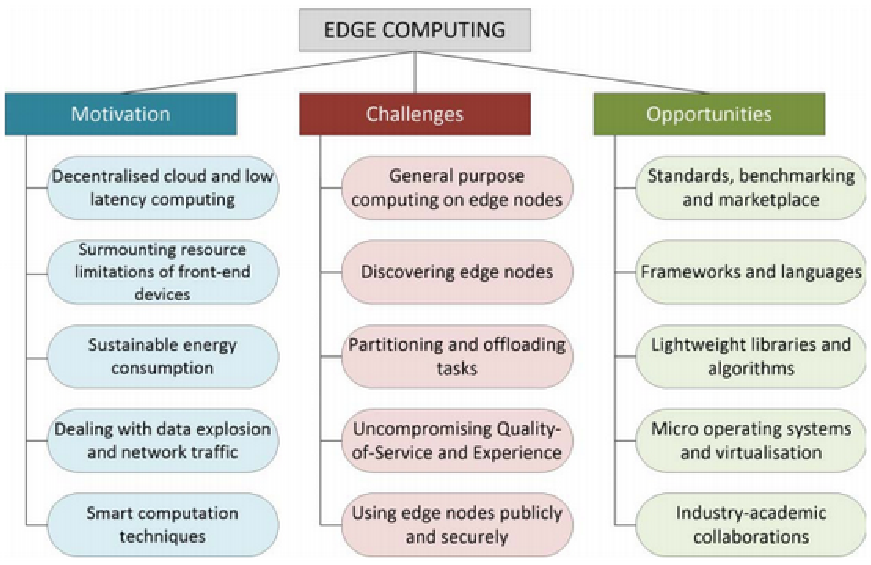
EDGE COMPUTING
Motivation
Decentralised cloud and low latency computing
Surmounting resource limitations of front-end devices
Sustainable energy consumption
Dealing with data explosion and network traffic
Smart computation techniques
Challenges
General purpose computing on edge nodes
Discovering edge nodes
Partitioning and offloading tasks
Uncompromising Quality-of-Service and Experience
Using edge nodes publicly and securely
Opportunities
Standards, benchmarking and marketplace
Frameworks and languages
Lightweight libraries and algorithms
Micro operating systems and virtualisation
Industry-academic collaborations

Figure 2.7: Motivation, challenges and opportunities of edge computing [20]

2.3 Microservices

This section describes the microservices architecture [5] [13, 14, 23] . Its roots in the Service-oriented Architecture (SOA) are explained in the first subsection, followed by the general transition to a RESTful composition in the second subsection. The last subsection approaches with more detail the characteristics and design patterns of microservices architecture.

2.3.1 Service-oriented Architecture

There has been a model shift in the way enterprises build their IT infrastructures. The traditional approach is a full integrated solution with all the software built and managed internally. In many cases, it has been replaced by business networks where companies provide each other with specialized web services to interoperate and achieve cooperative multi-functional systems. This service-oriented architecture model is only possible if each company uses standard-based definitions to specify the description, communication, composition, discovery and quality of service of their services. The first defined standards were SOAP for communication, WSDL for service description and UDDI for service discovery.

Service-oriented architecture can be viewed as a pyramid composed of three layers that represents the structure of services. The lowermost is the description layer where the basic services reside, provided by service providers, along with their descriptions. Then, a service aggregator can pick some of these services and, through specifications

[5]https://martinfowler.com/articles/microservices.html

like BPEL4WS [5], merge them into a composite service that resides in the middle layer, called the composition layer. A service client can use services from both of these layers. The uppermost layer is the management layer. Services in this layer are business-size services that need to be carefully managed to make sure the functionality and performance requisites are met.

Currently, most of these protocols have fallen into disuse. Their complexity and the new RESTful solution might have been the cause.

2.3.2 RESTful Service Composition

Initially service composition was mostly done using SOAP and with it came two distinct approaches, orchestration, which is a single-party perspective of a system built with web services, and choreography, which is the exchange of messages from multiple parties that have their own web services.

Then a restful approach was introduced because of the possibility of specifying a service based on its resources instead of its actions. Restful services use well-known web standards like HTTP, XML and URI, which make it lightweight and very simple to use because little to no extra tools are needed. It also takes advantage of the web infrastructure since there are HTTP clients and servers available in many languages, which makes the transition smoother. Because of these properties and its stateless nature, restful services can scale very well.

Since there is no service discovery method for REST (like UDDI for SOAP), the only way to find published services is by searching webpages that collect and categorize them. These pages are called mashups and require human interaction, which is a disadvantage compared with the automation of UDDI-like discovery. The lack of standards for this approach, which leads to ad-hoc solutions and subsequently reduced interoperability and the need to write extensive API documentation, is another reason why some companies are still fearful of making the shift into restful service composition.

2.3.3 Microservices Approach

The term microservices refers to an architectural style in which a system is composed of a variety of small autonomous services, each one with its own function and that communicate through a lightweight API to achieve the intended high-level functionality of the system. This style has its roots in the Service Oriented Architecture and, much like it, follows the philosophy that a system should be divided into functional services developed and maintained by small independent teams throughout its complete lifecycle.

In this context each service is a web service that is self-contained and platform-neutral, and capable of adapting and interacting uniformly, preventing tight coupling between services. These services can be specified with a class definition or a XML-based composition specification. The former is more widely used in the development stage because existing class definitions can be reused or extended to create new services simply by adding or overriding the definition (much like Java).

The microservices paradigm is used by an increasing number of companies, such as Amazon, Netflix, and Ebay, which can compensate the complexity of the resulting distributed system by achieving meaningful benefits when compared to the monolithic system approach. Some of these advantages are:

- It supports a very resilient system in the sense that each service is executed on a separate process so, when failures occur, they do not compromise the entire system and are easier to isolate and patch.

- Because of the fact that each service is autonomous and communicate through APIs (HTTP requests for example), it is possible to use different technologies and different programming languages to develop each one. This gives the companies the possibility of trying/using recent state of the art technologies with less risk.

- It is easier to allocate resources to specific sections of our system because microservices live in different processes. Also upon experiencing high traffic to a certain service, several instances of that service can be started, which improves scalability.

- Any service can be easily understood and rewritten in about two weeks given its small codebase, which gives flexibility when trying to debug, mantain, upgrade or refactor the overall system.

- Another great advantage is the fact that if a new feature or improvement is needed in our system, it is not required to “stop everything”. A simple replacement of the service in runtime would suffice, excluding the need to deploy the entire system again, which would take a lot of time and effort (this property in inherited directly from SOA).

The choice of using a microservices architecture causes communication between services to be the main issue. The ability of each service to have the same view of the whole system comes with a toll on efficiency. For this purpose or any other it is not recommended to allow tight coupling between services because that would break the modularity of the system and erase some of the advantages mentioned above.

Another aspect is that remote calls are slower than common function calls or method invocations, which delays the responsiveness of the system causing fewer operations to be made and consequently leads to less capability to handle high traffic.

A good example of the transformation from a monolithic system to microservices would be an e-commerce application [6] that deals with accounting, inventory management and shipping of its products. Each of these responsibilities can be attributed to a specific microservice that has its own full stack and communicates using a RESTful API. The resulting design pattern can be seen in figure 2.8. It is clear that an application using this design can scale very easily (by deploying more instances of the service in need) while the other services continue to run oblivious of the changes made. Flexibility is also a property of this design because the same service (e.g. shipping) can be used in different applications.

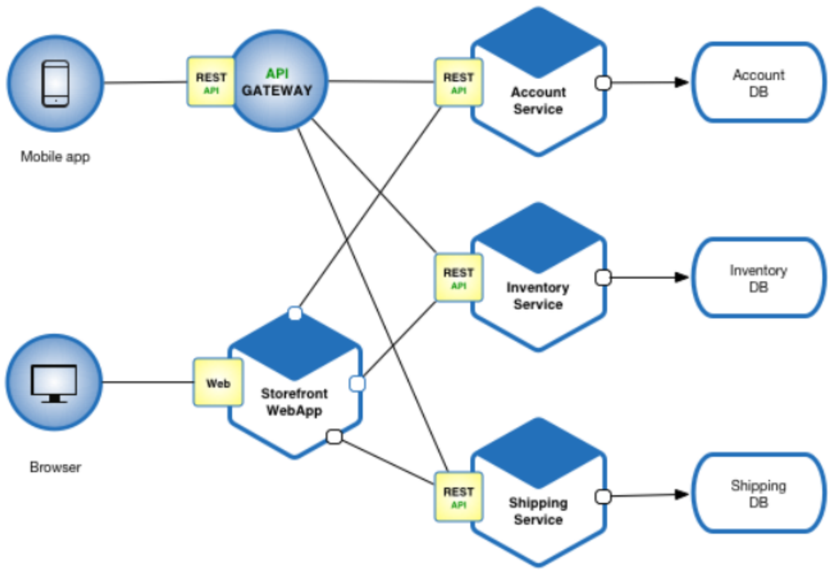

Figure 2.8: E-commerce design pattern

Other pattern options are possible to fulfill specific needs that applications might have [7]. Some of them are called Aggregator, Chained, Branch and are represented in figures 2.9, 2.10 and 2.11. The aggregator pattern is very common when services do not have a direct dependency with each other and the e-commerce application mentioned before is a good example of this design. The chained pattern is mostly used when the workflow of the application requires direct dependency from one service to another. Finally, the branch design is a combination of the aggregator and chained patterns and provides some leeway for application that have dependencies only in some services.

[6] http://microservices.io/patterns/microservices.html
[7] http://blog.arungupta.me/microservice-design-patterns/

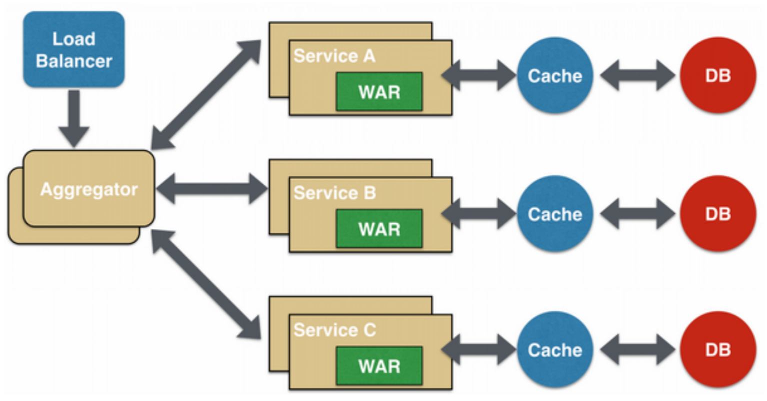

Figure 2.9: Aggregator design pattern

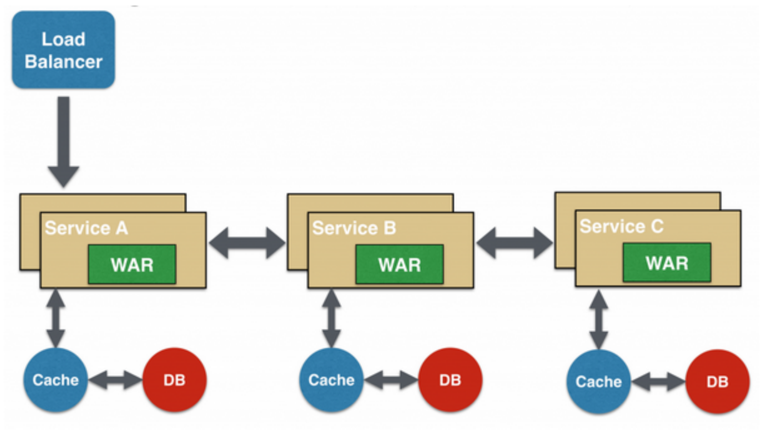

Figure 2.10: Chained design pattern

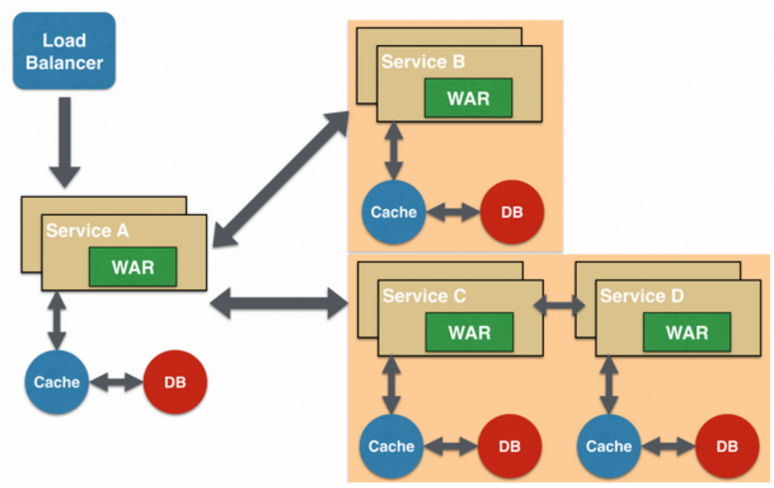

Figure 2.11: Branch design pattern

2.4 Monitoring

Cloud and Edge environments are the setting where many applications reside, and as their complexity increases (partly because of microservices architectures), the necessity for monitoring also increases . In this section different approaches to monitoring [1, 7, 25] are specified. The first subsection targets the cloud while the second targets the edge and the microservices paradigm.

2.4.1 Cloud Monitoring

Monitoring a cloud system is paramount for its efficient resource management and to verify the properties agreed upon in SLAs by providers and consumers. It is also very useful for developers and system administrators as it provides data about the performance and QoS of the application, which allows dynamic reactions to internal/external events. Some services that enable the monitoring of Cloud application are Application Insights from Microsoft and Cloud Watch from Amazon.

Different monitoring topologies can be used with respect to the resources of the cloud provider and its costumers' requirements. In these topologies three working agents can be specified: root agents "RA", which decide scheduling algorithms and management strategies to be applied in the system; federation agents "FA", which receive the collected monitoring data, process and store it persistently; and collection agents "CA"(also called sensors), which locally collect the data and forward it to the federation agents.

Figure 2.12 shows the four most prominent topologies, which are:

- centralized topology – all CA's communicate directly with a single RA (single point of failure), which makes for a very easily manageable system but it does not scale well;
- layered topology – hierarchical approach where the work is delegated along all agents in a three-like structure and it is highly scalable;
- P2P topology – decentralized approach where all agents can fulfill any of the three roles, which avoid a single point of failure but adds complexity to the system;
- hybrid topology – a combination of the layered and the P2P topologies, it inherits the benefits from both making a more easily manageable system but that can still scale well;

For monitoring purposes, a cloud can be divided into seven layers where different monitoring strategies can be applied. These layers are (from bottom to top):

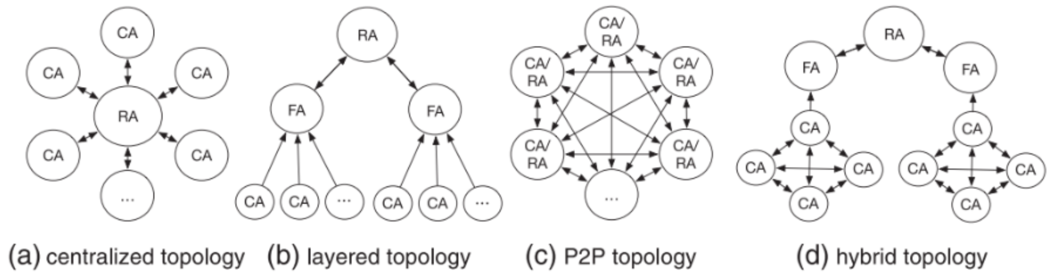

Figure 2.12: Cloud monitoring topologies [25]

1. facility – cloud infrastructure and organizational component;
2. network – the links inside the cloud and between cloud and user;
3. hardware – physical components and networking equipment;
4. OS – software operating in the machines of the cloud system;
5. middleware – software that allows communication between the OS and user applications;
6. application – user applications that run inside the cloud;
7. user – user applications running outside of the cloud (e.g. clients, browsers, etc.);

There are two levels of monitoring, high level and low level. High level monitoring focus on the overall status of the applications that are running in the cloud system, and the data is usually collected at the uppers layers (middleware, application and user). Properties like QoS, performance and SLA compliance are addressed at this level and are the main concern of the consumer. Low level monitoring is more concerned about the status of the infrastructure and collects information from the lower layers. Properties like CPU usage, network throughput and software vulnerabilities are prioritized at this level. This information is very useful to the provider and, in most cases, it is hidden from the consumers.

When monitoring a system, two types of metrics are possible, computation-based and network-based. Computation-based metrics gather data related to computation such as CPU usage, memory page exchange rate and disk throughput. Network-based metrics focus on the network of the system and collect data that is related to the transfer of data such as Round-Trip Time (RTT), jitter, packet-loss ratio, etc.

The monitoring process can be divided in 4 steps:

1. generation of monitoring data;
2. data management;
3. processing of monitoring data;
4. dissemination and presentation.

Multiple implementation choices must be made at each step when designing a monitoring system based on the requirements of both the system to be monitored and the system that will use the collected data. First, it is necessary to decide at which level the monitoring probes will operate (service level, platform level, or host level). Second, the generated data must be stored, which opens the possibility of using centralized or decentralized storage and using dedicated servers or a third-party service (SaaS). Third, it is also necessary to distribute data among nodes, which can be achieved using pull or push techniques. Within a pull strategy each interested entity decides when and where to collect data. Since each node can aggregate information locally, this method reduces network traffic but opens the possibility of losing data in the event of a failure in one of these nodes. When a pushing strategy is used, all the data collected by any node is immediately transferred to the central point of aggregation (which is a bottleneck of the system). This enables real-time monitoring but creates high network traffic. Fourth, the processing stage can focus on short-term and/or long-term analysis. If the interest is on producing long term reports, all the data must be aggregated to achieve meaningful results for the system. On the other hand, for root-cause analysis the data in its native form is more relevant but can only be stored for limited periods of time, otherwise the amount of stored data can become massive.

Some of the most important properties for a cloud monitoring system are: (i) scalability, being able to increase the number of probes used and still be efficient; (ii) elasticity, the ability to deal with the dynamic nature of the monitored system; (iii) adaptability, which is the ability to continue monitoring even with varying computation and network loads; (iv) timeliness, being able to deliver information while it is still relevant; (v) autonomicity, the ability to self-manage itself in the face of unpredicted changes; (vi) comprehensiveness, extensibility, and low intrusiveness are, respectively, the ability to support different type of resources, being able to add even more if necessary, and requiring low instrumentation costs; (vii) resilience, reliability, and availability are the ability to continue functioning properly in the event of failures, being able to fulfill its purpose timely, and providing its service uninterruptedly (respectively); lastly, (viii) accuracy, which is delivering information as close as possible to the real values.

The monitoring systems used today only tend to offer a subset of the properties mentioned above and/or are only applied on specific layers of the infrastructure.

Monitoring a cloud system adds significant overhead to the computation and the network, which affects providers and costumers. Probes need to be instantiated which consumes cloud resources or instrumented directly into the application code which is troublesome. Also, dedicated bandwidth can be allocated for monitoring purposes which, again, uses resources; otherwise, the costumer will suffer with reduced network capacity. Nevertheless, it is still useful. Some of the primary motivations are: be able to plan and manage the resources needed for specific applications; manage the data center infrastructure itself; verify the SLA compliance; supply accurate billing proportionate to the service usage; troubleshoot problems and discover its root cause; and guarantee a degree of performance and a degree of security.

The main focus of state-of-the-art cloud monitoring research is to develop ways of making monitoring more effective and efficient by developing new tools and techniques, creating cross-layer and cross-domain monitoring [21], creating new standards and more accurate-to-the-reality testbeds, and implementing monitoring in other types of systems like Edge Computing.

2.4.2 Edge/Microservices Monitoring

Monitoring the Edge is directly associated with monitoring microservices since systems supported by that paradigm are mainly built using a microservices' architecture. Services run in different processes over different hosts, which increase the complexity of collecting, processing, and distributing monitoring data.

The main concerns when developing a system for monitoring microservices should be: first, identify the stakeholders, or who is interested in this endeavor (project managers, developers, quality assurance professionals, etc.); then, reach an understanding about what is the objective of monitoring the specific system (problem detection, reporting, performance analysis, etc.); after knowing the objective, it is then possible to extrapolate what information is required to achieve the desired objective. A thorough depiction of these concerns is shown in figure 2.13.

In the context of microservices, more often than not, containers are used and so the monitoring system must be aware that the target system resides inside a container. In such cases monitoring can be instrumented by each application (which is troublesome for the developer) or by a kernel-level approach that is able to collect metrics of each application even if they are split in different containers. It is also important to provide a monitor API that is tailored to each stakeholder and that allows access to specific information such as end-point metrics (slowest, most used) or system call traces.

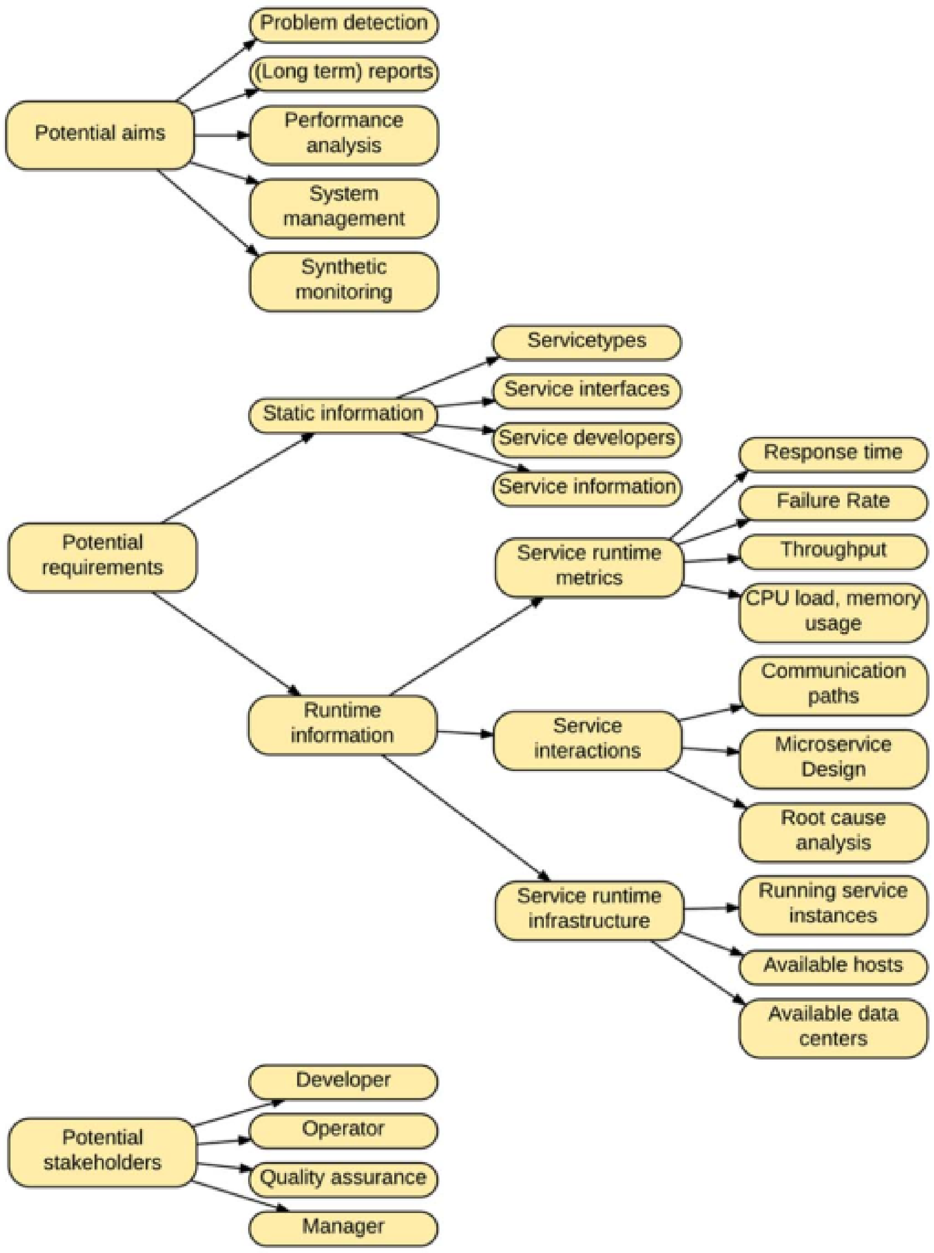

Figure 2.13: Aims, requirements and stakeholders in microservices monitoring [7]

2.5 Monitoring Systems

In this section we talk about the efforts that are being made towards building monitoring systems that can perform favorably in Edge environments, some technologies that exist today that are helping those efforts and why they are not enough.

2.5.1 Prometheus

Prometheus[8] is an open-source monitoring and alerting service. It can be used to monitor multiple applications located in multiple machines and collect a vast array of different metrics. It is a modular system, meaning it is divided into independent parts with specific functions that can communicate with each other via REST API's. It is frequently deployed using the docker technology in which each part (a microservice) is inserted in a docker container to ease deployment and enhance mobility.

In terms of architecture, the Prometheus model divides its services in two groups: Prometheus itself (accompanied by minor help services) and targets or scrapers which are the services that effectively collect metrics. This model is shown in figure 2.14.

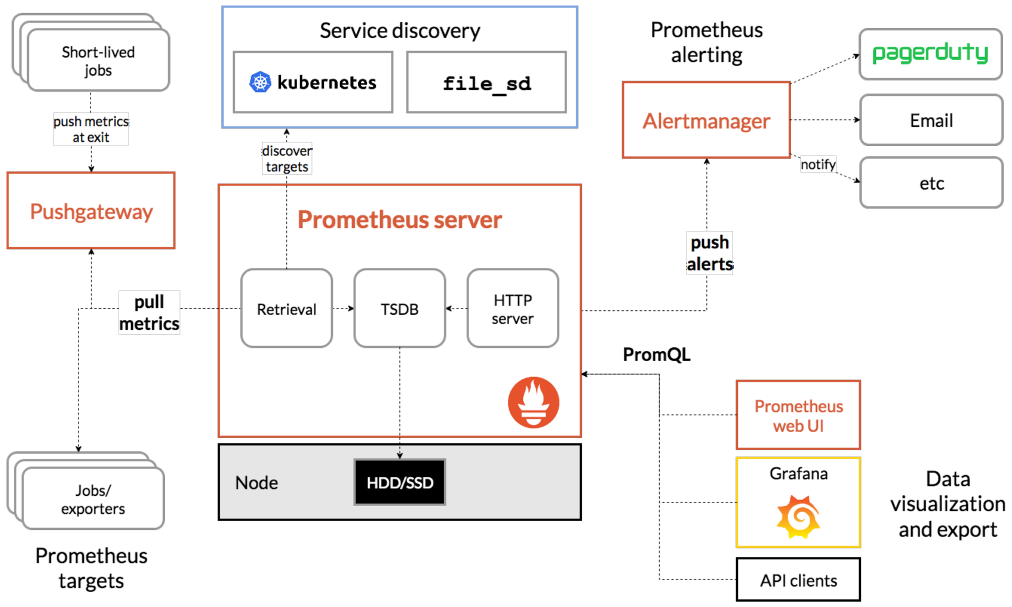

Figure 2.14: Prometheus architecture

Prometheus service works as a metrics database and rule engine capable of alerting via email, pager, etc. in the event of a metric firing some predefined rule. It is this service that "asks" for metrics from the available targets using a pull strategy meaning targets only deliver metrics when asked directly by Prometheus.

Configuration of the Prometheus service is done via a configuration file that defines all the targets in the system, their location and the scraping interval. This configuration can be changed in runtime by updating the file and sending a SIGHUP signal to Prometheus, which will reload the file and update the configuration. This reload functionality is very useful when we are dealing with mutable environments like the Edge, in which component can change location arbitrarily (mobile device) or disappear completely (IoT

[8]https://prometheus.io

device's power runs down).

For Prometheus to scrape these targets it must first know their location (IP:PORT). To obtain such information it relies on Service Discovery mechanisms. While Prometheus does not possess a Service Discovery functionality itself, it can seamlessly integrate those services from Kubernetes or Docker DNS allowing the system to adapt to architectural changes.

The majority of the targets provided by Prometheus is used to scrape metrics of a specific kind (node metrics, container metrics, database metrics, network metrics, etc.). These targets can be implemented in any programming language and are completely independent from the main Prometheus service.

Prometheus possesses a library[9] of different targets released by the Prometheus team and also by its user community. Some notable mentions are:

- Node-exporter – a low-level metrics target (CPU, RAM, etc.);
- Cadvisor – a container metrics target developed by Google;
- Pushgateway – a metrics cache that does not scrape metrics but offers a REST API to push metrics from the outside to its cache;

To easily provide these metrics to clients and system administrators, Prometheus can integrate with services such as Grafana[10] to display useful information in a user-personalized manner.

Prometheus was design to be deployed in Cloud environments where network constraints are negligible. For optimal use it is required to have high-load, high-frequency communication with its targets so that recent metrics can always return to Prometheus in time for its alerts to be delivered in a useful way.

This constitutes a problem when we transition to Edge environments because nodes can be very far from each other and have sizable network-traffic constraints.

2.5.2 Elastic

Elastic[11] is a company that provides a myriad of open-source services that can be used to build a distributed monitoring system with focus on scalability and query speed. An

[9] https://prometheus.io/docs/instrumenting/exporters/
[10] https://grafana.com
[11] https://www.elastic.co

overview of this system's architecture can be seen in figure 2.15.

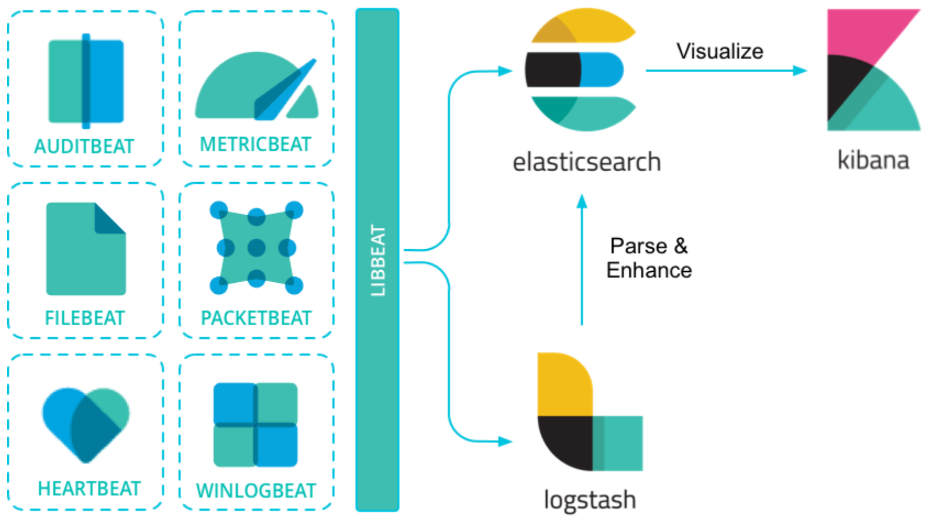

Figure 2.15: Elastic architecture

The main components necessary to build such a system are described below:

- Elasticsearch - this is the main component, it is a No-SQL database built upon the Apache Lucene engine[12], which is known for enabling extremely fast search queries. Its data structure and query language are JSON and the other components can communicate with Elasticsearch via its HTTP endpoint;
- Logstash - a data processing pipeline service that is used to ingest and transform data, so that it can be fed to Elasticsearch in a structured way;
- Beats - are lightweight data shippers that can be deployed independently to collect specific metrics that are then sent to Logstash for pre-processing or directly to Elasticsearch. Some of these Beats collect: logs (Filebeat), system metrics (Metricbeat), network metrics (Packetbeat), windows events (Winlogbeat), etc.;
- Kibana - is a stateless visualization tool that lets the user or system administrator inspect the data collected so far via personalized charts and graphics in real-time;

In terms of service discovery, Elastic does not delegate the responsibility of knowing the system's structure to the bigger components located in the Cloud, instead, when Beats (data shippers) are deployed, the location where data should be sent is configured in install-time. If Elasticsearch or Logstash were to be moved, all the Beats would need to

[12]http://lucene.apache.org/core/

be reconfigured, meaning this system does not handle well with the dynamism of its big components.

2.5.3 Osmotic Monitoring

An Osmotic environment is a new definition that encapsulates Cloud, Edge, and IoT devices' environments. The authors in [17], which came up with this definition, propose a system which is capable of effectively function in such environment.

They propose a composite system with two types of components: monitoring agents, which tend to reside in the outer layers of the system (IoT and Edge) and management agents, which need to be deployed in nodes with considerable resources (Cloud and powerful Edge nodes). Monitoring agents are tasked with the collection of metrics, while management agents' job is to receive those metrics, persist them and make them available to the client through a predefined API.

The communication in this system is done via a push mechanism directed from the outer layers to the inner layers. This way, the monitoring agents can migrate from environment to environment, or simply disappear without the awareness of the management agents, which simplifies the system and increases its mobility.

All monitoring agents in this system extend the basic SmartAgent implementation which is capable of three operations:

- register (HTTP Put) - notifies the management that this monitoring agent just entered the system and will start to send metrics;
- sendData (HTTP Post) - effectively sends data (scraped metrics);
- setConfiguration (HTTP Get) - inquiries the management agent about what configuration it should use and sets them as specified;

This implementation is then extended to provide metrics collection for specific parts of the system. Those extensions are:

- SystemAgent - provides infrastructure (low-level) metrics;
- NetworkAgent - provides network metrics for all networks present in the system;
- ProcessAgent - monitors a specific process through all of its life cycle;
- DeviceAgent - monitors a single Internet of Things device;

2.5.4 Nonintrusive Monitoring

The authors of [15] developed a monitoring system for microservice architectures capable of collecting network metrics without requiring instrumentation of the application components or the use of probes. This is done by adding, to a gateway component, the ability to collect metrics whenever a component requests information to communicate with another component. This method employs the notion of black-box monitoring, which prioritizes the reduction of invasiveness and disruptiveness as it, by principle, requires no modification at the microservice level. The metrics directly gathered by this system are inter-component response time, request origin, request destination and function activated by the request, which can, consequently, be aggregated to determine average response time, system topology and provide an overall service characterization.

To achieve such a system, the authors utilized open-source Netflix[13] and Apache[14] components that provide specific and necessary functionalities, namely:

- Eureka - service discovery component;
- Ribbon - load-balancing component;
- Zuul - gateway component that connects the application and monitoring microservices, giving access to their functionalities. This is the component that was modified to allow the collection of networking metrics;
- JMeter - a component used to simulate heavy load on a application with the objective of measuring its performance;

This architecture also possesses a service registry, a time-series database (InfluxDB[15]) and a visualization tool (Grafana[16]). All these components are containerized using Docker[17] technology and managed using the Docker swarm feature. An overview of the architecture is presented in figure 2.16.

2.5.5 Conclusion

All these systems contribute with implementations or concepts that try to improve the distributed monitoring paradigm but until now they failed to handle successfully the congestion created in the network when too many components are communicating in real-time. Nonintrusive Monitoring offers a good solution to collect network metrics while

[13]https://github.com/netflix
[14]https://www.apache.org
[15]https://www.influxdata.com/products/influxdb-overview/
[16]https://grafana.com
[17]https://www.docker.com

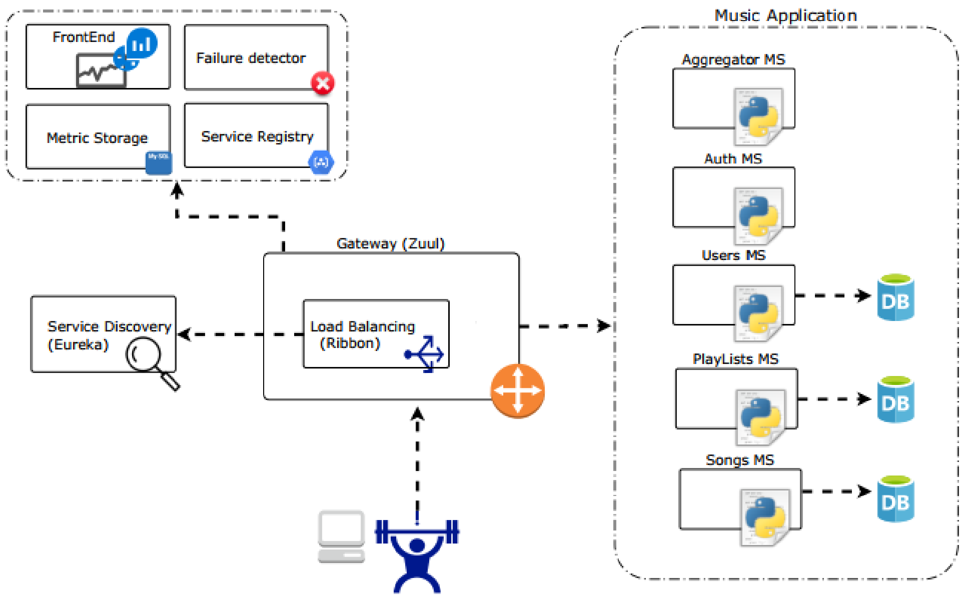

Figure 2.16: System architecture

requiring very few modifications in the original system but does not tackle the network congestion problem and also the variety of metrics collected is small, providing limited insight. In Elastic, Beats communicate via a push mechanism to Elasticsearch or Logstash, and as such, if their number increase drastically, they can cause big enough constraints in the network to halt the system. The only guard Elastic has in these situations is if these Beats send data to Logstash which subsequently send it to Elasticsearch. In doing so, Logstash can throttle the data flow and queue the data that is arriving too fast to be processed and stored directly. This protects the consistency and correctness of the system but does not address the network part of the problem. Osmotic Monitoring also uses the same mechanism to receive metrics and therefore suffers from the same problems. Prometheus used the opposite strategy, its Metric Scrapers send data to the Prometheus component only when requested (pull mechanism) and its frequency can be parameterized to be a big enough interval so that the network does not congest. This can solve the network issue but has the side effect of losing the freshness of the metrics received and increasing the time taken to receive urgent alerts.

Is is for this reason that we developed an architecture that has this notion as a primary concern but without sacrificing the freshness of its metrics. This architecture is presented in the next chapter.

Chapter 3

Proposed Solution

Inspired by the Prometheus architecture, we present a monitoring solution for hybrid infrastructures (Cloud/Edge) that was envisioned to mitigate the problems encountered when transitioning a monitoring system from a Cloud-only solution to a solution in those heterogeneous environments. First, we give an overview of the architecture created and explain its capabilities. This architecture is generic in nature and as such, many implementations can be derived from it. In chapter 4, we implement one of these. Then, we describe each component of the architecture and classify them according to their location in the system. Finally, we explain how each component works and is used to fulfill the system's capabilities.

3.1 Architecture Overview

This architecture is based on the layered topology presented in section 2.4.1. It is composed of three layers, which correspond to the layers of the infrastructure where it will reside: the Cloud, which is made of many homogeneous instances, but can be visualized as one powerful instance with almost unlimited resources (storage and computational power); the Edge, composed of some mid-level instances of heterogeneous nature with reasonable power that are not very far from the Cloud layer; and Devices, composed of many resource-constrained devices, which can be mobile, geographically distributed and therefore, very far from the Cloud layer. A high-level overview of the architecture is shown in figure 3.1.

The architecture's components can be distinguished by their position in the infrastructure. Heavy components need to be hosted in the Cloud node while components with

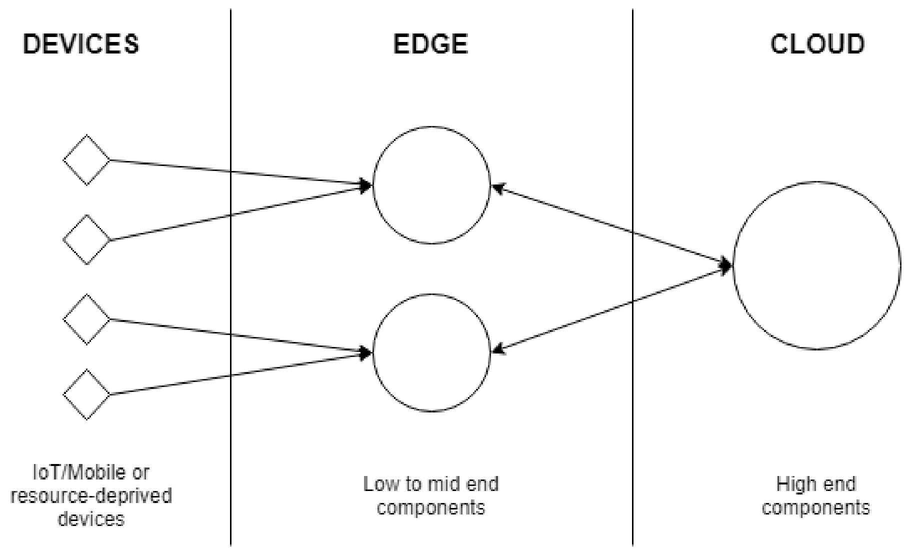
DEVICES
EDGE
CLOUD
IoT/Mobile or resource-deprived devices
Low to mid end components
High end components

Figure 3.1: High level composition of the monitoring architecture.

lower resource requirements can reside in the Edge nodes. The latter are deployed in these nodes with the objective of providing functionalities that, in conventional monitoring systems, were provided by components in Cloud nodes. By doing so, we reduce the Cloud node's required computations and preserve the network used for Edge-Cloud communication. The devices from the Devices layer are used primarily to produce and export metrics to the Edge layer. The functionalities provided by the components of this architecture are explained in details in section 3.2.

The strategic positioning of these components (by employing our architecture) creates a hierarchical structure supported by the infrastructure, with one powerful Cloud node (which can be an abstraction of many Cloud nodes) communicating with many less powerful Edge nodes. Each of these Edge nodes also communicate with devices of the Devices layer located in their vicinity. This tree-like structure facilitates the distribution of information in the form of metrics and alerts while also controlling the amount of information that is sent upwards through the system.

By creating an architecture that considers the Cloud as the top layer, Devices as the bottom layer and Edge as the intermediary layer between the two we can reduce the network load by limiting communication to the Cloud. Only Edge instances communicate with the Cloud node, and do so with a frequency that does not congest the network. In alignment with this strategy, the devices from the Devices layer only communicate with the Edge node closest to them, which possibilitates a much more frequent communication and controls the amount of data that gets sent to the Cloud. Therefore, the Edge layer functions as a bridge between the Devices and Cloud layers, reducing the network and data loads that are sustained by the Cloud node, making the systems using this architecture in hybrid environments run more smoothly and efficiently.

Communication is done, in its majority, through REST API's and the components communicate directly with each other. This architecture also takes into consideration the communication techniques called push and pull. Push is when a message is directly sent to the recipient without previous notice. In contrast, pull is when the recipient asks the sender to send any message they have previously accorded. Pull is safer as it allows control over the frequency of which messages are received. By imposing a message frequency, we can assure that the network connection remains healthy and no overloads occur. For this reason, our architecture utilizes pull as its primary communication technique while push is only used by specific components. A more detailed explanation of when both these mechanisms are used is presented in section 3.2.

Figure 3.2 shows a mid-level overview of the architecture in which the interaction with the end-users is illustrated. The components in the figure (which we will address in chapter 4) are a materialization of the conceptual components explained in section 3.2. The users can communicate directly with the components of the Cloud layer to visualize the global state of the metrics in the system and also to receive alerts in real-time. Edge nodes are also available for interaction with the objective of managing individual node settings and visualizing local metrics stored by the node being accessed.

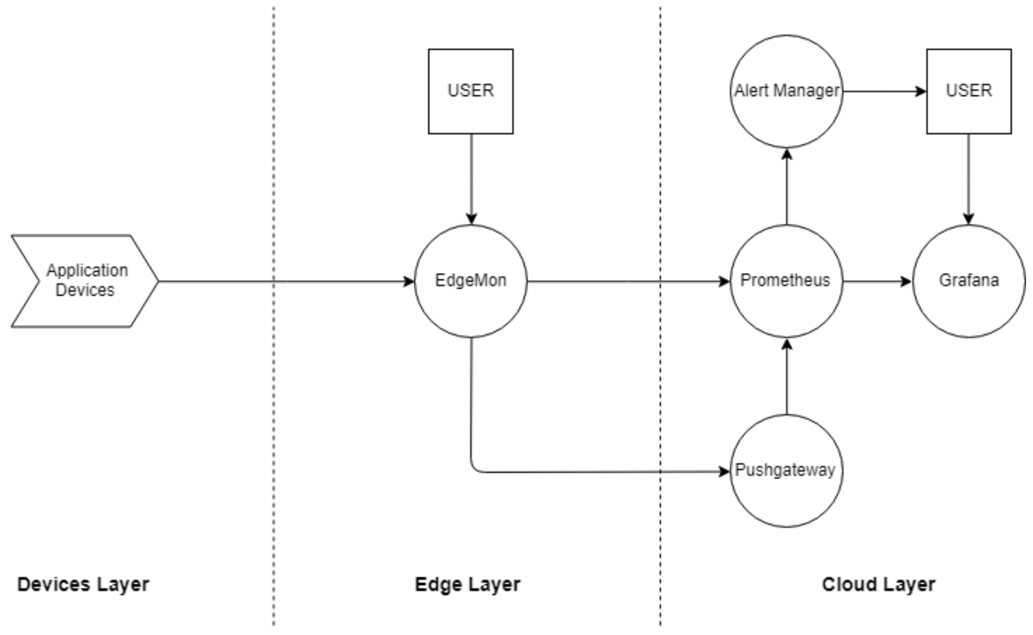

Figure 3.2: Mid-level overview of the monitoring architecture, highlighting the possible end-user interactions with the system.

A system following the architecture presented should be capable of:

- Metrics Acquisition and Storage - metrics are collected primarily through the use of specialized components that collect both high level/application and low level/-machine metrics using the pull mechanism. These components reside both in the Cloud and the Edge layers and therefore collect metrics generated in these parts of the infrastructure. To collect metrics from the Devices layer, it is necessary that the components from the application being monitored that reside there, send their metrics using the push mechanism to a specific component located in the nearest Edge node. The system does not differentiate between the type of metrics being collected/stored and deals with them in the exact same way. All metrics of the system are persistently stored in the Cloud node. Metrics that were collected from the Edge and Devices layers are first temporarily saved in caches located at the Edge nodes before being sent to the Cloud database.

- Metrics Processing - the system is capable of applying aggregation functions (for example average, maximum, minimum, count, etc.) to the metrics stored in the Cloud node, thus creating new metrics which are also kept in the database.

- Metrics Filtering - the metrics sent from each individual Edge node can be dynamically configured so that only a relevant subset of all the metrics kept in that Edge node are forwarded to the Cloud layer. This filtering allows the reduction of network congestion and storage requirements by limiting what metrics effectively arrive at the Cloud layer at specific moments of the system's life.

- Rule Checking and Alerting - metrics are periodically checked in the Cloud node using a complex and resource heavy rule engine (the rules definition and checking periodicity are parameterized by the person tasked to configure the system before its deployment). When a rule is fired, an alert is immediately send to a human via a human interface which can be e-mail, SMS, etc. There is also rule checking in each Edge node, done by a lightweight rule engine that only allows simple threshold checking rules (manually defined at deploy time). Upon activation, an alert is sent from the affected Edge node, using the push mechanism, to the Cloud node which in turn, uses its mechanisms to forward it to a human. The rule checking engine present in every Edge node has two settings regarding the periodicity of which it checks metrics. It can be activated every time new metrics are collected and only these metrics are checked or it can check all the existing metrics in a dynamically defined time interval. These two options can be in effect simultaneously.

  The reason for having a different alerting method in the Edge layer is to reduce network congestion. For the alert system to be useful, it must send alerts in a meaningful time window. To achieve this for the Edge layer, a constant and very frequent stream of metric requests (pull mechanism) from all the Edge nodes to the Cloud node would be necessary, resulting in extremely high network traffic caused by the number of Edge nodes and their distance from the Cloud layer. Instead, by using the push mechanism the system is capable of receiving real-time alerts without generating big amounts of network traffic. These alerts, when generated in the Edge layer pertain to specific application or infrastructural occurrences associated with the alerting node (response time, CPU/memory load, etc.), while those generated in the Cloud layer pertain to system-wide occurrences (overall health of the system, average response time, etc.).

- Visualization - the system allows the visualization of the global metrics that are persisted in the Cloud and also the local metrics cached in each Edge node (which may not have yet been forwarded to the Cloud or will never be forwarded). The access to these metrics is accomplished through a REST API. This functionality is intended for end-users (clients or system administrators) and can be exposed in a browser in the form of graphs, charts and other visualization methods.

3.2 Description of Cloud and Edge Components

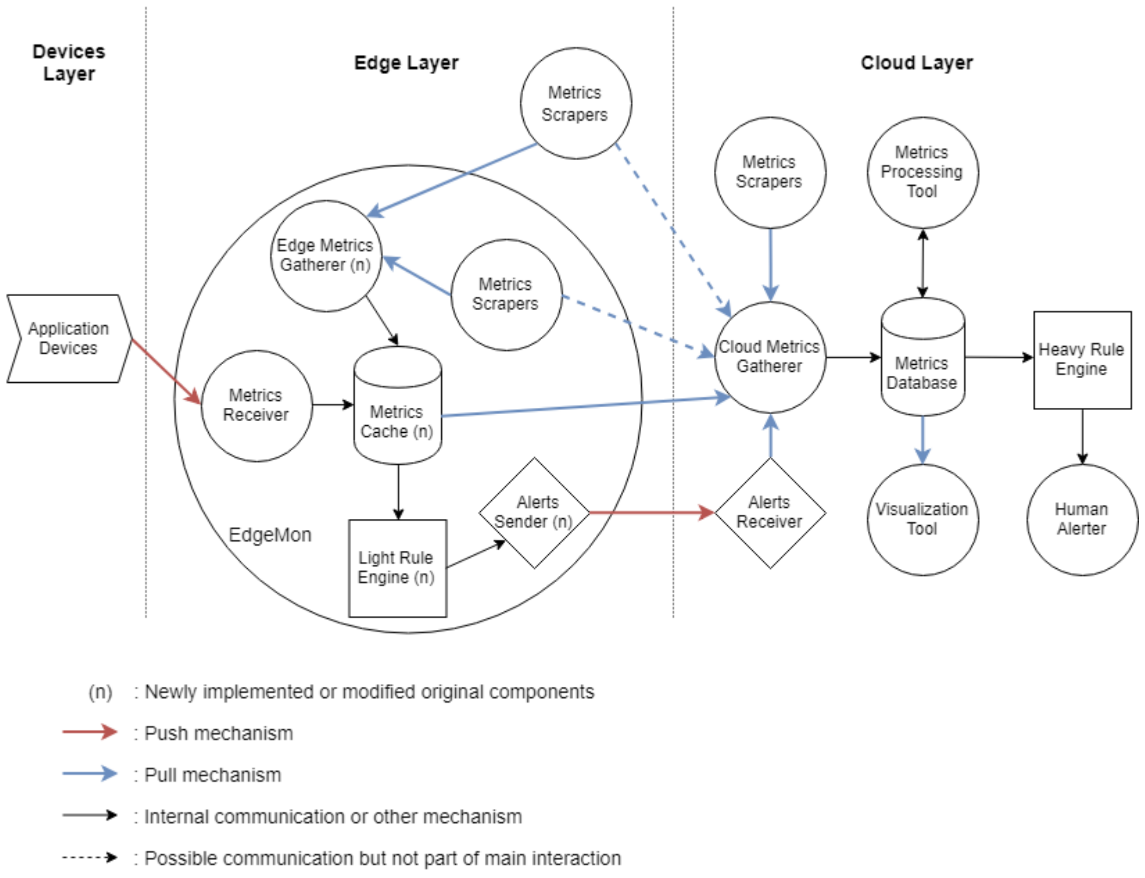

Figure 3.3: Composition of the monitoring architecture. The arrows represent the moving direction of the metrics and alerts in the system.

In this section, we divide the components by their location in the infrastructure (Cloud and Edge) and proceed to explain their purpose (for the mapping of these components to the system's functionalities, see section 3.3). A diagram of the entire architecture and its components is shown in figure 3.3.

Cloud components:

- Metrics Database - Component in charge of storing all collected metrics persistently;
- Metrics Processing Tool - Component executing aggregation functions that act upon the stored metrics in order to produce new metrics. These functions, along with the processing periodicity must be defined before deploying the system;
- Visualization Tool - Tool capable of producing visual representations of the existing metrics. Provides a graphical interface for end-users to interact;
- Metrics Scraper (Cloud) - Component that produces specific metrics (application, network, node, etc.) and sends them to the Cloud Metrics Gatherer upon request (responds to pull requests);
- Cloud Metrics Gatherer - Component capable of requesting and gathering metrics produced by all Metric Scrapers located in the Cloud and Edge layers (using the pull mechanism);
- Heavy Rule Engine - Engine that evaluates the metrics present in the Metrics Database against predefined rules (configured before deployment) and instructs the Human Alerter to send alerts. The rules can be of significant complexity and therefore require the engine to operate in a high-resources instance;
- Human Alerter - Component that is in charge of forwarding system alerts to predefined human interfaces (e-mail, pager, GUI, etc.);
- Alerts Receiver - Component that receives alerts (via the push mechanism) from the Edge layer and stores them persistently or not. This component is also considered a Metrics Scraper by the system so that the Cloud Metrics Gatherer can gather its metrics seamlessly;

Edge Components:

- Metrics Cache - Component that temporarily stores metrics from the Edge layer. Viewed by the upper layer system as a Metrics Scraper;
- Metrics Scraper (Edge) - Similar to the component of the same name in the Cloud layer but now tasked to produce metrics in the Edge layer;
- Edge Metrics Gatherer - Component similar to the Cloud Metrics Gatherer but with the particularity of only gathering metrics from Edge-located Metrics Scrapers. The Metrics Scrapers location must be preconfigured;

- Light Rule Engine - Similar in concept to the Heavy Rule Engine from the Cloud layer but simpler and lighter (with simpler rules) to accommodate the Edge layer resource restrictions. Checks metrics present in the Metrics Cache;

- Alerts Sender - Complementary component to the Alerts Receiver in the Cloud layer. This components sends alerts (using the push mechanism) to the Cloud node (specifically to the Alerts Receiver) when some metric fires a specific rule imposed on the Light Rule Engine;

- Metrics Receiver - Component capable of receiving metrics sent from application components (using the push mechanism). This is useful for parts of the monitored application that reside in extremely resource-deprived nodes where monitoring components cannot be deployed and, in alternative, send their metrics to this component.

3.3 Components' Responsability in System Capabilities

In this section we explain how the existing components function and interact in order to fulfill the system capabilities enunciated in section 3.1.

3.3.1 Metrics Acquisition and Storage

The number of metrics acquired by the system is directly dependent on the number of active Metrics Scrapers and the frequency of requests sent from the Cloud Metrics Gatherer. To acquire metrics, first the Metrics Scrapers must produce them, which they do constantly; then the Cloud Metrics Gatherer residing in the Cloud layer requests them periodically and when they are received, they are stored in the Metrics Database for safekeeping and future analysis. This method is also used for the Metrics Scrapers from the Edge layer, particularly, the Metrics Cache, which is populated by the Metrics Receiver and Edge Metrics Gatherer. This Edge Metrics Gatherer also gathers metrics from the Edge layer and stores them in the Metrics Cache to enable the Light Rule Engine to act upon them, providing a real-time alerting functionality of the Edge layer.

There is also the special case of wanting to collect metrics coming from Edge nodes with low resources available or from devices from the Devices layer. These types of nodes/devices cannot host any monitoring tool and, as such, the application itself must be instrumented in order to scrape metrics. Then, these metrics can be send, using the push mechanism, to the Metrics Receiver which is scraped by the Edge Metrics Gatherer that, finally, stores the metrics in the Metrics Cache.

3.3.2 Metrics Processing

After the Metrics Database have been populated, the Metrics Processing Tool can periodically apply aggregation functions to the metrics, which in turn produce new metrics that are again stored in the Metrics Database.

3.3.3 Metrics Filtering

The Metrics Cache is responsible for sending, upon request, the metrics it has saved to the Cloud Metrics Gatherer residing in the Cloud layer. Which metrics are effectively sent, can be dynamically configured in the Metrics Cache, so that only a relevant subset of its metrics pool arrives at the Cloud node (first to the Cloud Metrics Gatherer and then to the Metrics Database).

3.3.4 Rule Checking and Alerting

In this system, alerting works differently depending on the layer it is occurring (Cloud/Edge). It happens in the Cloud layer when the Heavy Rule Engine is fired by checking the rules against the metrics present in the Metrics Database. In this instance a message is sent, notifying the Human Alerter, which in turn, forwards it to a human. In the Edge layer, the Light Rule Engine can periodically or reactively (every time new metrics are collected and deposited in the Metrics Cache) check the metrics present in the Metrics Cache and when it is fired, it sends an alert to the Alerts Sender, which subsequently sends it to the Alerts Receiver in the Cloud layer (using the push mechanism). As the Alerts Receiver is viewed by the system as a Metrics Scraper, its alerts are also gathered by the Cloud Metrics Gatherer and stored in the Metrics Database. These special metrics are tailored to immediately fire a rule from the Heavy Rule Engine, which in turn, starts the Cloud alerting process.

3.3.5 Visualization

To enable the view and analysis of the metrics created by the system to users and system administrators, the Visualization Tool collects the metrics that are stored in the Metrics Database and creates graphs and other visualization techniques to further fulfill its objective.

CHAPTER 4

IMPLEMENTATION

This chapter explains how the architecture described in chapter 3 was implemented, what technologies were used and specifically which parts were developed. This implementation was also used to conduct tests which are discussed in chapter 5.

4.1 Architecture Implementation

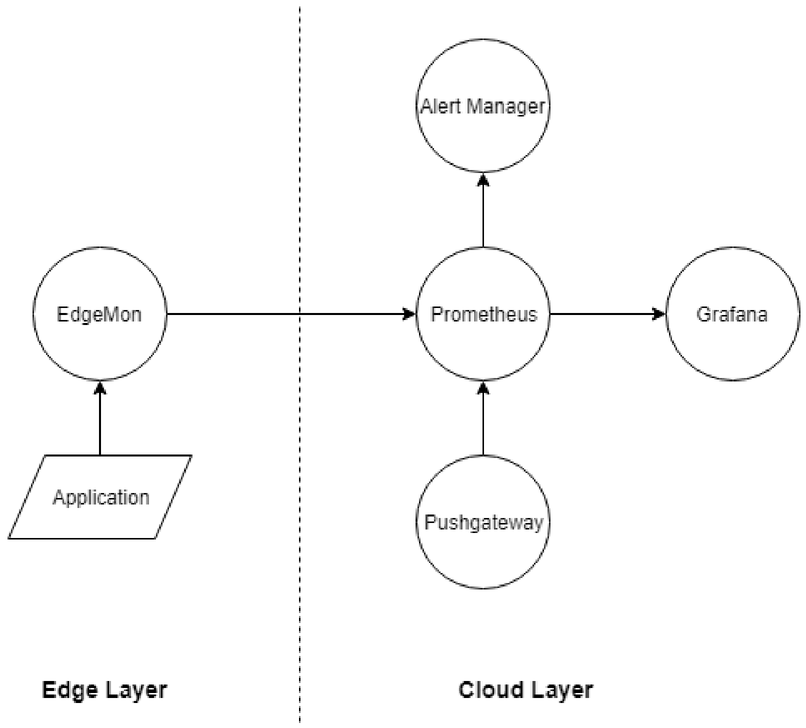

Figure 4.1: Implemented architecture schematic spanning both layers.

The components used to construct this architecture come mainly, or are extended, from implementations that were downloaded from the community-driven, open-source, Prometheus repository[1] and are containerized and deployed using Docker technology. The schema of the implemented architecture can be viewed in figure 4.1.

4.1.1 Components

Prometheus as explained in subsection 2.5.1 is a monitoring system focused on monitoring distributed applications using microservices as its building blocks. Some of its principal components that are also relevant to this implementation are:

- Prometheus - this is the main component of the system, very heavy resource-wise but with many functionalities. Namely, metrics acquisition, storage, processing and evaluation are the relevant functionalities for this architecture;
- Alert Manager - dispatches received alerts to human interfaces. Typically is deployed in conjunction with Prometheus;
- Node Exporter - general-purpose low-level metrics scraper;
- Pushgateway - metrics' storage capable of receiving metrics via the push mechanism.

Other used components are:

- Grafana[2] - open-source visualization tool with feature rich metrics dashboards and graph editor;
- EdgeMon - newly developed component to reside in the Edge layer that can act as a lighter Prometheus main component for this layer. It can store and collect metrics, evaluate them against predefined rules and send alerts accordingly. The component is further explained in section 4.2.

4.1.2 Correlation of Components

This subsection addresses how to correlate the conceptual components described in chapter 3 to the real ones presented in subsection 4.1.1. We group them based on their location in the infrastructure, as was similarly done in section 3.2.

Cloud layer components:

[1]https://prometheus.io/docs/instrumenting/exporters/
[2]https://grafana.com/grafana

The Prometheus self-titled component has an array of capabilities, namely it can perform the roles of Metrics Database, Heavy Rule Engine, Metrics Processing Tool and Cloud Metrics Gatherer. If we need Metrics Scrapers in this layer (for instance, because we have application components residing in the Cloud) we use Node Exporters. Finally we use a Pushgateway as Alerts Receiver, an Alert Manager as Human Alerter and Grafana as a Visualization Tool.

Edge layer components:

In this layer we basically use various instances of EdgeMon in the nodes that we want to monitor. EdgeMon is capable of being a Metrics Scraper, Edge Metrics Gatherer, Metrics Cache, Light Rule Engine, Alerts Sender and Metrics Receiver. If the collection of metric types not supported by EdgeMon is required, other Metric Scrapers could also be deployed in this layer but in our implementation, this is not done. For the collection of application metrics or metrics coming from resource-deprived nodes (Devices layer), it is necessary to instrument the code of these components, so that they can send their metrics to EdgeMon using the push mechanism.

4.1.3 Docker

A monitoring system that follows this architecture has many moving parts and therefore, will require a method to describe and deploy it in a effective way. There is where Docker becomes very useful. We encapsulate each component of our architecture inside a Docker container (section 2.1.3 explains this method) so it can be easily deployed and managed.

We can utilize a data serialization language supported by Docker called YAML[3] to describe very specifically how to deploy each component (as a Docker container) and in which node. Then we can transform the available worker nodes of our infrastructure into a swarm (using Docker swarm functionality[4]). This functionality is used by entering in each node (via ssh or other method) and executing a Docker swarm command that adds the node to the swarm (nodes can only be a part of a single swarm). This way Docker has a sense of our working infrastructure and can be tasked to deploy our system when provided with the YAML configuration file. The YAML file used in this dissertation to deploy our monitoring system is shown in listing 4.1. This file contains the definition of all the services(components) being deployed, their Docker image, placement in the infrastructure, ports, the network used for communication and other additional configurations.

Listing 4.1: YAML configuration file

```
version: '3'

networks:
```

[3] https://yaml.org

[4] https://docs.docker.com/engine/swarm/

```
    monitoring:
      driver: overlay

services:

    grafana:
        image: grafana/grafana
        networks:
          - monitoring
        ports:
          - 3000:3000
        deploy:
          placement:
            constraints: [node.hostname == node13]

    prometheus:
        image: prom/prometheus
        networks:
          - monitoring
        volumes:
          - ./prometheus.yml:/etc/prometheus/prometheus.yml
          - ./rules/alert.rules_nodes.yml:/etc/prometheus/alert.rules_nodes.yml
          - ./rules/alert.rules_tasks.yml:/etc/prometheus/alert.rules_tasks.yml
        command:
          - '--config.file=/etc/prometheus/prometheus.yml'
        ports:
          - 9090:9090
        deploy:
          placement:
            constraints: [node.hostname == node13]

    alertmanager:
        image: prom/alertmanager
        networks:
          - monitoring
        ports:
          - 9093:9093
        volumes:
          - ./alertmanager.yml:/etc/alertmanager/config.yml
        command:
          - '--config.file=/etc/alertmanager/config.yml'
        deploy:
          placement:
            constraints: [node.hostname == node13]

    pushgateway:
        image: prom/pushgateway
        networks:
          - monitoring
        ports:
          - 9091:9091
        deploy:
          placement:
            constraints: [node.hostname == node13]

    edgemon:
        image: edge:1.0
        networks:
          - monitoring
        ports:
          - 9999:9999
        deploy:
          placement:
            constraints: [node.hostname == node13]
```

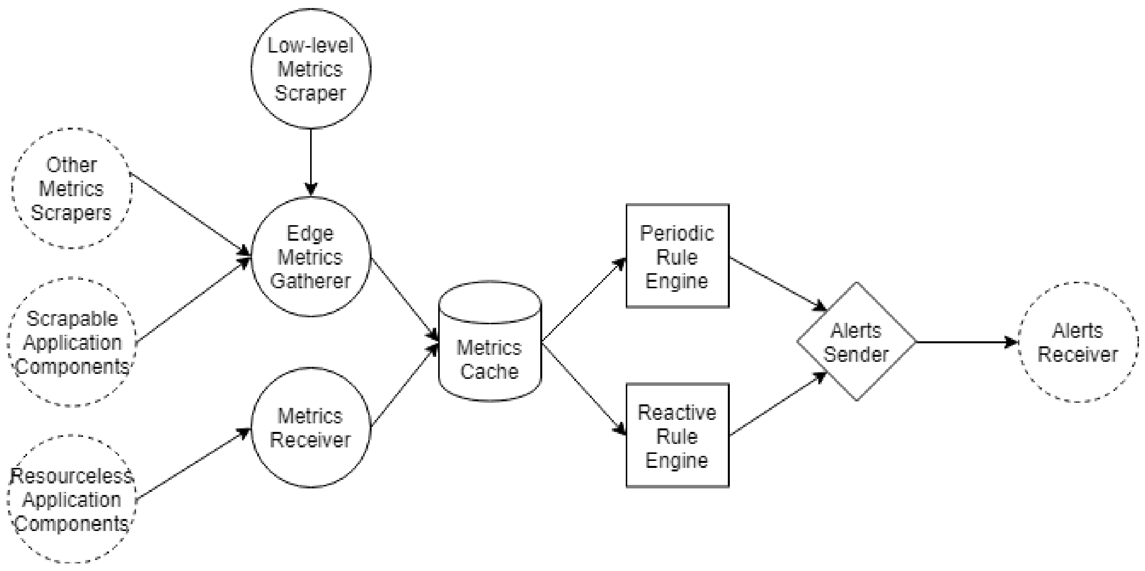

Figure 4.2: Internal composition of the EdgeMon component. Dash lined components are not a part of EdgeMon but interact with it directly. The arrows represent the moving direction of the metrics and alerts in the system.

4.2 New Component: EdgeMon

A component capable of fulfilling the capabilities envisioned by the proposed architecture for the Edge layer was lacking, which is why we built our own. Capabilities such as real-time alerting with minimal network usage, temporarily saving metrics in the Edge nodes and lightweight rule-checking of these metrics were the primary to be considered. We called this new component EdgeMon (Edge Monitoring) because its purpose is to provide supporting functionalities for a monitoring system in the Edge layer. It was written using the Go programming language and is the result of combining two components from the Prometheus repository (Node Exporter and Pushgateway) plus some new code to fill the still missing funtionalities. A diagram of its internal composition is shown in figure 4.2.

EdgeMon provides a list of API endpoints, so that exterior components can communicate with it and activate some of its functionalities, such as sending collected metrics, change metric filters, manage its Metrics Cache, activate and configure its Rule Engine, request application scraping, etc. This functionalities are further explained in subsections 4.2.1 through 4.2.4. It also provides a list of flags that configure some parameters to be used when the component is initialized, such as the port from which EdgeMon is exposed, its REST endpoints and rule engine configurations.

4.2.1 Metrics Scraping

This component incorporates a Node Exporter, which is capable of scraping low-level metrics from the node where it is deployed. In addition, it can actively collect metrics from application components using their REST API, provided that their location is known (using the pull mechanism). Because of the fact that EdgeMon also incorporates a Pushgateway, it is capable of receiving metrics via a REST API, but now using the push mechanism, meaning it passively receives metrics from senders without requesting them.

When there is an interest to collect metrics given by a specific component of the application being monitored, the */cache/app* endpoint can be called. It scrapes a component whose location is hardcodded in the function and adds the metrics to the Metrics Store, which subsequently can be collected by requesting the metrics found in the Metrics Store. This implementation is a proof of concept and could be easily extended by providing, at initialization-time, a file with the complete set of locations pertaining to components whose metrics are desired.

4.2.2 Metrics Filtering

When Prometheus wishes to gather metrics from the EdgeMon, a call to the endpoint */metrics* is done, returning the metrics filtered based on the internal ScrapeMode variable. The value of this variable can be changed via the endpoint */scrape/*mode*, where the mode can be *full*, *node* or *store*. With the *node* configuration, EdgeMon uses the *prometheus/node_exporter/collector* library to collect low-level metrics and directly returns them to Prometheus using the *prometheus/client_golang/prometheus* library. The *store* setting is used when only the contents of the Metrics Store are of interest and the *full* setting executes both of these processes. The code for the metricsHandler function, which is called whenever the */metrics* endpoint is activated, is shown in listing 4.2. In this listing it can be seen how the ScrapeMode variable conditions the metrics that are returned to the entity requesting them. It is also worth mentioning that the act of serving the metrics is delegated to a Prometheus library, which facilitates the development of this type of scraping components.

Listing 4.2: metricsHandler function

```go
func metricsHandler(w http.ResponseWriter, r *http.Request) {
  registry := prometheus.NewRegistry()
  var gatherers prometheus.Gatherers

  if scrapeMode == "/full" || scrapeMode == "/node" {
    filters := r.URL.Query()["collect[]"]
    log.Debugln("collect query:", filters)

    nc, err := collector.NewNodeCollector(filters...)
    if err != nil {
      log.Warnln("Couldn't create", err)
      w.WriteHeader(http.StatusBadRequest)
      w.Write([]byte(fmt.Sprintf("Couldn't create %s", err)))
      return
    }

    err = registry.Register(nc)
    if err != nil {
      log.Errorln("Couldn't register collector:", err)
      w.WriteHeader(http.StatusInternalServerError)
      w.Write([]byte(fmt.Sprintf("Couldn't register collector: %s", err)))
      return
```

```
        }
    }

    if scrapeMode == "/store" {
        // Inject the metric families returned by ms.GetMetricFamilies into the default Gatherer:
        gatherers = prometheus.Gatherers{
            prometheus.GathererFunc(func() ([]*dto.MetricFamily, error) { return ms.GetMetricFamilies(), nil }),
        }
    } else if scrapeMode == "/node" {
        gatherers = prometheus.Gatherers{
            registry,
            prometheus.DefaultGatherer,
        }
    } else {
        gatherers = prometheus.Gatherers{
            prometheus.GathererFunc(func() ([]*dto.MetricFamily, error) { return ms.GetMetricFamilies(), nil }),
            prometheus.DefaultGatherer,
            registry,
        }
    }

    // Delegate http serving to Prometheus client library, which will call collector.Collect.
    h := promhttp.InstrumentMetricHandler(
        registry,
        promhttp.HandlerFor(gatherers,
            promhttp.HandlerOpts{
                ErrorLog:      log.NewErrorLogger(),
                ErrorHandling: promhttp.ContinueOnError,
            }),
    )
    h.ServeHTTP(w, r)
}
```

4.2.3 Metrics Storing

As mentioned before, EdgeMon incorporates a Pushgateway that enables storing metrics in memory (serving as a cache for the monitoring system in the Edge layer). This cache is where the metrics collected by its metrics scraping mechanisms are stored and can later be delivered to Prometheus at the Cloud layer when requested. The storing ability also fills a hole in the Node Exporter concept in which metrics are only scraped when requested and delivered immediately. This means they are never stored in the cache and therefore can never be checked against rules locally upon request. The endpoint created to store these low-level metrics in the Metrics Store is */cache/node*.

The Metric Store from Pushgateway was extended using the Wrapper design pattern so that whenever new metrics were added, they would first pass through the Rule Engine and therefore be checked immediately. This is particularly useful because, this way, the Rule Engine only checks the freshly arrived metrics instead of all the metrics present in the Metrics Store, preventing unnecessary computations but still providing timely alerts. This also means that when the endpoints */cache/node* and */cache/app* are called to cache the low-level and application metrics respectively, they are also immediately checked. A snippet of the extended Metrics Cache (MonitoredMetricStore) is displayed in listing 4.3

to demonstrate the use of the Wrapper design pattern and how it adds the rule checking funcionality (which is explained in subsection 4.2.4) in the SubmitWriteRequest function.

Listing 4.3: MonitoredMetricStore

```go
//MonitoredMetricStore a DiskMetricStore with monitoring capabilities
type MonitoredMetricStore struct {
  ms *storage.DiskMetricStore
}

//NewMonitoredMetricStore Instanciates a new monitored DiskMetricStore
func NewMonitoredMetricStore(
   persistenceFile string,
   persistenceInterval time.Duration,
) *MonitoredMetricStore {
  dms := storage.NewDiskMetricStore(persistenceFile, persistenceInterval)
  mms := &MonitoredMetricStore{}
  mms.ms = dms
  return mms
}

// SubmitWriteRequest implements the MetricStore interface.
func (mms *MonitoredMetricStore) SubmitWriteRequest(req storage.WriteRequest) {
  if checkReactiveMonitoring() {
    metricFamilies := req.MetricFamilies

    mfArray := make([]*dto.MetricFamily, len(metricFamilies))
    for _, metric := range metricFamilies {
      mfArray = append(mfArray, metric)
    }
    startRuleEngine(mfArray)
  }
  mms.ms.SubmitWriteRequest(req)
}

// GetMetricFamilies implements the MetricStore interface.
func (mms *MonitoredMetricStore) GetMetricFamilies() []*dto.MetricFamily {
  return mms.ms.GetMetricFamilies()
}
```

4.2.4 Rule Checking and Alerting

Having metrics stored locally enables the ability to run them against a rule-checking engine. To create such a rule engine we followed Martin Fowler's article [5] on the matter and implemented a very simple and lightweight engine that checks metrics periodically or on-demand (these definitions can be set using the REST API provided by EdgeMon). When some metric goes above or below a certain threshold defined by a rule, the engine fires, and an alert, using the push mechanism is sent to the Pushgateway residing in the Cloud layer to notify the monitoring system of such occurrence. The function responsible with sending alerts to the Cloud node is shown in listing 4.4. This function uses the Prometheus libraries (prometheus and push) to create the alert (which is, in reality, a metric) and push it to the upper layer.

The two types of Rule Engines, as shown in figure 4.2, are called Reactive Rule Engine and Periodic Rule Engine. The Reactive Rule Engine checks the metrics whenever they

[5] https://www.martinfowler.com/bliki/RulesEngine.html

enter the Metrics Store. It can be toggled using the */rules/reactive/*state* endpoint, where the state can be *on* or *off*, or via the *rules.reactive* boolean flag, which can be set at start-up.

The Periodic Rule Engine checks all the metrics in the Metrics Store periodically. It can also be toggled with the *rules.periodic* flag or parameterized by an interval of activation using the */rules/periodic/*interval* endpoint. The interval is the number of seconds in between each check and 0 stops the Rule Engine. The Periodic Rule Engine runs separately from the main execution of the service. It runs in a goroutine, which is a lightweight thread managed by the Go runtime. The communication with this Engine, either to change the checking interval or to turn it off requires the use of a channel, a built-in Go data structure that enables synchronously sending data between different goroutines.

The rules defined for these Engines are currently hardcodded in the source code but can be easily changed so that, upon startup, a file containing all the rules is read and stored in memory. The code for the function that starts the Reactive Rule Engine can be seen in listing 4.5 along with the hardcodded rules present in lines 8, 13 and 18.

Listing 4.4: SendAlert: function responsible for sending alerts to the Cloud Layer

```
//SendAlert Sends an alert to the alertAddress
func SendAlert(alertName string, alertHelp string, fromReactive bool) {

  if fromReactive {
    fmt.Printf("Sending reactive alert: %s\n", alertName)
  } else {
    fmt.Printf("Sending periodic alert: %s\n", alertName)
  }

  alert := prometheus.NewCounter(prometheus.CounterOpts{
    Name: alertName,
    Help: alertHelp,
  })
  alert.Inc()

  registry := prometheus.NewRegistry()
  registry.MustRegister(alert)

  pusher := push.New(*alertAddress, "edge_alerts").Gatherer(registry)

  if err := pusher.Add(); err != nil {
    fmt.Println("Could not push to Pushgateway:", err)
  }
}
```

Listing 4.5: startRuleEngine: function responsible for starting the Reactive Rule Engine

```
func startRuleEngine(families []*dto.MetricFamily) {
  re := core.NewDefaultRulesEngine()

  for _, family := range families {
    switch family.GetName() {
    case "cpu_load_pushed":
      for _, metric := range family.GetMetric() {
        cr := newMetricRule(metric, true, 75, "cpu_overload_", "cpu␣usage␣over␣defined␣threshold␣on␣")
        re.AddRule(cr)
      }
    case "node_memory_Active_bytes":
      for _, metric := range family.GetMetric() {
        cr := newMetricRule(metric, true, 3.5e+09, "memory_active_bytes_overload_", "active␣memory␣bytes␣over␣defined␣threshold")
        re.AddRule(cr)
      }
    case "node_cpu_frequency_hertz":
      for _, metric := range family.GetMetric() {
        cr := newMetricRule(metric, true, 2.5e+09, "cpu_frequency_hertz_overload_", "cpu␣frequency␣over␣defined␣threshold␣on␣cpu␣")
        re.AddRule(cr)
      }
    }
  }

  err := re.FireRules()
  if err != nil {
    fmt.Println("Rules␣failed!")
  }
}
```

Chapter 5

Validation

This chapter focus on how we tested the benefits of employing our architecture on a distributed and hybrid monitoring system. First, we address the technologies used and how the whole experiment was set up. Then, we describe all the scenarios tested along with the reasons for doing so. Finally, we show the tests' results and elaborate on their interpretation.

5.1 Technologies Used

This section addresses the technologies that were used, in conjunction with the implemented monitoring system presented in chapter 4, for testing purposes. Namely, a demo application called Sockshop and a testing component called Load-test.

5.1.1 Sockshop

On prior chapters, we repeatedly talked about the application being monitored as an abstract concept. Now, to be able to test the effectiveness of our architecture we needed a concrete application to monitor and so we used one called Sockshop[1]. Sockshop is a sock shop, primarily used for testing purposes with no real life production value. It was developed by Weaveworks[2] and is composed of several microservices, written in different programming languages. It can be deployed using docker-compose, similarly to our monitoring system, which is the main reason for choosing this method of deployment (through a YAML configuration file). Communication with this application is done through its Frontend microservice via a REST API, which is the same method the other

[1]https://microservices-demo.github.io
[2]https://www.weave.works

composing microservices use to communicate with each other. A complete ilustration of the Sockshop's architecture along with the programming languages used to develop each individual microservice is shown in figure 5.1.

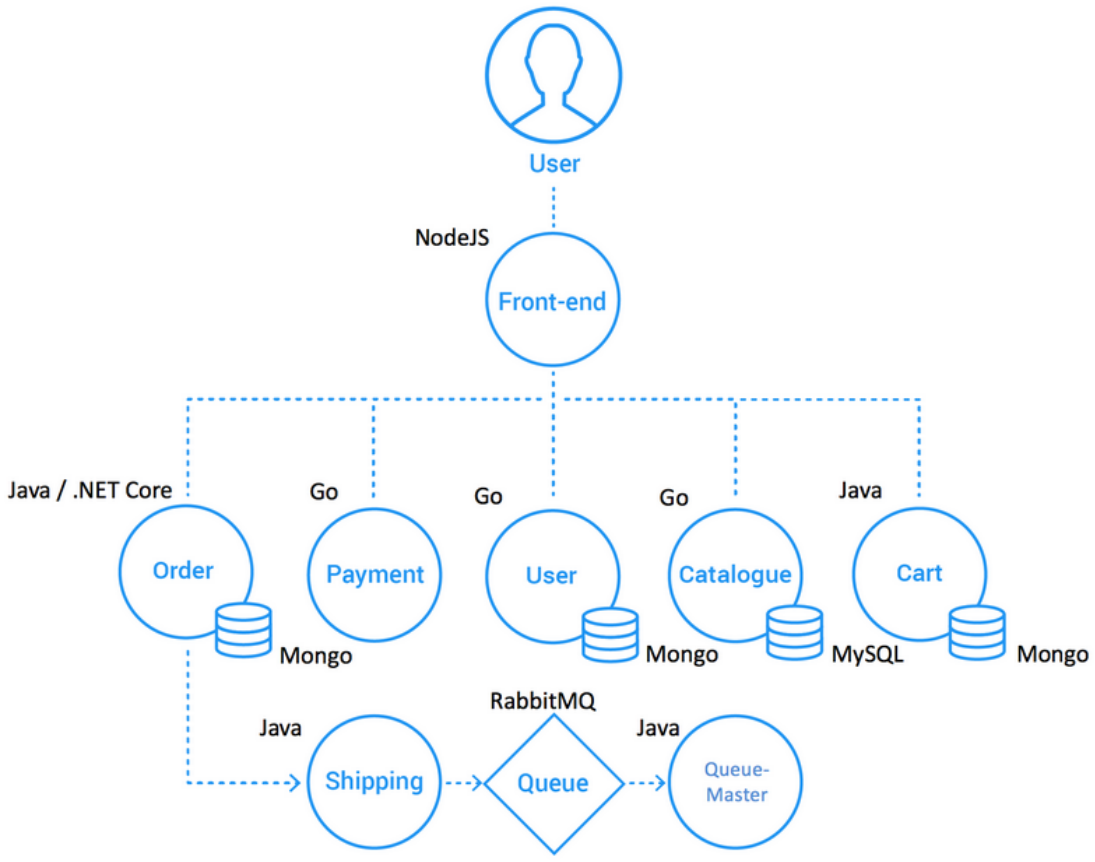

Figure 5.1: Sockshop's Architecture.

5.1.2 Load-test

The test used in the scenarios described in section 5.2, consisted in running a script that executed a tester microservice while measuring the data transferred (input and output) and memory usage by each component of the system (application and monitoring system) via the docker stats functionality[3]. This microservice is called Load-test[4] and was also developed by Weaveworks to accompany Sockshop, allowing tests such as this one to be done. In summary, what Load-test does is emulate x number of users doing y number of requests simultaneously, x and y being values inserted as parameters. The script used for testing can be seen in listing 5.1.

[3]https://docs.docker.com/engine/reference/commandline/stats/

[4]https://microservices-demo.github.io/docs/load-test.html

Listing 5.1: Test script

```
#!/bin/bash

FORMAT="'{{.Name}}\t{{.CPUPerc}}\t{{.MemUsage}}\t{{.NetIO}}\t{{.BlockIO}}'"
FRONTEND=$1
CLIENTS=$2
REQUESTS=$3

echo "### Cloud before load-test ###" > result
ssh $FRONTEND docker stats --format $FORMAT --no-stream >> result

docker run --net=host weaveworksdemos/load-test -h $FRONTEND -r $REQUESTS -c $CLIENTS

printf "\n### Cloud after load-test ###\n" >> result
ssh $FRONTEND docker stats --format $FORMAT --no-stream >> result
```

5.2 Scenarios' Description

In this section we explain the scenarios used for testing, their relevance for the overall purpose of this dissertation and the infrastructure upon which they were executed. The scenarios are as follows:

1. A stand-alone Sockshop with a varying number of clients and requests;
2. Prometheus scraping Sockshop with pull frequency of 1 and 5 seconds;
3. Prometheus scraping Node Exporter, which is collecting low-level metrics from nodes where Sockshop is residing with a pull frequency of 1 and 5 seconds;
4. Similar to 3. but using EdgeMon instead of Node Exporter.

Each subsection describes a scenario and contains an illustrative figure. In all applicable scenarios, both the monitoring system and the monitored application were deployed in the same node. The reason for this decision is because the cluster where the tests were executed is composed of relatively powerful nodes, connected by fast ethernet links (a more detailed description of the testing environment is presented in subsection 5.2.5). These characteristics make the communication between nodes in the cluster quite fast and therefore, using different nodes for testing would not differentiate very much the measured times. Furthermore, our tests had a greater focus on the amount of data transferred between components and compared the measured times relatively to each other, instead of focusing on their absolute value.

5.2.1 Stand-alone Sockshop

This first scenario served as a reference to compare against the following ones. It consisted of using the test script on the Sockshop without any other component present. This process was repeated with a varying number of requests and emulated clients.

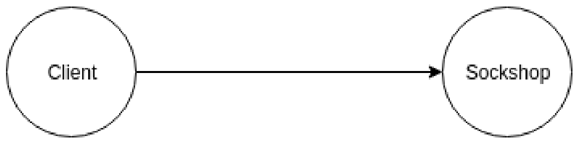

Figure 5.2: Stand-alone Sockshop

5.2.2 Prometheus scraping Sockshop

In this scenario we added Prometheus to collect metrics directly from the Sockshop's microservices (which served as Metrics Scrapers and scraped application metrics). The frequency of the pull requests by Prometheus was set to 1 and 5 seconds in different instances and the test script was run in both to measure the toll on QoS caused by directly collecting metrics from the application microservices.

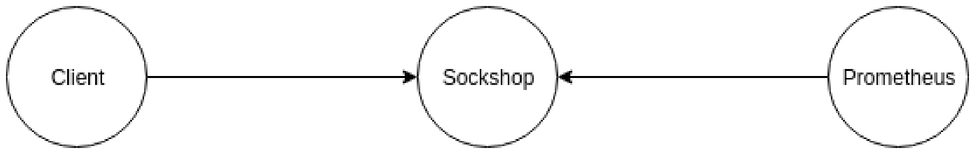

Figure 5.3: Prometheus scraping Sockshop

5.2.3 Prometheus scraping NodeExporter scraping Sockshop

Then, we deployed a scenario that represents a simple setup for a hybrid monitoring system. This time, a NodeExporter is used to scrape an array of machine-level metrics (default setting of NodeExporter[5]) from the node where Sockshop is being hosted. Similarly to the previous scenario, Prometheus scraped this NodeExporter with pull frequencies of 1 and 5 seconds.

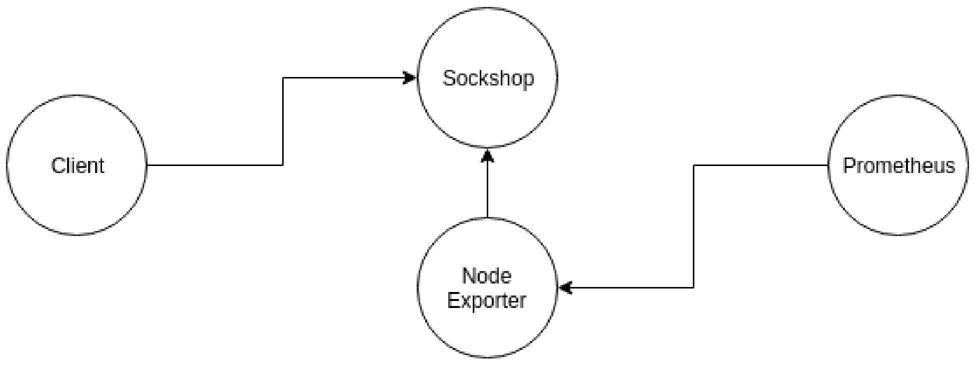

Figure 5.4: Prometheus scraping NodeExporter scraping Sockshop

5.2.4 Prometheus scraping EdgeMon scraping Sockshop

Finally, we created a scenario that is basically a stripped down version of the proposed architecture in which an EdgeMon is used to collect the same metrics as the previous scenario (EdgeMon possesses the same scraping capabilities as NodeExporter because it

[5]https://github.com/prometheus/node_exporter#enabled-by-default

was made using NodeExporter as one of its building blocks) plus metrics from some application components which are subsequently sent to Prometheus via its pull mechanism. This scenario does not possess the remaining components from the proposed architecture because they are considered secondary components that do not require significant resources. As such, they are not important material for the test scenario.

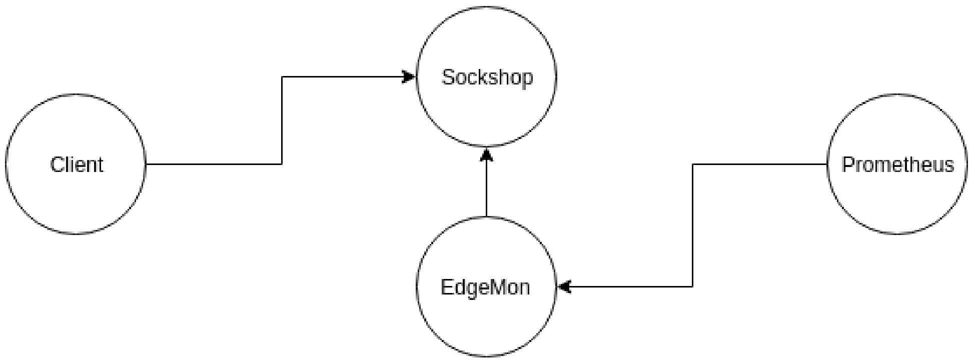

Figure 5.5: Prometheus scraping EdgeMon scraping Sockshop

5.2.5 Infrastructure

The monitoring system, along with the application being monitored were deployed inside a node of the DI/FCT/UNL's own cluster. Nodes in this cluster are connected via 1GB ethernet cables and are shared among students and teachers with access rights. Specifically, the node used to deploy our system was a 24-core node and had the following specifications:

- 2 Intel Xeon E5-2620 v2 (with Hyper-Threading)
- 64 GB of RAM
- 2 NIC Intel Corporation I210 Gigabit Network Connection

The test scenarios were not executed in isolation as other students were also using the cluster for their thesis.

5.3 Results Analysis

Firstly, we address the memory usage of all components involved in creating the proposed architecture so that we can see that their utilization costs are not excessive compared to the functionalities the system provides. Also, we present the same costs for the application being monitored. The results are shown in figures 5.6 and 5.7, respectively, and provide data referent to values measured before and after the test has been run. As mentioned in subsection 5.1.2, the Load-test service is used and the memory consumption, along with the data transferred (Input/Output) by all of the components involved in the test environment is recorded before the Load-test service starts emulating user requests and right after finishing.

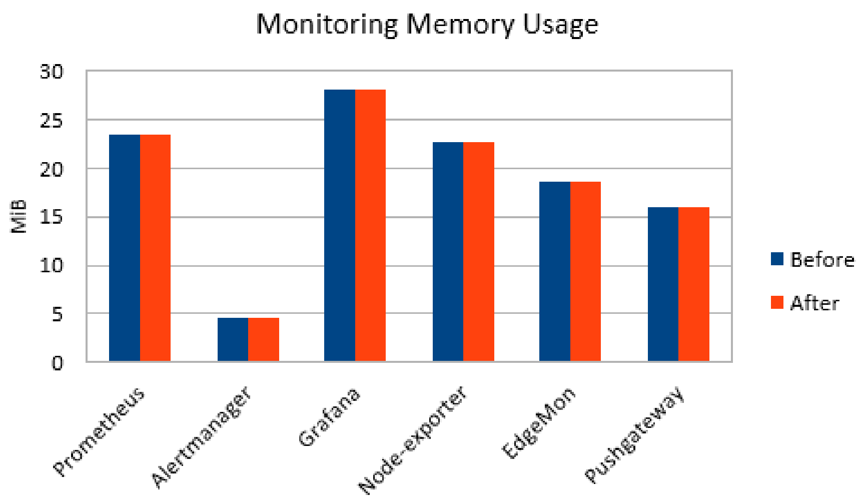

Figure 5.6: Memory used in megabytes(MB) by the monitoring system's components before and after testing.

As we can see all the monitoring components have very low and constant memory usage cost with an average of less than 20 megabytes(MB) which is not hard for most available computational nodes to provide.

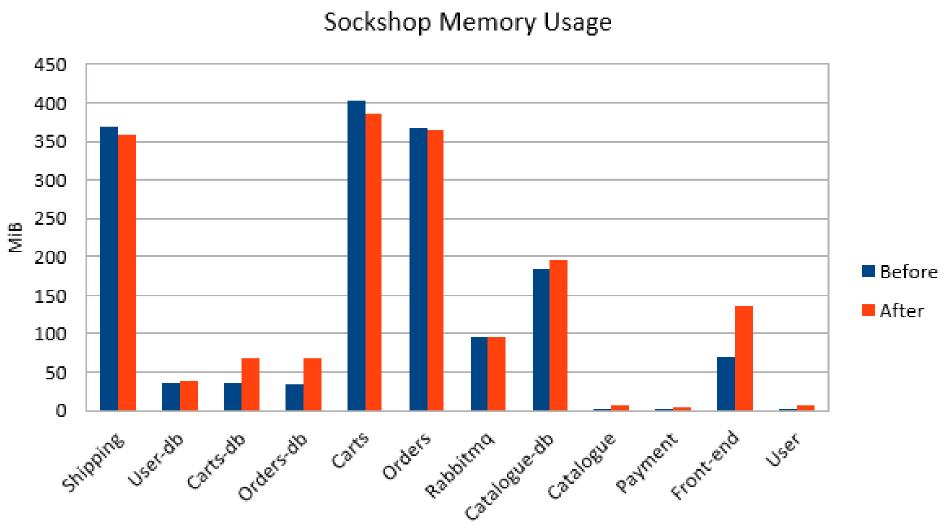

Figure 5.7: Memory used in megabytes(MB) by the Sockshop application before and after testing.

Comparatively, the application being monitored is fairly heavier with components reaching the 300 megabytes(MB) and having slight variations related to their operational requirements, but these factors are not relevant for our analysis as they are not a concern for the monitoring system.

5.3.1 Stand-alone Sockshop

In this category we tested how the application reacted against a varying number of clients and requests. This scenario is used as a basis for the rest and as such does not have monitoring services involved. Specifically, we did three tests: 16k requests with 1 client, 16k requests with 4 clients and 32k requests with 4 clients. These clients represent the number of concurrent calls that can be made to the frontend component and do not limit the number of real clients that can interact with the application. The calls mimic the normal interaction users have when accessing the application, which can be viewed in detail in listing 5.2. The metrics being evaluated are the data transferred in megabytes(MB) through the network by each component (Input/Output) and the duration of the test in seconds as shown by figures 5.8, 5.9 and 5.10.

Listing 5.2: Load-test service output for the test with 4 clients and 32k requests

```
Locust file : /config/ locustfile .py
Will run /config/ locustfile .py against node13. Spawning 4 clients and 32768 total requests.
[2018-11-14 02:44:40,722] node11/INFO/locust.main: Starting Locust 0.7.5
[2018-11-14 02:44:40,723] node11/INFO/locust.runners: Hatching and swarming 4 clients at the rate 5 clients/s...
[2018-11-14 02:44:41,527] node11/INFO/locust.runners: All locusts hatched: Web: 4
[2018-11-14 02:44:41,527] node11/INFO/locust.runners: Resetting stats

[2018-11-14 02:49:45,316] node11/INFO/locust.runners: All locusts dead

[2018-11-14 02:49:45,316] node11/INFO/locust.main: Shutting down (exit code 1), bye.
 Name                                                  # reqs     # fails     Avg    Min     Max |  Median  req/s
-----------------------------------------------------------------------------------------------------------------
 GET /                                                   3641    0(0.00%)       7      3      54 |       7  18.50
 GET /basket.html                                        3641    0(0.00%)       7      3      25 |       7  18.60
 DELETE /cart                                            3640    0(0.00%)      24      6    1024 |      12  18.60
 POST /cart                                              3639    2(0.05%)      40     12    2024 |      19  18.60
 GET /catalogue                                          3641    0(0.00%)      21      6    1019 |      11  18.50
 GET /category.html                                      3641    0(0.00%)       8      3      53 |       8  18.60
 GET /detail.html?id=03fef6ac-1896-4ce8-bd69-b798f85c6e0b  389    0(0.00%)       8      4      44 |       7   1.70
 GET /detail.html?id=3395a43e-2d88-40de-b95f-e00e1502085b 418    0(0.00%)       8      4      20 |       8   2.10
 GET /detail.html?id=510a0d7e-8e83-4193-b483-e27e09ddc34d 404    0(0.00%)       7      4      24 |       7   2.20
 GET /detail.html?id=808a2de1-1aaa-4c25-a9b9-6612e8f29a38  427    0(0.00%)       8      4      22 |       8   3.10
 GET /detail.html?id=819e1fbf-8b7e-4f6d-811f-693534916a8b  396    0(0.00%)       8      3      20 |       8   2.20
 GET /detail.html?id=837ab141-399e-4c1f-9abc-bace40296bac  416    0(0.00%)       8      4      20 |       8   1.80
 GET /detail.html?id=a0a4f044-b040-410d-8ead-4de0446aec7e  403    0(0.00%)       7      3      45 |       7   1.20
 GET /detail.html?id=d3588630-ad8e-49df-bbd7-3167f7efb246 376    0(0.00%)       7      4      21 |       7   1.70
 GET /detail.html?id=zzz4f044-b040-410d-8ead-4de0446aec7e  412    0(0.00%)       8      4      24 |       7   2.60
 GET /login                                              3641    0(0.00%)      89     47    2062 |      55  18.60
 POST /orders                                            3157  484(13.29%)    119     59    3064 |      70  15.00
-----------------------------------------------------------------------------------------------------------------
 Total                                                  32282  486(1.51%)                                  163.60

Percentage of the requests completed within given times
 Name                                                  # reqs    50%    66%    75%    80%    90%    95%    98%    99%   100%
-----------------------------------------------------------------------------------------------------------------
 GET /                                                   3641      7      8      9     10     12     13     16     18     54
 GET /basket.html                                        3641      7      8      9     10     12     13     16     17     25
 DELETE /cart                                            3640     12     13     14     15     17     20     26   1000   1024
 POST /cart                                              3639     19     21     22     23     26     31     53   1000   2024
 GET /catalogue                                          3641     11     13     14     15     17     21     47     55   1019
 GET /category.html                                      3641      8      9     10     11     12     14     17     19     53
 GET /detail.html?id=03fef6ac-1896-4ce8-bd69-b798f85c6e0b  389      7      9     10     10     12     14     14     17     44
 GET /detail.html?id=3395a43e-2d88-40de-b95f-e00e1502085b  418     8     9    10    10    12    13    14    15    20
 GET /detail.html?id=510a0d7e-8e83-4193-b483-e27e09ddc34d  404     7     8     9    10    11    13    15    18    24
 GET /detail.html?id=808a2de1-1aaa-4c25-a9b9-6612e8f29a38  427     8     9    10    11    12    14    16    17    22
 GET /detail.html?id=819e1fbf-8b7e-4f6d-811f-693534916a8b  396     8     9    10    10    12    14    16    17    20
 GET /detail.html?id=837ab141-399e-4c1f-9abc-bace40296bac  416     8     9    10    10    12    13    15    16    20
 GET /detail.html?id=a0a4f044-b040-410d-8ead-4de0446aec7e  403     7     8     9    10    12    13    15    16    45
 GET /detail.html?id=d3588630-ad8e-49df-bbd7-3167f7efb246  376     7     9    10    10    12    13    16    17    21
 GET /detail.html?id=zzz4f044-b040-410d-8ead-4de0446aec7e  412     7     9    10    10    12    14    17    19    24
 GET /login                                               3641    55    57    59    59    62    68  1100  1100  2062
 POST /orders                                             3157    70    73    75    76    81    90  1100  2100  3064
--------------------------------------------------------------------------------------------------------------------

Error report
 # occurences       Error
--------------------------------------------------------------------------------------------------------------------
 484                POST /orders: "HTTPError(u'406␣Client␣Error:␣Not␣Acceptable␣for␣url:␣http://node13/orders',)"
 2                  POST /cart: "HTTPError(u'500␣Server␣Error:␣Internal␣Server␣Error␣for␣url:␣http://node13/cart',)"
--------------------------------------------------------------------------------------------------------------------

done
```

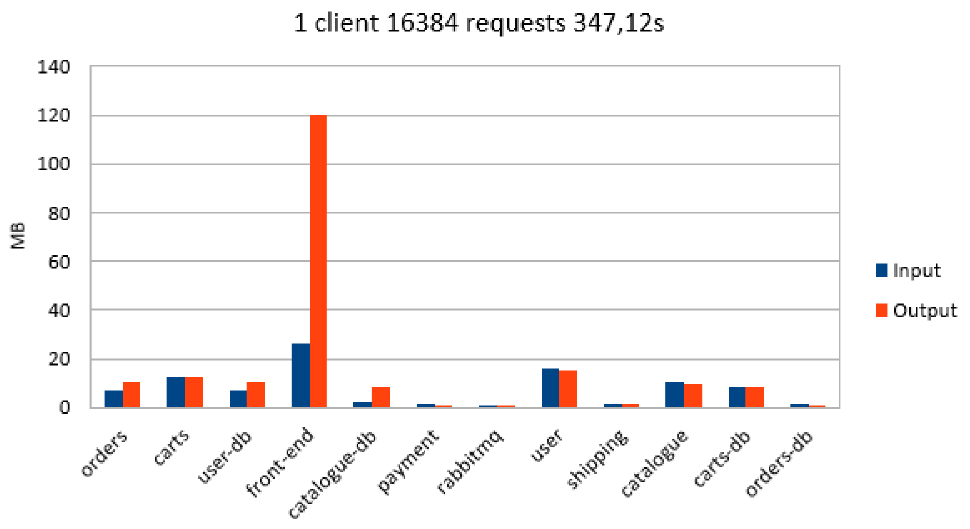

Figure 5.8: Data transferred in megabytes(MB) by the Shockshop application using 1 client to do 16k requests (duration: 347,12 seconds).

As expected the first test took the longest as it had only one client doing requests. In contrast, with the addition of 3 more clients the time was reduced more than threefold. When doubled the number of requests with the same 4 clients, the duration also doubled which was expected. With these tests we can also verify that the front-end microservice is the one that communicates with the outside and sends 120 MiB of data to respond to 16k requests (240 to 32k).

5.3.2 Prometheus scraping Sockshop

With this set of tests, we wanted to check if directly scraping application metrics from the majority of the application components with Prometheus had any impact on performance

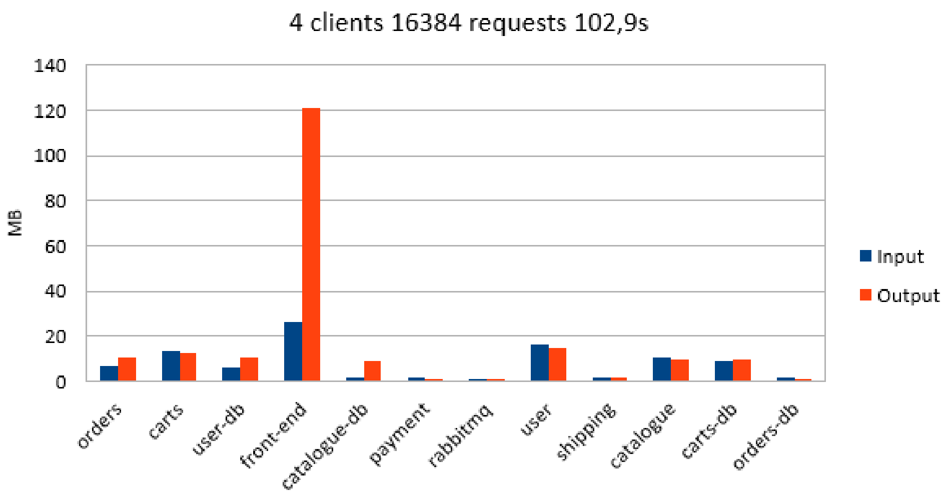

Figure 5.9: Data transferred in megabytes(MB) by the Shockshop application using 4 clients to do 16k requests (duration: 102,9 seconds).

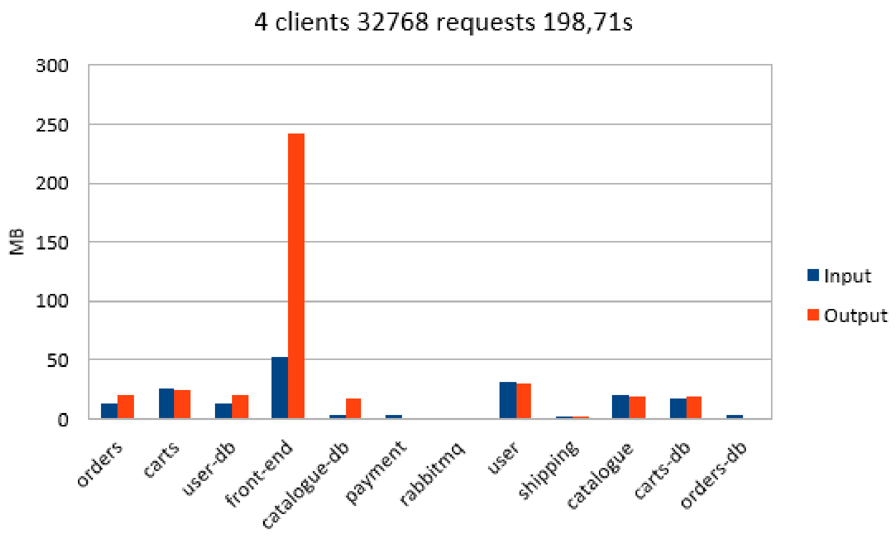

Figure 5.10: Data transferred in megabytes(MB) by the Shockshop application using 4 clients to do 32k requests (duration: 198,71 seconds).

and what was the load sustained by the network. The results are presented in figures 5.11 and 5.12.

We tested using a 5 second and a 1 second scrape interval and the results show that while taking relatively the same time to complete the test, Prometheus input load increased significantly (from 3 MB to 14 MB). If we were to scale this test to days or weeks of time, which is a normal time frame for a monitoring system to operate, then, the data transferred through the network would grow considerably.

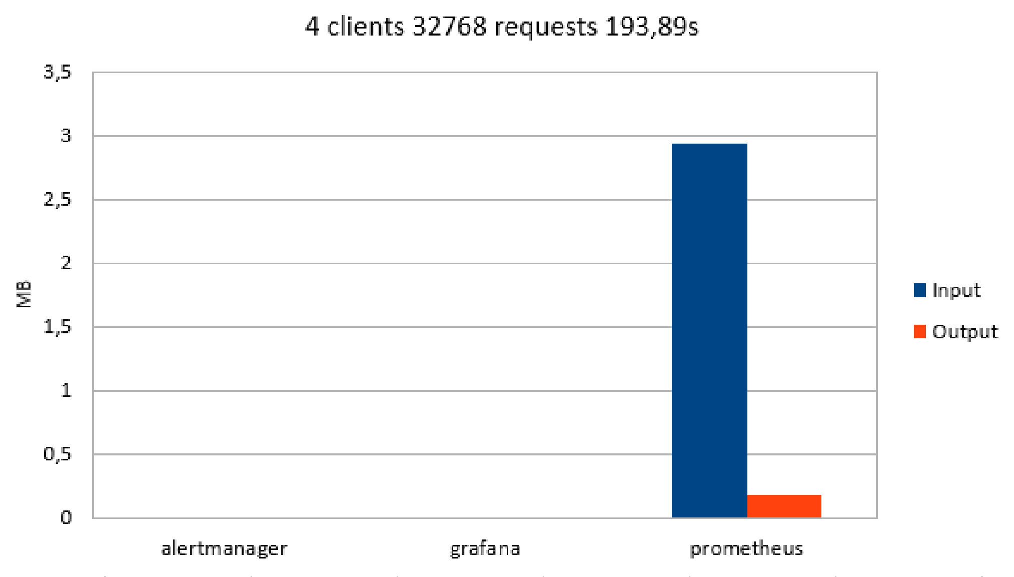

Figure 5.11: Data transferred in megabytes(MB) by the Shockshop application and the Prometheus stack (with pull frequency of 5 seconds) using 4 clients to do 32k requests (duration: 193,89 seconds).

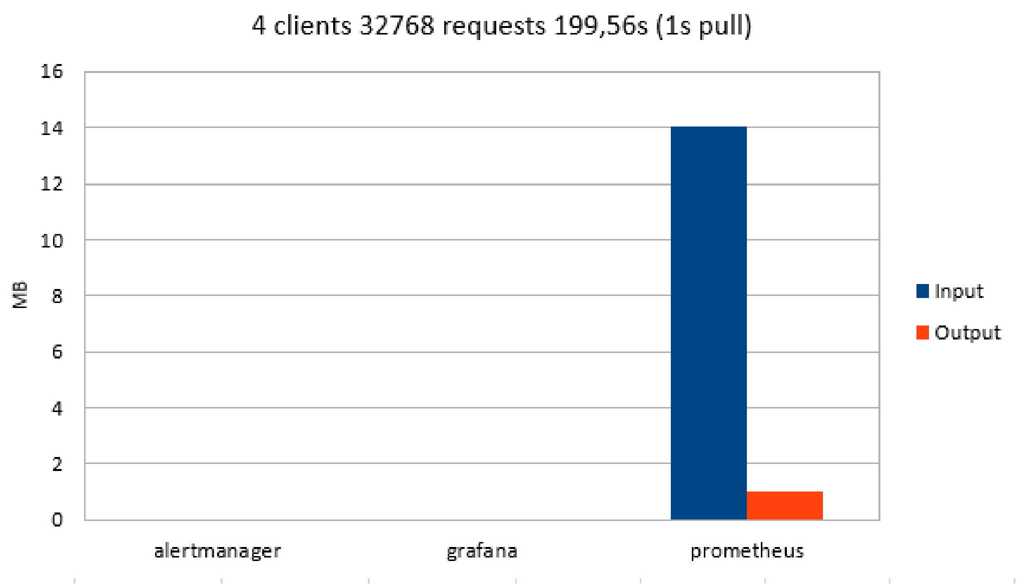

Figure 5.12: Data transferred in megabytes(MB) by the Shockshop application and the Prometheus stack (with pull frequency of 1 second) using 4 clients to do 32k requests (duration: 199,56 seconds).

5.3.3 Prometheus scraping a Metric Scraper scraping Sockshop

In this subsection we will make a parallel between using NodeExporter, which is the common practice when employing monitoring systems using Prometheus, and our new component EdgeMon on a stripped down version of the proposed architecture. This effort has the objective of finding out what changes EdgeMon has on the infrastructure requirements and if the advantages and functionalities it provides are relevant enough compared to the increased necessary resources. Figures 5.13 through 5.16 show the data transfered (Input/Output) by the Sockshop application and monitoring component where in one case a NodeExporter is used as Metrics Scraper and in the other EdgeMon is used. Prometheus scrapes them with a pull frequency of 1 and 5 seconds.

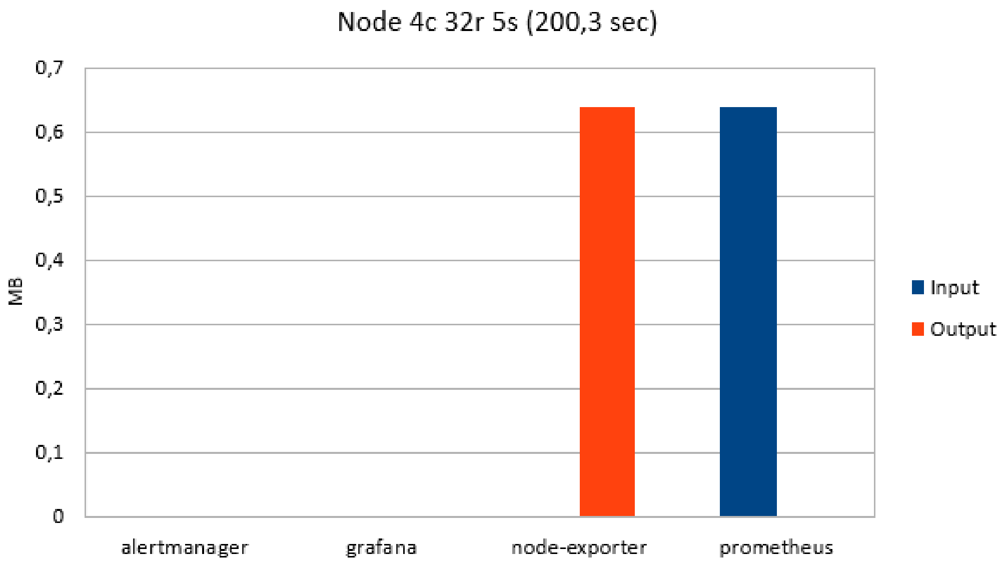

Figure 5.13: Data transferred in megabytes(MB) by the Shockshop application and the monitoring stack using NodeExporter (with pull frequency of 5 second) and 4 clients to do 32k requests (duration: 200,3 seconds).

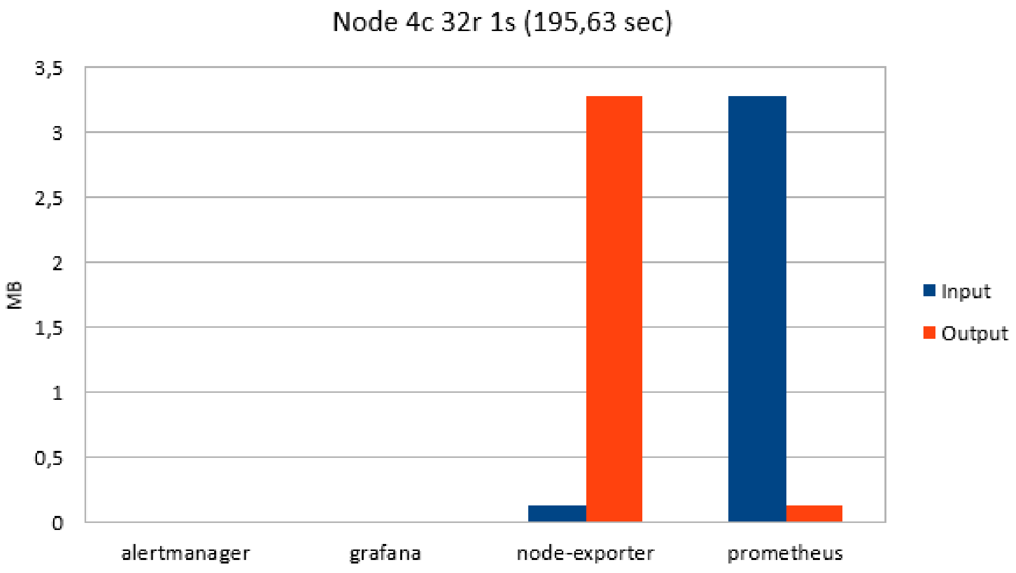

Figure 5.14: Data transferred in megabytes(MB) by the Shockshop application and the monitoring stack using NodeExporter (with pull frequency of 1 second) and 4 clients to do 32k requests (duration: 195,63 seconds).

As the data shows, both cases are very identical resource-wise but with EdgeMon,

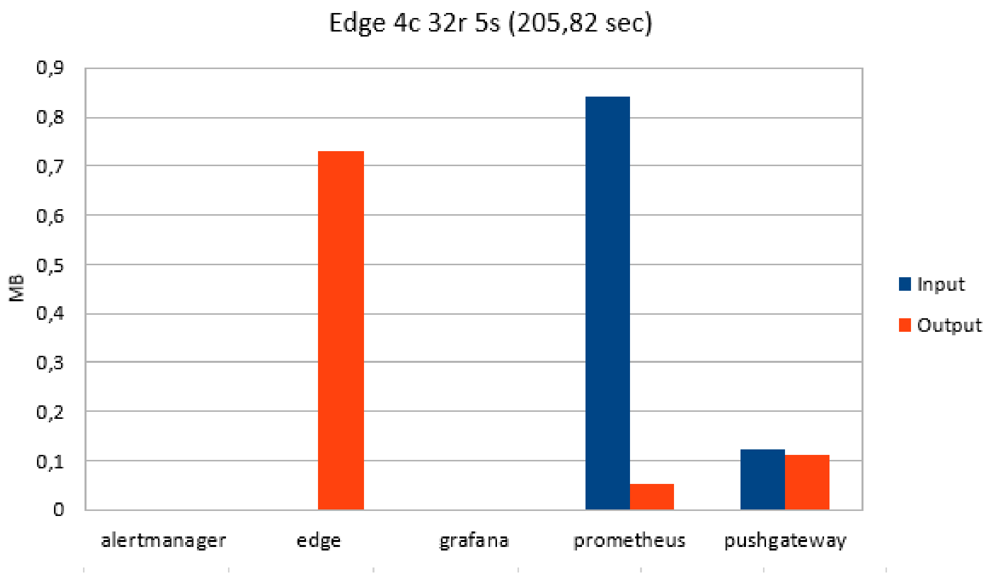

Figure 5.15: Data transferred in megabytes(MB) by the Shockshop application and the monitoring stack using EdgeMon (with pull frequency of 5 second) and 4 clients to do 32k requests (duration: 205,82 seconds).

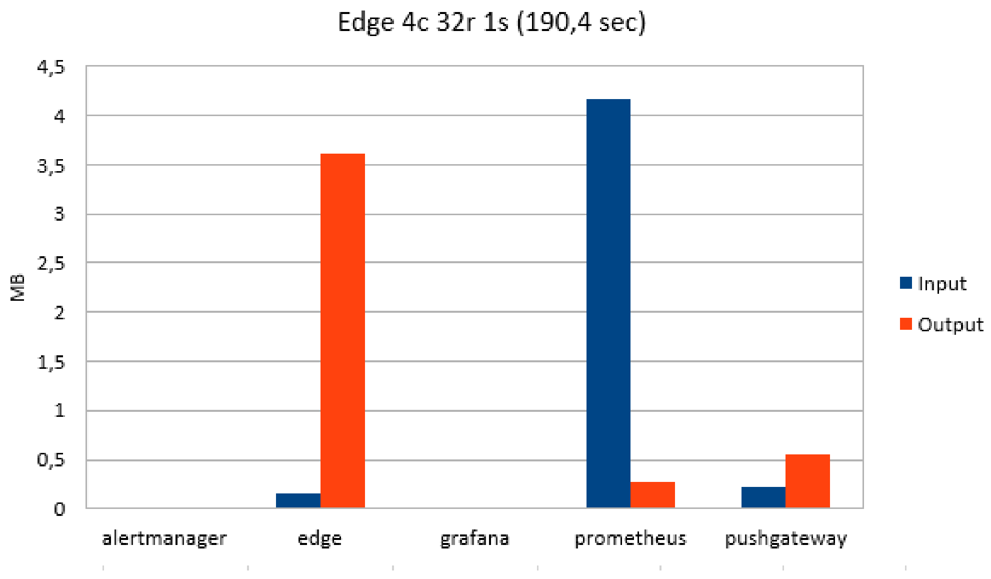

Figure 5.16: Data transferred in megabytes(MB) by the Shockshop application and the monitoring stack using EdgeMon (with pull frequency of 1 second) and 4 clients to do 32k requests (duration: 190,4 seconds).

the system is capable of dealing with the conditions of an Edge environment more favorably. In addition to scraping low-level metrics (the same scraped by Node Exporter[6]), using EdgeMon as the Metrics Scraper enables the system to provide the functionalities explained in subsection 4.2.

[6]https://github.com/prometheus/node_exporter#enabled-by-default

These functionalities increase the quality and effectiveness of monitoring systems located in hybrid environments as they provides capabilities that were lacking when using conventional monitoring methods. The Prometheus approach to monitoring in these situations (deploying Node Exporters or other simple Metrics Scrapers in each node) starts to fail when the system scales significantly, resulting in the congestion of the communication network. Using EdgeMon bypasses this problem as it employs specific communication mechanisms with the objective of minimizing the data transferred through the network.

CHAPTER 6

CONCLUSION

In this dissertation we developed an architecture for monitoring systems residing in hybrid environments (Edge/Cloud). This architecture has the main focus of mitigating the network congestion existing in conventional monitoring systems where all metric scrapers, residing in all infrastructure layers, communicate directly with the Cloud to send metrics for storage and analysis. This method, in addition to overloading the network, also stresses the Cloud infrastructure in terms of storage and processing requirements. Our architecture employs strategies to reduce these problems by taking additional advantage of the resources present in Edge nodes. By using these nodes to temporarily store metrics, evaluate them and relay alerts to the Cloud layer while reducing the communication frequency, it is possible to reduce the network's traffic, decrease the resource requirements of the Cloud layer and still preserve important properties like metrics' freshness and real-time alerting.

To prove the feasibility of these strategies, we implemented a monitoring system based on the proposed architecture presented in chapter 3, developed a new component to provide the required functionalities and created a testing environment, featuring different relevant scenarios. These scenarios tested the network load and memory usage of the system when monitoring an application hosted in the Edge layer. The results show that our system, composed of lightweight components in the Edge and average components in the Cloud, can collect relevant metrics and provide real-time alerting with minimal Edge requirements and using only a fraction of the requests' frequency when compared to conventional solutions reducing significantly potential network congestion. This is possible by utilizing our newly-created EdgeMon component, that has the capabilities of a lightweight Prometheus. It can store metrics temporarily, in the Edge layer, coming from various sources such as application components, regular Metrics Scrapers and components residing in resource-deprived nodes via its push mechanism. The evaluation of these metrics is also accomplished without requiring the intervention of the Cloud layer, as EdgeMon is featured with two Rules Engines (explained in subsection 4.2.4) capable of checking periodically and reactively the metrics stored. These rules, upon activation, trigger EdgeMon's alerting mechanism which forwards a message to the Cloud, informing the upper layer of the occurrence so that it can act accordingly.

6.1 Limitations and Future Work

The proposed architecture is still the first step towards a dynamic monitoring system for applications (e.g. based on micro-services) deployed on hybrid Cloud/Edge execution environments and so it still requires some improvements. One limitation is the fact that it cannot dynamically adjust (without human intervention) the configuration of its Metrics Gatherers, being it moving, adding or removing, in order to follow the migration of its Metrics Scrapers. Likewise, if the components of the application being monitored are to change their location, the Metrics Scrappers will not be able to dynamically migrate, upscale or downscale themselves in order to follow those components and to continue scraping their metrics. A possible improvement of this architecture can be the incorporation of a sophisticated service discovery component tasked with providing information, to the relevant components, about the dynamic behavior of application components and Metrics Scrapers.

The need to predefine rules within the Rules Engines is also a limitation. These fixed rules are not able to conform to the changing requirements that a monitoring system residing in such a hybrid dynamic environment will tipically have to respond to. The rules could be dynamically changed based on the analysis of other collected metrics. One way to accomplish a first version of this functionality is to store the rules in a configuration file and to provide a REST API to change them. This API can be exposed by the Rules Engines for invocation. Additionally, most components residing in the Edge nodes expose REST APIs that are used to communicate with the other monitoring components and with the Cloud layer. These APIs could also be used to provide additional functionalities by allowing end-users to interact with them. For instance, they could allow insight on the metrics present in a specific Metrics Cache (providing fresher metrics) or they could be used to examine the set of rules of a specific Rule Engine and its activation statistics.

Bibliography

[1] G. Aceto, A. Botta, W. De Donato, and A. Pescapè. “Cloud monitoring: A survey.” In: *Computer Networks* 57.9 (2013), pp. 2093–2115.

[2] M. T. Beck and M. Maier. “Mobile edge computing: Challenges for future virtual network embedding algorithms.” In: *Proc. The Eighth International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP 2014)*. Vol. 1. 2. Citeseer. 2014, p. 3.

[3] M. T. Beck, M. Werner, S. Feld, and S Schimper. “Mobile edge computing: A taxonomy.” In: *Proc. of the Sixth International Conference on Advances in Future Internet*. Citeseer. 2014, pp. 48–55.

[4] T. Bui. “Analysis of docker security.” In: *arXiv preprint arXiv:1501.02967* (2015).

[5] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. “The next step in web services.” In: *Communications of the ACM* 46.10 (2003), pp. 29–34.

[6] R. Dua, A. R. Raja, and D. Kakadia. “Virtualization vs containerization to support paas.” In: *2014 IEEE International Conference on Cloud Engineering*. IEEE. 2014, pp. 610–614.

[7] S. Haselböck and R. Weinreich. “Decision guidance models for microservice monitoring.” In: *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. IEEE. 2017, pp. 54–61.

[8] S. Kartakis and J. A. McCann. “Real-time Edge Analytics for Cyber Physical Systems using Compression Rates.” In: *ICAC*. Vol. 14. 2014, pp. 153–159.

[9] R. Mahmud, R. Kotagiri, and R. Buyya. “Fog computing: A taxonomy, survey and future directions.” In: *Internet of Everything*. Springer, 2018, pp. 103–130.

[10] N. Manohar. “A survey of virtualization techniques in cloud computing.” In: *Proceedings of international conference on vlsi, communication, advanced devices, signals & systems and networking (vcasan-2013)*. Springer. 2013, pp. 461–470.

[11] D. C. Marinescu. *Cloud Computing: Theory and Practice*. Newnes, 2013.

[12] P. Mell, T. Grance, et al. “The NIST definition of cloud computing.” In: (2011).

[13] S. Newman. *Building microservices: designing fine-grained systems*. "O’Reilly Media, Inc.", 2015.

[14] M. P. Papazoglou. "Service-oriented computing: Concepts, characteristics and directions." In: *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*. IEEE. 2003, pp. 3–12.

[15] F. Pina, J. Correia, R. Filipe, F. Araujo, and J. Cardroom. "Nonintrusive Monitoring of Microservice-Based Systems." In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2018, pp. 1–8.

[16] J. Povedano-Molina, J. M. Lopez-Vega, J. M. Lopez-Soler, A. Corradi, and L. Foschini. "DARGOS: A highly adaptable and scalable monitoring architecture for multi-tenant Clouds." In: *Future Generation Computer Systems* 29.8 (2013), pp. 2041–2056.

[17] A. Souza, N. Cacho, A. Noor, P. P. Jayaraman, A. Romanovsky, and R. Ranjan. "Osmotic monitoring of microservices between the edge and cloud." In: *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2018, pp. 758–765.

[18] H. J. Syed, A. Gani, R. W. Ahmad, M. K. Khan, and A. I. A. Ahmed. "Cloud monitoring: A review, taxonomy, and open research issues." In: *Journal of Network and Computer Applications* 98 (2017), pp. 11–26.

[19] K. Toczé and S. Nadjm-Tehrani. "Where Resources meet at the Edge." In: *Computer and Information Technology (CIT), 2017 IEEE International Conference on*. IEEE. 2017, pp. 302–307.

[20] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. "Challenges and opportunities in edge computing." In: *Smart Cloud (SmartCloud), IEEE International Conference on*. IEEE. 2016, pp. 20–26.

[21] M. Villari. *Achieving Federated and Self-Manageable Cloud Infrastructures: Theory and Practice: Theory and Practice*. IGI Global, 2012.

[22] L. Xu, Z. Wang, and W. Chen. "The study and evaluation of ARM-based mobile virtualization." In: *International Journal of Distributed Sensor Networks* 11.7 (2015), p. 310308.

[23] J. Yang. "Web service componentization." In: *Communications of the ACM* 46.10 (2003), pp. 35–40.

[24] D. Yin and T. Kosar. "A data-aware workflow scheduling algorithm for heterogeneous distributed systems." In: *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE. 2011, pp. 114–120.

[25] X. Zhao, J. Yin, C. Zhi, and Z. Chen. "SimMon: a toolkit for simulation of monitoring mechanisms in cloud computing environment." In: *Concurrency and Computation: Practice and Experience* 29.1 (2017).