Bruno André Pitta Grós do Valle e Anjos

Bachelor in Computer Science and Engineering

LowNimbus: A decentralized autonomic cloud to edge deployment framework

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

Master of Science in
Computer Science and Engineering

Adviser: João Carlos Antunes Leitão,
Assistant Professor, NOVA University Lisbon

Examination Committee:

Chair: Bernardo Parente Coutinho Fernandes Toninho,
Assistant Professor, Nova University Lisbon

Rapporteur: João Nuno de Oliveira e Silva,
Assistant Professor, Instituto Superior Técnico

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2021

LowNimbus: A decentralized autonomic cloud to edge deployment framework

Copyright © Bruno André Pitta Grós do Valle e Anjos, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

To my friends and family.

Acknowledgements

The work present throughout this thesis would not have been possible without the help and guidance of several people for which I am deeply grateful. I would like to thank my advisor João Leitão for his mentorship, dedication and friendship which guided me through the path of hard work that made this thesis possible. Without such outstanding advisor, this work would not have been possible and I would not have the commitment, motivation and hard work mentality that was developed throughout this process.

I also want to thank Pedro Ákos and Pedro Fouto, without whom this work would not have been possible, for challenging and guiding me through hard times and ultimately making me a better engineer and a better person. My thanks also extend to the Department of Informatics of the NOVA School of Science and Technology and the NOVA LINCS research center, which provided me with the resources needed to conduct this work. Furthermore, I would like to thank all my colleagues and friends that helped me push through hard times by creating an environment that encouraged laughter and dedication.

Lastly, none of this would be possible without my sister Bárbara, my grandmother Idalina and my mother Eliana. I will always cherish your guidance, compassion and dedication on making me a better person and supporting me in the pursuit of my goals. To all my family that was not named here I will always be grateful to you.

The work presented in this thesis was partially supported by FCT through NOVA LINCS (grant UIDB/04516/2020) and NG-STORAGE (PTDC/CCIINF/32038/2017). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Abstract

The cloud has established itself as the preferred deployment platform for most distributed services, providing its users with low maintenance requirements, and nearly infinite scalability with an attractive cost model. Nevertheless, with the increasing scale of many mobile and Internet of Things applications, which translates to an increase on the number of devices at the edge of the network, the network links connecting such devices to the cloud will inevitably saturate, and applications will experience a penalty on latency due to the cloud highly centralized architecture.

In order to tackle these challenges, paradigms such as Edge Computing have emerged. Edge computing focuses on developing solutions that can make the most out of the resources near the edge or even further along the path between end devices and cloud infrastructures. A solution that successfully explores these resources will be able to provide considerably lower latencies, while offloading some computations of applications to edge computational resources.

In this work, we propose a prototype solution that enables the efficient deployment of application components to the edge, while taking into consideration some of the challenges that arise from using edge nodes in a decentralized manner, such as higher failure probability and efficient communication. Furthermore, we also propose an application benchmark that is inspired by real life applications and that can show the advantages and drawbacks of deployment systems that make use of edge nodes.

Throughout the document, not only will we analyse deployment solutions, but also other works that have been applied in an edge environment, providing insights on the limitations and advantages of such environment.

Lastly, the experimental results obtained will be the result of from running the before mentioned benchmark on the proposed prototype, running in different configurations on the Grid'5000 cluster.

Keywords: Edge Computing, distributed, deployment, decentralised control

Resumo

A *cloud* tem-se estabelecido como a plataforma de eleição para suportar a execução de serviços distribuídos, e providenciar recursos virtualmente infinitos que escalam com facilidade a um custo acessível, mantendo baixa a necessidade de manutenção por parte do utilizador. O aumento da escala em muitas aplicações como *mobile* ou *IoT*, traduz-se num aumento do número de dispositivos presentes na extremidade da rede. Este fator leva a que as conexões entre os dispositivos na extremidade da rede e a nuvem (*cloud*) inevitavelmente fiquem saturadas, penalizando assim as aplicações com latências elevadas, em grande parte devido ao modelo altamente centralizado da *cloud*.

Recentemente surgiram paradigmas de computação como a computação na periferia (Edge Computing), que se focam em desenvolver soluções que consigam tirar partido dos recursos na extremidade da rede, ou mesmo dos que se encontram no caminho entre os dispositivos na extremidade e as infraestruturas da nuvem. Soluções que consigam tirar proveito com sucesso dos mesmos, conseguem obter latências consideravelmente mais baixas e executar algumas das computações da aplicação nos nós presentes na extremidade.

Neste documento, propomos uma solução que consiga gerir eficientemente a instalação e execução dos componentes das aplicações, tendo em conta os desafios que surgem em usar os dispositivos na extremidade da rede num sistema distribuído, tais como maior probabilidade de falha e comunicação com qualidade variável. Adicionalmente, propomos ainda uma aplicação de *benchmarking* inspirada em aplicações reais que permita averiguar as vantagens e desvantagens do uso de nós da periferia em sistemas de operacionalização de aplicações.

Ao longo do documento analisaremos outras soluções de instalação e execução de trabalhos que tenham sido aplicados na extremidade da rede, providenciando assim uma visão preliminar sobre as vantagens e desvantagens de tal ambiente.

Em último, os resultados experimentais aqui apresentados serão o resultado da execução do *benchmark* mencionado previamente no protótipo desenvolvido em diferentes configurações de nós na plataforma Grid'5000.

Palavras-chave: Computação na periferia, instalação, execução, distribuído, controlo descentralizado

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

Up until recently, in order to deploy applications that required large amounts of computing and/or storage resources, the primary solution was the cloud. The cloud is a materialization of the Cloud Computing model, where the fundamental idea is to have a centralized cluster of computing and storing nodes, that can be seen to the clients as a virtual infinite amount of resources available and that can scale vertically (allocate more resources for a given machine) and horizontally (rent more machines). Moreover, the attractive price models, and the plethora of deployment models and support services/abstractions that it offers to its users resulted in a lot of applications switching to the cloud for reliable deployment.

As IoT and mobile applications started to become more common, naturally these applications also sought to take advantage of the resources and abstractions provided by the cloud as to fulfill their needs to process the huge amounts of data produced by the end devices. However, soon these applications started to saturate the links to the cloud due to the amount of data that had to flow into and from the cloud, leading to a high penalty on latency. This lead to a need for a platform that can minimize the latency from the client to the application, while being able to handle an ever-increasing number of users.

1.2 Motivation

Even though cloud computing can fulfill the computational and storage needs of most applications, centralizing these resources presents some limitations. Although it can be easier to manage computational resources and application components in the cloud infrastructure, clients that are far from this central point will potentially experience high latency. Cloud computing uses replication techniques to reduce the impact on such clients, by using multiple datacenters around the world, thus minimizing the distance from an end user to the closest datacenter.

However, in a geo-replicated scenario, each replica still suffers from being a central point to its region, so as mentioned before, in an era where IoT is becoming increasingly relevant, the large volumes of data being generated by IoT devices can still saturate the bandwidth of links between devices and the cloud. Another type of application that suffers from a cloud deployment are human centered applications [17] (such as mobile interactive games similar to Pokemon Go), where the application has to wait for a message to go to the cloud and back just to interact with a user that can be located by its side.

In recent years a new computing paradigm has emerged that aims to push the services closer to the user. This paradigm is called Edge Computing, and the main idea is to intercept traffic going to the cloud and execute tasks or store data locally and possibly never having to go to the cloud. While the edge computing paradigm is a recently new concept, its core idea has been previously employed (in a limited fashion) in solutions such as CDNs [30], that deploy caching servers near the clients, so that if two users in the same region request the same content, the server deployed near them only requests it once to the centralized infrastructure (i.e. the cloud), thus reducing latency to the clients and bandwidth usage on the lines providing access to the cloud.

Edge computing is not a panacea, since it faces its own challenges, such as security concerns, deployment complexities, resource management, and achieving data locality when access patterns are dynamic. Some of these challenges turn out to be hard to solve since edge computing has a more decentralized nature and tighter relation with clients than considering only the highly centralized cloud infrastructures. Another disadvantage is that the devices that comprise the edge network are more heterogeneous than in a cloud environment, since in the cloud the provider has the ability to standardize its resources (at least at the presentation layer, for instance taking advantage of virtualization).

Even though edge computing is hereby presented by comparing advantages and disadvantages with cloud computing, in fact these are not to be seen as orthogonal concepts because even though in most cases edge computing decreases the impact of the disadvantages of cloud computing, the same can be said the other way around, for example, whereas edge computational resources lack computing power, the cloud excels at it. In fact these concepts are best applied in a symbiotic manner.

1.3 Objective

In this thesis we will tackle the challenges inherent to the deployment of services at the edge, in a decentralized manner, while aiming to provide lower latencies for applications end users when compared to cloud deployments. Therefore, we will focus on service migration, scalability, fault tolerance and latency optimization.

Challenges related with data locality, such as moving the data that a service needs to operate on to its destination will be left as a feature research topic. Resource monitoring will also not be addressed directly in this work. This is due to these challenges being orthogonal to our solution and being researched in parallel works.

As a case study we will consider human centered mobile applications, since these are predicted to have bigger potential improvements. This is due to real time human interaction having a high need for low response time, where latency can be a decisive factor for the success (in user base but also economic) of such applications.

1.4 Main Contributions

The main contributions of this work are twofold. First we design and implement a benchmark called PouchBeasts that is modular and emulates the different interactions patterns present at the edge for an interactive mobile application. Furthermore, we also propose a novel deployment framework that will meet previously mentioned objectives. The contributions can be partitioned into the following:

- Design and implementation of a benchmark that can express complex interaction patterns, with a modular architecture, portability and ease of deployment as key features (through containerization).
- Propose a novel framework that decouples the infrastructure management from the service deployment.
- Implementation of a prototype of this framework, addressing fault tolerance, efficient communication, low latency, and scalability.
- Evaluation of the proposed solution with the previously mentioned benchmark comparing performance metrics, such as latency, with centralized cloud solutions.

1.5 Document Structure

The document is organized as follows:

Chapter 2 presents the related work, giving insights on cloud computing, edge computing, and some techniques that might be important for a deployment solution, such as serverless computing, code offloading, service migration as well as other deployment frameworks.

Chapter 3 presents our deployment framework and the multiple microservices by which it is comprised. It also gives insight on how the applications and end users interact with our framework, and lastly discusses several implementation choices.

Chapter 4 details the design choices of the PouchBeasts benchmark, followed by the implementation and a description of the network model employed.

Chapter 5 presents the framework evaluation of LowNimbus, our deployment framework, which focuses on two scenarios, cloud and cloud-edge. This chapter also presents the experimental results followed by a discussion on the results obtained.

Chapter 6 ends this document providing the main conclusions and pointers for future work.

Chapter 2

Related Work

In this chapter we will give an insight on distributed computing, discussing key strengths and weaknesses alongside some examples used in production systems nowadays. It further analyzes cloud computing as a starting point for edge-based solutions, mainly focused on techniques that are employed to extend cloud based systems towards the edge, what it offers to the users among other characteristics. The schedulers implemented in cloud-based solutions are analyzed in greater detail since these are key components for achieving an adequate distributed deployment solution. Moreover, scheduling techniques might translate, even though if not entirely, to edge settings.

We follow this by explaining the concept of edge computing, some of its properties, and survey the most relevant works that have been conducted in this area. This is followed by a discussion on how edge computing can address some disadvantages of cloud computing.

We will then present and discuss serverless computing, which is a potential technique that can be leveraged for supporting edge-cloud deployments.

Lastly, we will analyze works that tackle the multiple challenges of a distributed deployment solution with some emphasis in the context of edge scenarios, namely through code offloading, service migration and constraint solving.

2.1 Computing Paradigms

2.1.1 Cloud Computing

Cloud computing has been concisely defined by NIST as follows [39]:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, *servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

According to the National Institute of Standards and Technology there are five essential characteristics that characterize the cloud model. In summary, these state that (1) when a client wants to provision resources it should not need human intervention on the provider side; (2) provisioning resources should be possible from any kind of client platform (e.g. mobile, laptop, desktop); (3) computational and network resources should be collocated in a pool ready to be allocated by a client using a multi-tenant model; (4) resources allocated should scale according to the consumer demand; and finally, (5) the provider should be able to monitor the usage of all allocated resources.

For more detailed definitions the reader is referred to [39].

Fundamental Concepts

As mentioned by the NIST definition, a cloud infrastructure should implement a multi-tenant model, this means that even though the resource pool is a single conglomerate, there should be a hard (logical) barrier between any two users of the cloud infrastructure. This implies that one user should never be allowed to have any kind of impact on another user's resources. In order to provide such isolation, there are two major techniques: virtual machines (VMs) and containers.

Virtual machines use a Virtual Machine Monitor (VMM) as the main component that manages all the virtualized systems (these are called *guests*). The VMM can control the existing hardware in order to offer small partitions of the whole system to different *guests*. The usage of hardware level virtualization by VMs is one of its strengths since it enforces a hard barrier between the different systems. Nevertheless, this kind of lower level virtualization translates into a bigger overhead from the VMM, since everything from the operating system upwards has to be run individually per *guest*, which ultimately means longer startup times. Cloud providers have employed some techniques to mitigate this problem such as keeping a warm pool of virtual machines, that are already initialized but still waiting to be assigned a tenant, thus cutting down the time necessary to spin up the machine from the moment the client requests it. Regardless the optimizations employed in the cloud, virtual machines still impose a heavy overhead, which eventually lead to research advances in more lightweight isolation mechanisms, that do not pose such high impact on startup.

One of these techniques that has been recently becoming increasingly popular is containerization. The use of containers aims to be a more lightweight virtualization mechanism that execute directly on top of a host operating system, hence minimizing overhead. Even though ideally containers aim to provide the same level of isolation as virtual machines, in all fairness, software defined virtualization is still more susceptible to allow possible unwanted interference from a container to the main system due to its architectural design [54].

The first container-based system was Linux containers (LXC) [38] that used *chroot*, *namespaces*, and *cgroups* to achieve OS level isolation. As mentioned in [7] the original purpose of these three kernel components was to protect applications from noisy, nosy, and messy applications executing within the same machine. The *chroot* UNIX operation changes the root directory of the current running process and its children to the specified one, thus denying access to files outside the specified directory tree. *Namespaces* are used to partition resources so that different processes in different namespaces see different sets of resources. Finally, *cgroups* (control groups), are used to limit resource usage (e.g. CPU, disk I/O, network) for a given collection of processes. *cgroups* can also measure, prioritize, or even freeze the usage of these resources. These three components were the basis of other containerization solutions besides, LXC, such as Docker [11]. Docker would eventually popularize containers in the industry, becoming one of the most popular container platforms [42] in part due to its ease of use and the existence of platforms like Docker Hub [12] that provide a vast library of container images that can be easily downloaded to a machine or used as the base image to build new containers on top of them. Regardless of its popularity, containerization is still not a mature technology when compared to VMs that have decades of development, testing, and optimization.

This lack of maturity, has not stopped major cloud providers such as Google, IBM/Softlayer, and Joyent to use containers as an alternative to VMs due to its better performance [5]. Google has been one of the main industry researchers on containerization technology throughout the years, with systems such as Borg [56], Omega [50] and Kubernetes [33]. Borg was the initial solution developed by Google to manage huge scale distributed resources, and to this day it remains the primary container-management system within Google due to its scale, breadth of features and extreme robustness. Omega started as an offspring of Borg, trying to improve the software engineering of Borg, but eventually many of the Omega innovations were merged into Borg (see Section 2.5 for a more detailed analysis on both Borg and Omega). Eventually interest from outside developers in containerization started to increased at the same time that Google was developing Kubernetes, leading Google to open-source the system contrary to the other two [7].

Solutions such as the ones previously mentioned can manage resources at a cloud infrastructure scale, but how these are offered to the clients still poses a different question. Cloud providers usually offer three main service models:

Infrastructure as a Service (IaaS) allows the user to provision different types of resources (e.g. processing, storage and networks) where it can deploy arbitrary software, including the operating system. The client also has the ability to control infrastructure-level configurations such as firewalls (although this feature may change according to the provider).

Platform as a Service (PaaS) provides a deployment platform where the user develops its own software (that is entwined with the platform). It also provides mechanisms to enable the deployment of the user application and then deploys it to the cloud.

Usually this is done through a middleware that supports multiple programming languages. One of the most known examples is Google App Engine.

Software as a Service (SaaS) is a software that can be accessed by the users usually via a web browser (or other thin client software), where the client has access to a number of services as features that are executing in the cloud, thus reducing the client-side hardware requirements. Gmail, Hotmail and Google Apps [46] are popular examples of SaaS.

Cloud computing service models have evolved to offer deployment solutions that reduce the need for infrastructure configuration as to accommodate smaller users that only want to deploy a small piece of code. This service model is called Function as a Service (FaaS) and is a materialization of a model named Serverless Computing that will be discussed in greater detail in Section 2.2 due to its recent applications in edge computing environments.

These different service models provide a very extensive interface for the users to interact with the cloud, addressing the needs of a casual user, up to an international company. The problem that stems from this high user coverage, is that every application is now deploying some component to the cloud, in part due to the attractive pricing models.

Among these applications, are ones that have a big portion of users at the edge of the networks such as IoT or mobile applications. Both IoT and mobile devices are less powerful, which ultimately implies that most of the processing and storage is done in the cloud, therefore a lot of data has to flow to and from these end devices, either for being produced or consumed there. All this data flowing to the cloud can ultimately saturate the links, since geographically the cloud is a central point. Moreover, the physical distance to the end users can incur in high latency, thus providing an overall unpleasant experience.

Discussion

Throughout this section we analyzed the cloud and its features and limitations. The limitations are mainly due to the centralized model that it follows, the growing number of clients which increases the amount of data produced and consumed and the distance between the servers and the clients.

In the next section we will see how Edge computing can address these challenges and possibly solve some of them.

2.1.2 Edge Computing

Edge computing is an emergent distributed computing model that aims at pushing the computation from the cloud data centers to the periphery of the network, leveraging on existing computational resources for performing computations closer to the data producer/consumer. The devices that make up this model are called *edge devices* (i.e. an edge computational resource) and these can be any device from ISP Servers up to IoT devices. The main idea to retain is that, every device in the path between the end device and the cloud can be seen as an *edge device*, since it can intercept data that is flowing between the extremes (more often from end devices to cloud data centers) and process it, thus removing the need to transport (all) data to the cloud. The set of devices that comprise this path up to the cloud is considerably heterogeneous, which lead to works structuring the edge as a multi layer model that segregates devices according to their proximity to the cloud or computing capacity [17, 37, 52].

Other models such as Fog Computing [32, 41, 29] and Osmotic Computing [57], enrich the cloud model by proposing to take advantage of resources along the path to the cloud to process the data. The main factors that lead to envision these models and edge computing itself are:

Proximity In the case when a user wants to send a message to another it is faster if it has to go through as few nodes up the path to the cloud as it can. Peet-to-Peer networks or mechanisms in the context of edge computing would make inter-user communication faster, while avoiding delays due to congestioned network links to the cloud infrastructures.

Intelligence Computing capacity in the edge is increasing, so autonomous decision-making can be implemented on edge devices, which can make a big difference by enabling faster decisions to be made without the need to wait for data to go to the cloud and decisions to be communicated back to the edge.

Trust Personal and sensitive data can be only located at the edge, since that's where the users are, avoiding its disclosure to third parties (i.e. cloud operators), and reestricting sensitive data to reside in devices under the administrative domain of the user to whom the data belongs.

Control The decisions of what to deploy and when to deploy are also coming from the edge, since most of the users requests originate from there.

IoT and mobile applications are case studies that contain most of the properties above described. Both IoT and mobile devices are close to the user (*proximity*), mobile devices are considerably powerful (*intelligence*), the data that the two types of devices handle (produce or consume) is usually personal (*trust*), and finally the access patterns from mobile users are related with application demand (*control*). Moreover, the Fog Computing model was designed having in mind IoT applications, further cementing the role of these applications in edge computing.

Fundamental Concepts

The term edge computing is quite general, and it includes previous architectural patterns that have been proposed in the past that focus on limited aspects of the edge. These include Fog Computing and Osmotic Computing that we discuss in the following paragraphs.

Fog Computing has been envisioned as a "*promising extension of Cloud computing paradigm to handle IoT related issues*"[41]. In fact, the main idea behind fog computing is to deploy nodes closer to the devices that are producing and acting on IoT data. These devices are called *fog nodes* and they can be materialized by switches, routers, or any other device that has a network connection, computing power and storage capacity.

The benefits of using these fog nodes is not their computing capacity *per se*, but instead the scale and location in which these are deployed. According to [41] the literature uses different keywords to describe these fog nodes, which include micro servers, micro datacenters, nanoservers, etc. Other works also characterize fog servers according to their functionalities such as cache servers, computation servers, storage servers, etc.

One of the main challenges that fog computing inherits from edge computing is the device heterogeneity, being that in fog computing this difference is even more noticeable due to the predominance of (small) IoT devices. These devices, in addition to having low computation capabilities, tend to have limited storage and power supply, which adds additional challenges.

Osmotic computing [57] is a Hybrid Cloud-Edge environment that unlike Fog Computing, takes a less edge-focused approach and tries to achieve a seamless osmosis of microservices that compose an application between the cloud and the edge infrastructure (thus the term osmotic computing). Figure 2.1 illustrates the ideology behind osmotic computing.

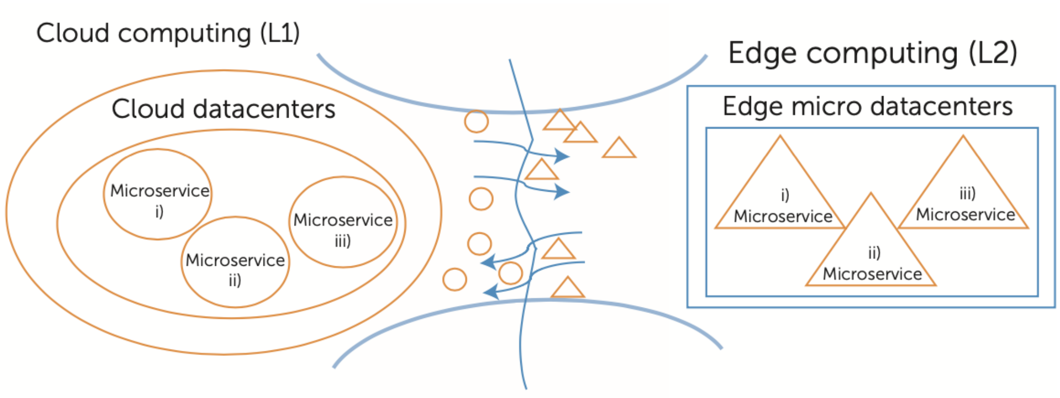


Figure 2.1: Osmotic Computing ideology [57].

In the osmotic computing model, the applications are decomposed into microservices to be able to change their deployment environment between the edge and the cloud. This decomposition allows a fine-grained control over the deployment and lifecycles of the application services, therefore enabling cloud-edge transition for a single service instead of the whole application. Nevertheless, challenges such as dynamic and efficient management of virtual networks, without the degradation of quality of service are still hard to tackle. Osmotic computing aims to address these, by decoupling the networking management from the application through container-based solutions, such as Docker and Kubernetes. These platforms allow deployments of microservices while being application agnostic. The networking management is done through the use of Software Defined Networks, providing abstraction of the underlying technologies.

However, the osmotic computing model is still a very recent research topic, and to our best knowledge no work has materialized service deployment, enabling cloud-edge migration, as proposed by the model.

Discussion

One of the main characteristics that makes cloud computing so popular is its (nearly) infinite resource capabilities that enable on demand elasticity. This is a great solution for the client that deploys its application and does not have to take care of any additional infrastructure configuration. However, since the cloud is (on a regional perspective) a centralized point, an end user might experience higher latencies when she is further from a datacenter. In a model where the cloud is the core, most of the traffic will be going inwards, whereas in the more advanced hybrid cloud-edge models, some traffic will be preprocessed and ideally prevented from reaching the cloud. As we have discussed previously, edge computing is not a panacea that can substitute the cloud completely, therefore combining these two solutions will translate into an overall more flexible and robust architecture.

Some challenges related with edge computing have already been mentioned, such as device heterogeneity and, in particular, the existence of some resource restricted devices. Other challenges such as system modeling and communication are reiterations of challenges present in distributed systems and sometimes more specifically peer-to-peer systems. This is due to the number of devices and the fact that some might exhibit mobility, where a centralized system would struggle to deal with the bandwidth needs and be less able to evolve dynamically with client movement.

The following are important challenges that have to be tackled for edge-focused solutions:

Resource Management This creates the bottom layer on top of which the remaining solutions will be built. This layer is responsible for two different subjects: network membership and resource cataloging. Firstly, there has to be mechanisms regarding network membership, potentially by building an overlay network that is capable of tracking resource membership while providing mechanisms to guarantee message delivery and withstand node failures. Works such as [58, 36] have developed interesting solutions for maintaining a membership upon which decentralized protocols that achieve high probability message delivery can be implemented.

Resource cataloging is responsible for keeping track of the resources available throughout the system and can possibly implement some logic to support the creation and decommission of an application components, thus managing the virtualisation software.

Application Agnostic Execution Environment The runtime environment should be agnostic enough so that it does not enforce a user to tailor their application to an edge deployed environment. Although specializing a given component for an edge environment can make the most out of the advantages of such environment, a relevant challenge to take advantage of edge computing is to be able to support a transparent and efficient redeployment of the applications already deployed in the cloud to the edge.

Dynamic Deployment It is also important to determine autonomously when to switch between the cloud and the edge environments. Such a component has to be able to analyze what conditions make an edge deployment beneficial to the overall execution of the application. The conditions can be application dependent or formalized in a model such that captures clear conditions for this to happen, for instance if the number of requests originating from the same point of presence goes above a threshold, the system should deploy some components of the application to a close by edge location.

The main focus of this thesis is to explore dynamic deployment and distributed scheduling, allowing decisions to be made at runtime regarding where to deploy different application components. To develop a suitable solution, it is important to understand the current runtime environment solutions on which to run application components. Some of the latest work on lightweight runtime environments is focused on serverless computing, so in Section 2.2 we present existing recent proposals that might be beneficial in devising our own solution.

2.2 Serverless Computing

Serverless computing is a computing model that aims to hide the server infrastructure as much as it can from the user so that the developer does not have to worry about lower level configurations. Functions as a Service (FaaS) is one of the most recent serverless computing solutions in the cloud ecosystem. The first cloud provider to adopt serverless platforms was Amazon with AWS Lambda [3], later on other major cloud providers followed with solutions such as Google Cloud Functions [19], Microsoft Azure Functions [43], and IBM Cloud Functions [28] [16]. There are open source solutions for serverless computing, such as OpenLambda [26] and Apache OpenWhisk [2], with the latter being analyzed in greater detail later on this section.

The key points for a serverless computing environment are:

Isolation An application should not be able to have unwanted interaction with another application. This isolation has to take into account different resources such as memory, storage space, and CPU.

Resource Provisioning One should be able to limit the resources that an environment is able to use. Once again this applies to the three resources previously mentioned, more specifically how much memory and storage space one application can allocate and for how long can these resources be occupied.

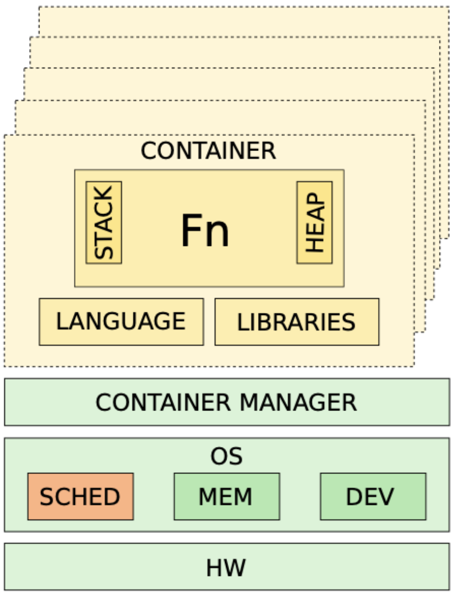
Flexible Programming Model In order to support as many services as it can, most of the platforms providing FaaS services usually support multiple programming languages.

Another key feature of serverless computing is that the model is seen as being stateless, therefore an environment that executed a task can then be deleted, since there is no relation between executions. The three key points are very similar to what a container-based system provides, plus the fact that it is stateless, would allow containers to be deleted as soon as they finished executing the tasks, thus freeing resources immediately. In fact most of the FaaS services are implemented using container-based solutions such as LXC [38], Docker [11], Windows Containers [62], and Kubernetes [33] [16]. Some cloud providers also have VM-based solutions such as Amazon's Firecracker [14] and Kata Containers [31].

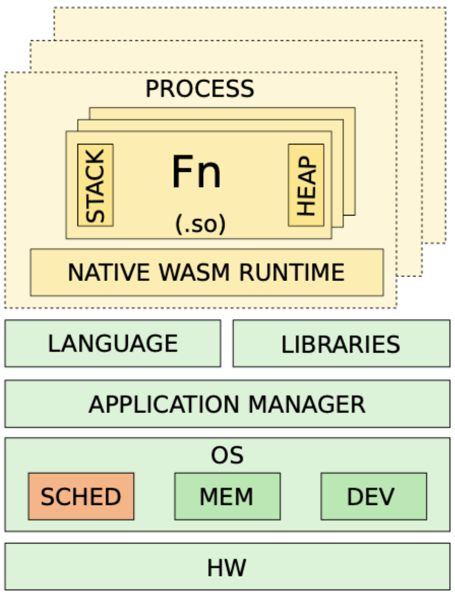
Recently, there has been research on serverless computing deployed in edge environments, since a serverless approach can offer on-demand resource provisioning, which fits adequately with the lower computation capabilities of the devices present at the edge. Cloud providers are also starting to offer solutions that deploy serverless solutions (such as FaaS) at the edge. This includes Amazon's AWS Lambda@Edge [1] that essentially runs an FaaS service in its CDN's points of presence.

In Section 2.1.1 we have discussed virtualization technologies, that were divided into two categories: virtual machines and containers; it's known that the latter is more lightweight than a virtual machine in terms of startup time and CPU usage, but it is still not ideal for an edge environment where the devices with less capacity may not be capable of running multiple containers. Therefore, a key research challenge in the context of serverless computing at the edge is developing runtime environments that are more lightweight and can provide similar levels of isolation.

The most recent solutions resort to WebAssembly [22] to create an isolated environment, similar to containers. These solutions use WebAssembly (Wasm) as a portable binary format and a runtime environment to run the binary [16] [23]. Some examples of such runtime environments are Google's V8 [55], Mozilla's SpiderMonkey [53], and Microsoft's Chakra [8]. Usually the runtime environments are based on browser engines, since these have APIs that support most system features (disk I/O, network communication, and tasks executors). The main factor that make Wasm-based solutions more



(a) Containers runtime environment architecture



(b) Wasm-based runtime environment architecture

Figure 2.2: Comparison between the architectures of containers and Wasm-based solutions architecture [16].

lightweight than containers is that the language environment (such as libraries) is the same for all the tasks executing (see Figure 2.2(b)). Figure 2.2 presents the models of both a container and a Wasm solution, where we can see that in the latter, there is an extra layer of shared resources between different running tasks.

Next, in Sections 2.2.1 and 2.2.2, we will detail two existing solutions for serverless computing that differ on how they handle the deployment runtime, one using containers and the other using WebAssembly sandboxes.

2.2.1 Apache OpenWhisk

OpenWhisk [2] is a distributed open source serverless cloud platform, that provides the users with an interface similar to FaaS. In order to trigger the execution of a function, a command is sent from an *event producer* which uses a *trigger* to start an *action*. The *event producer* can be anything that uses the OpenWhisk API, exposing a service as a *feed* (stream of *triggers*). The *trigger* is what characterizes a request, e.g., the endpoint to which the request is sent to. The *action* is the function code itself, which will be run depending on the *trigger*. Finally, the coupling between the *trigger* and the *action* is achieved through a *rule*.

The architecture of OpenWhisk is comprised by five different services:

Nginx which exposes the public endpoints, through a stateless web server that can easily be scaled (by replication).

Controller is the layer that implements the OpenWhisk API, and is responsible for performing authorization and authentication.

CouchDB is a JSON data store, that maintains the state of the system, storing the credentials, the metadata, namespaces, the definitions of actions, triggers, etc. The controller uses it for executing both authentication and authorization.

Kafka acts as a buffer storing the messages sent by the controller that will eventually be consumed by the Invokers.

Invoker is the final stage of the execution process, which is responsible for spinning up a new container and inject the code into it. After the execution, it stores the results in CouchDB.

OpenWhisk has become more popular lately for its ease of use and its vast support, from multiple resource management platforms (Kubernetes, Docker, Mesos) and support for different programming languages (e.g., Go, Java, Ruby, Python), to its scalability. Systems such as this can be an important component of the envisioned solution, by taking care of the resource management layer through a lightweight solution, providing scalability and robustness.

Unfortunately, the architecture that comprises OpenWhisk and that makes it easy to use, scalable and robust is also what makes it harder to deploy on the edge, mainly because of its resource requirements. Nevertheless, there have been efforts to address this limitation, through specialized implementations like Lean OpenWhisk [35].

2.2.2 aWsm

An alternative to container-based environments can be achieved through the use of WebAssembly as previously mentioned. These usually resort to a runtime environment such as Google's V8, which can be seen as a JavaScript virtual machine. Even though this virtual machine is much more lightweight than a normal VM or even a container, it is still an additional layer, which has an impact on performance when compared with a native solution.

The authors of [16] propose a solution that implements a native Wasm compiler and runtime framework, named *aWsm*. This runtime provides a higher level of resource sharing between functions as we presented in Figure 2.2, and enables a fine-grained control over the resources by bypassing the Linux Kernel, and taking full control of execution properties such as scheduling, sandbox switching, multicore execution, and profiling. The *aWsm* compiler is Rust-based and uses LLVM to enable hardware or software sandbox isolation.

The results presented by this work are very promising, mostly regarding memory footprint where, even though the examples used are considerably simple, the memory footprint was shown to be only around 100KB, with startup times of up to 39μs depending on the task. There is a cold-start cost due to the memory allocation and other initialization procedures but, as described by the authors, in an Edge environment this would only have impact once per function, when the runtime was running the function for the first time after its start.

Besides being an example of lightweight virtualization, this work provides a runtime that is capable of handling the control back to the developer, which is an important feat if the runtime will be used as a component of a more elaborate solution. Nevertheless, the maturity of this technology is a downside, since it is still in a novelty even outside the serverless computing subject.

2.2.3 Discussion

The biggest advantage of a Wasm-based solution is its lightweight environment, even when comparing with containers. In [23] the authors compared their solution (Wasm-based) with OpenWhisk (an open-source solution using Docker) and they got promising results considering multiple clients interacting with a system with different access patterns (see [23] for more details). One important observation made by the authors is related with the latency difference between the solutions, where even though a warm container would be faster at executing the task than the proposed solution, the unpredictability of user access would lead to some requests being executed in cold containers. In these cases, the time it would take for the container to startup, execute the task, and answer back to the client, would be, in some cases, an order of magnitude greater than the proposed solution.

Since containers are a more established technology and already used in production, these have been submitted to several improvements performance and have also evolved in terms of security mechanism. With time, Wasm solutions might also evolve to be as fast as containers.

Serverless computing is a promising model to employ at the edge since it enables task offloading, with small impact from the runtime, such as the Wasm-based solution we presented. Nevertheless, it implies that the runtime disregards any state between computations, so this restricts serverless deployments to stateless services. Our framework should allow applications to be deployed to the edge, regardless if they are stateless or stateful. This poses a significant challenge in the migration of components of an application, but in this case the benefits should outweigh the costs.

Next we will discuss other works that follow a similar decoupling between the environment and the execution tasks that allow execution portability through computation offloading.

2.3 Computation Offloading

In this section we will analyze existing solutions for task offloading. Computation offload can have different approaches such as the integration of RPCs (Remote Procedure Call) in the application that send a request to another machine in order for this to execute the task and return its result, or more complex methods such as the one used in COMET. Computation offloading aims to achieve some kind of improvement for a given task, through executing it in other computational resources. This improvement can be time related, by executing a task faster on a more powerful machine, it can also be power related, where in environments that are mostly composed of mobiles and IoT devices, the power supply is limited, so decreasing the amount of computation executed at the end device translates into longer battery life.

The usage of computation offloading usually implies that the application developer has to have a prior knowledge and reason about what would be advantageous to run in each environment, and it would be harder to change the behavior after deployment. This kind of hard coded solutions might not be ideal in our target environment since migrating at runtime is a key need when the workload to which applications are subject may vary frequently. Still some techniques adopted in code offloading can be useful to take into account when designing the final solution.

These solutions typically consider devices with lower computation capabilities and with limited power supply, with these being very common in the edge of the network. Moreover, a deployment solution between edge/cloud that follows an osmotic computing model should be able to migrate services between both environments, with service migration (see Section 2.4) being a particular instance of computation offloading.

Next, in Sections 2.3.1 and 2.3.2 we present two solutions with very distinct ways of offloading computation. MAUI employs an RPC-based solution, whereas COMET implements a Distributed Shared Memory model that enables execution transfer for any part of the application.

2.3.1 MAUI

MAUI [10] is a system proposed in 2010 that offloads mobile code to the infrastructure with a big focus on energy awareness. This system uses RPCs to offload the computation to a server located in the cloud instead of running the task locally. The system uses Microsoft's .NET CLR (Common Language Runtime) and its reflection API to inspect which methods are marked as *remoteable* through an annotation in the application source code.

One important factor taken into account in the developing of the MAUI system was the means through which the offloading was done, more specifically 3G and Wi-Fi. The work revealed that even though 3G is a nearly ubiquitous technology, it consumed three times more energy than offloading via Wi-Fi because of its longer RTTs (Round-Trip

Time) and limited bandwidths. This made the authors reconsider using 3G, with battery duration being one of key focuses of MAUI.

There are three main building blocks that make the MAUI system, both on the client and the server sides (with a slight nuance which we will point out). The MAUI runtime architecture is composed by a client/server proxy, a profiler and a solver. The proxy is responsible for the data transfer and the control of the offloaded methods, the profiler collects measurements such as size of the state transferred, size of the state returned, and CPU cycles for each offloaded method. Finally, the solver is responsible for using the data collected by the profiler and solve the equation that determines if a given method should be offloaded or not, to maximize the device's battery duration. It also takes into account the latency between the endpoints. In the client, the solver is just an interface to the solver present in the server, so that the equation solving does not drain battery from the user device. There is an extra component in the server called the controller that is responsible for authentication and resource allocation for incoming requests, in order to instantiate an application. The communication between the proxies and the solver present in both endpoints is achieved through RPCs.

As described previously, the component responsible for deciding which methods should be offloaded is the solver. It takes a global view of the program, which methods are remoteable and the measurements taken for each of these methods. Then it tries to find a program partitioning strategy that minimizes the energy consumption. This is done through building a call graph that identifies what other methods a method depends on and the energy consumed by each of them. An example of a face recognition application presented by the authors of MAUI is provided in Figure 2.3.

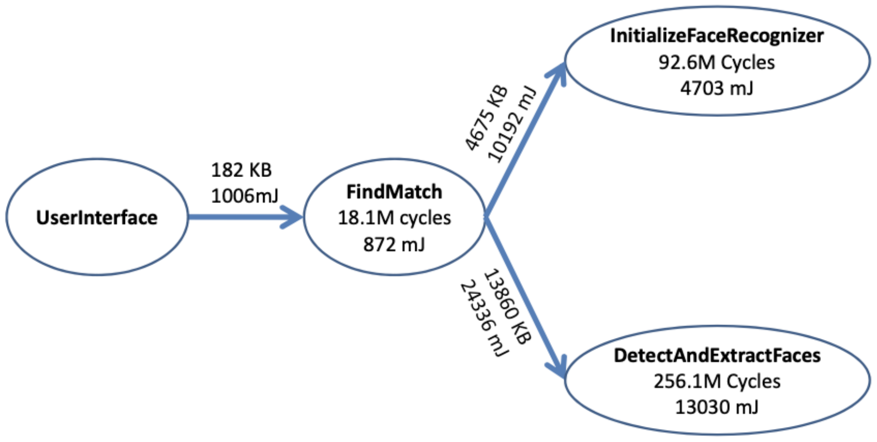


Figure 2.3: MAUI call graph of an example application [10].

The authors designed MAUI to handle failures having in mind that mobiles are, as the name says, mobile devices, so losing contact with the server would be a usual event, therefore handling failures is as simple as implementing a timeout mechanism that after not getting responses from the server, the control is given back to the device, and it might try to either contact another MAUI server and invoke the operation there, or simply run it locally.

One particular challenge that will be frequent throughout the analysis of the mobile offloading solutions is the fact that the offloading is done with a unidirectional mindset, this meaning that, ideally all the code should be offloaded to the most resourceful server, whether it be to minimize power consumption like in MAUI or to decrease the time it takes to run the task (as we will see next). In an edge computing environment, this is not true, since it might be best to offload the computation to the edge or to the cloud. It is not only a question about if it should be offloaded or not, like in mobile offloading, but instead where it should be offloaded to.

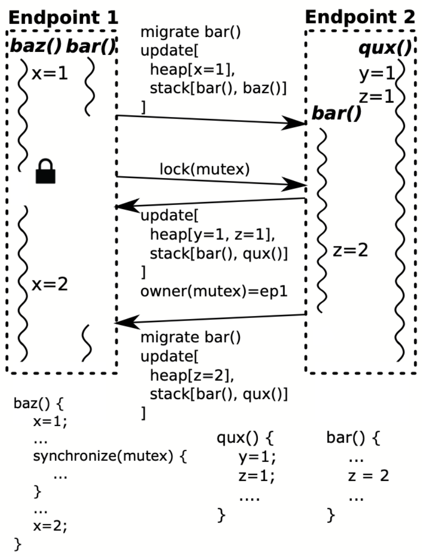
2.3.2 COMET

COMET [20] (Code Offload by Migrating Execution Transparently) is a system that aims to augment smartphones and tablets with machines present in the network by offloading computation to the latter. It can achieve this through the usage of a DSM (Distributed Shared Memory) mechanism that exchanges updates of the memory view between end-points.

COMET uses a virtual machine to access the memory of a given program, which allows the system to control multi-threaded applications since the access is done for the whole program and not per thread basis. The virtual machine used was DalvikVM which, at the time, was the virtual machine used to run executables on Android.

The DSM model used in COMET has a very fine grain, allowing the control of an object's fields. This is easier since the DalvikVM uses a register-based architecture, unlike the stack-based Java Virtual Machine, that allows a more transparent relation between a register and an object in memory. The computation migration in COMET is as simple as sending a trigger to the endpoint where we want to run the task, followed by a VM synchronization message. The VM synchronization mechanism relies on a message with the current heap and stack content, suspending all the threads to guarantee that there are no changes while building the message. A set of objects, called *Tracked Set* is kept to identify which objects are present in both endpoints, with a bit specifying which ones are *dirty* (were changed). This decreases the amount of data that has to flow through the network. In Figure 2.4 we can visualize the migration process.

An important question that is crucial for any migration mechanism is to decide the situation in which migration should occur. In COMET, the authors use a methodology called τ-Scheduling. This is a very simple approach that measures how long a given thread has been running without executing client-only code. This code is characterized by invoking native methods that might rely on a device resources (e.g., file system, display) or depend on native state. In case it exceeds the user provided parameter τ, it migrates the task to the server. A basic example is to set the τ to twice the RTT between the client and the server, afterwards changing it to twice the average of the VM-synchronization



```
migrate bar()
update[
  heap[x=1],
  stack[bar(), baz()]
]

update[
  heap[y=1, z=1],
  stack[bar(), qux()]
]

migrate bar()
update[
  heap[z=2],
  stack[bar(), qux()]
]
```

```
baz() {
  x=1;
  ...
  synchronize(mutex) {
    ...
  }
  ...
  x=2;
}

qux() {
  y=1;
  z=1;
  ....
}

bar() {
  ...
  z = 2
  ...
}
```

Figure 2.4: COMET migration mechanism [20].

time.

By relying on a virtual machine, COMET was able to achieve multi-thread migration while keeping some guarantees on the order that the operations are executed. Within a single thread the operations are totally ordered according to the order in which they happened, while across threads a "happens-before" strategy is adopted. To this end, when thread A acquires a lock that was previously held by thread B, all the operations that were executed up until the release of the lock are considered to have "happened before" all the operations that thread A will execute after acquiring the lock.

The proposed solution relies on some mechanisms that are not suitable in our scenario, in particular it assumes that it is always better to offload computation to the most resourceful endpoint, following the τ-Scheduling criteria. Another small factor to take into account is that the technology that was used in this work, more specifically the DalvikVM has been replaced by Android Runtime (ART) in more recent Android versions.

2.3.3 Discussion

As previously mentioned, the solutions presented here always follow a unidirectional ideology, where there is always a better place to run the computation (the endpoint with more resources). This might not be true since in some of our case studies such as Pokemon Go, offloading to the cloud is not always better, neither is always offloading to the edge. The simplicity of the schedulers presented here are adequate to a mobile-server context, whereas in a cloud-edge environment, there are a lot of cases that would make deploying to a less resourceful server at the edge advantageous, for example if many requests are coming from the same region, migrating to a Point-of-Presence close to that location might provide lower latency. The DSM technique proposed in COMET might also not scale since keeping a copy of the task running on multiple nodes leads to waste of resources.

Next we will discuss a different technique to achieve execution transfer to another machine called *service migration*.

2.4 Service Migration

The concept of *service migration* is a particular instance of *computation offloading* since it follows the same main idea of possibly moving the execution of computation to a different location. The term migration has some underlying meaning that the service transfer is conducted at runtime, which is very important on the osmotic computing model. This methodology has additional challenges, since it implies that the state has to be transferred or use some kind of replication that is keeping the state updated on multiple devices, thus allowing the transfer of computation at any time to one of these devices. This section analyzes migration solutions that either propose some generic deployment framework, focus particularly on decreasing migration related disadvantages or propose migration solutions that take into account limitations of some edge devices.

The solutions presented next in Sections 2.4.1, 2.4.2, 2.4.3, 2.4.4 and 2.4.5 target different aspects of service migration and the environment the migration is performed on. This can be migration technique (e.g., VM migration) or task scheduling, and it can focus on a specific metric such as quality of service, consumption of energy, or optimal resource utilization.

2.4.1 Live Service Migration in Mobile Edge Clouds

In [40] the authors propose a layered framework for live service migration using two virtualization methodologies, containers and virtual machines (VM), for edge to edge migration. The proposed solution could be applied in a cloud-edge setting since the framework is environment agnostic, even though the paper proposal is to follow the user as it moves, migrating the services always to the closest mobile edge cloud (MEC).

The framework uses Linux Containers (LXC) [38] and Kernel-based Virtual Machine (KVM) [34] as container and virtual machine implementations respectively. Initially the authors envisioned the full service migration through full system migration, this meaning that whenever a service migration was issued, the server would have to suspend the container/VM that the service was running on, and export the full state to a file. This file would then be transferred over the network to the destination machine where the service would be resumed using the received file. Both LXC and KVM support the idea of *checkpointing* (*saving* in KVM). The authors soon realized that transferring the whole state through the network was not feasible since in a container-based solution the state can have around 400 MB, and in a VM it is even worse with the state being as big as

2.7 GB. The bigger the state, the longer the service will be interrupted and unusable, so minimizing the size of the state transferred is crucial for a viable solution.

The proposed framework relies on a state partition strategy that splits the state into three layers: base layer, application layer and instance layer. The base layer is responsible for the guest OS, kernel and the rest of the system dependencies without including any applications. On top of this layer we have the application layer that includes the service applications and their dependencies, where the service applications are instantiated with an idle state and any application-specific data. Finally, the instance layer contains the service application running state (in-memory state). A service migration is the comprised of migrating individually each of these layers. This solution can decrease the amount of data transferred since ideally, the destination machine only needs the instance layer, assuming it had previously run this service. The authors also took into account a possible solution that only has two layers by joining the application and instance layer into one. A flowchart with the migration process and its dependencies is shown in Figure 2.5.

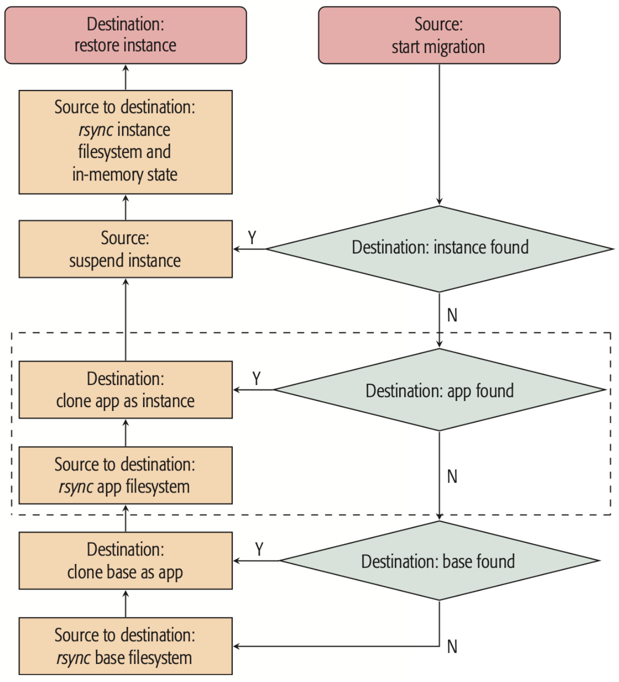


Figure 2.5: Migration mechanism in the proposed three layer model [40].

Another tool that the framework leverages to decrease transfer times is *rsync*, which does incremental data transfers, identifying which files are already present in the destination. It also applies compression to decrease transfer times.

The results proved that the proposed three layer model not only can have up to three times lower migration times than the simpler two layer model, but it can also reduce the amount of data transferred, in the best case, by thirty-six times. This case specifically has a very big installation footprint and a considerably high RAM usage, thus taking the most advantage of incremental transfers and a more partitioned model.

An important improvement that is left as an open issue, is the usage of a layered file system such as overlayFS that creates different layers for each operation modifying the file system, thus allowing containers to share the same layer without duplicating data that is not altered. Unfortunately, at the time that the research was done, LXC did not support overlayFS. This solution has a lot of improvements that an edge-cloud migration system can take advantage of. However, the migration process done in this work is only based on user relocation, whereas in an edge-cloud environment there are a lot of other determinant factors to take into account, such as quality of service (QoS), data locality, application workload, etc.

2.4.2 ABEONA

ABEONA [48] is an edge to cloud task migration system that focuses on decreasing the energy consumption on edge devices by migrating tasks to the upper layers (fog or cloud). In fact the model described has a three layer hierarchical infrastructure: **i) Edge** - the devices in this layer execute tasks as a cluster, since individually they do not have the required resources (i.e. computing or storage) to execute them in practical terms; **ii) Fog** - The middle layer is formed by devices with individual computing capacity, usually with low power. Given that these are capable of executing tasks individually, this layer supports task migration and has a lower response time than the cloud, because of its proximity to the end user; and **iii) Cloud** - This is the top level in the architecture, providing the highest computing capabilities, thus supporting the heavier tasks that less capable devices (such as the ones on the fog) can not perform in useful time. This layer usually consists of server-grade machines.

Even though the proposed solution has promising results in an energy-saving perspective, it still lacks detail on how some mechanisms, such as migration and scheduling, are implemented. The decision-making process that build the scheduling algorithm is also obscure and not detailed in relation to when a migration occurs or to what layer it should migrate to. This is mainly due to this still being a preliminary work.

2.4.3 SEGUE

A very important factor in service migration is quality of service (QoS). SEGUE [64] aims to build a system that is focused in maintaining QoS through a tweaked distance-based Markov Decision Process (MDP) and by migrating users between computational resources deployed at the edge (these are called *edge clouds*). The system is formed by four main modules:

State Collection Module This is the module that gathers all the edge clouds real time network statistics, workloads, client mobility patterns and other variables that serve as input for other QoS assuring modules.

QoS Prediction Module This module monitors and predicts the QoS. It is also responsible for detecting possible QoS violations, thus triggering a search for the new optimal edge cloud, which will then migrate the service to.

Edge Cloud Selection Module As the name says, this module selects the edge cloud, based on the information gathered by the State Collection Module and a trigger emitted by the QoS Prediction Module. It finds the optimal edge cloud by running the MDP and returning the optimal solution found through the process.

Service Migration Module Lastly, this module is responsible for performing the service migration by transferring the VM state from the source edge cloud to the outcome of the Edge Cloud Selection Module.

Unlike the solution proposed in *Live Service Migration in Mobile Edge Clouds* (the first solution presented on this section) that mainly focused on the implementation of migrating a process, SEGUE aims to address the decision-making process on how and when to migrate. Other migration systems that use MDP had been proposed previously to SEGUE (such as [61]), but the authors emphasize that SEGUE takes into account the dynamic aspects such as the network state and the workload for an edge cloud, thus granting the model a more detailed input upon which it can find a better optimal solution. Simpler MDP-based solutions only took into account the distance in hops between the edge cloud and the client, such that the optimal solution in that model could be an edge cloud that was completely overloaded with other user requests that would then offer a worse response time to the client than a further away edge cloud that had no other users issuing requests to it.

The authors also address the fact that running MDP is a computing intensive task, so running MDP in short intervals might overload the servers, whereas running it in longer intervals might induce a lazy migration mechanism, where the QoS deteriorates rapidly and there is still a long time before the MDP is run again to find a better edge cloud. The work presents a QoS focused approach, where the MDP is run once a QoS violation is predicted. A violation is characterized by exceeding a QoS threshold.

This work presents a well detailed and somewhat complex solution for the *when* and *where* migration problems, whereas the solution for the migration itself seems primitive. A combination of the previous work (*Live Service Migration in Mobile Edge Clouds*) and this one might lead towards a viable edge cloud deployment solution.

In addition to SEGUE, the work presented in *Dynamic Service Migration in Mobile Edge-Clouds* (DSMMEC) [61] was also studied, but since SEGUE presents a more elaborate MDP model, a discussion on DSMMEC does not bring any additional information to inform our work. It is important to notice that DSMMEC has a very in depth description of their MDP model, which might be useful to fully comprehend SEGUE's model.

2.4.4 FogTorch

In [6] the authors propose FogTorch that focuses on providing a solution to a resource constrained environment. The key challenge stems from the resource management layer, since in reality, the nodes on which the tasks will migrate to, do not have infinite resources. Deploying a task implies that the node on which we will deploy has to meet the minimum requirements to execute the task. These requirements can be related to the software, hardware, or even the properties of communication links. In this work the authors focus on developing a tool that given a set of inputs, can return optimal deployments that meet all requirements.

This kind of approach implies that a task requirements are known, which in the real world might not be easily achievable. In the mentioned work the authors emphasize that it does not bind to any particular specification format of the software/hardware requirements or offerings. This is also an interesting challenge in service deployments, which is usually solved by a previous knowledge of what requirements a task needs or is intensive on, and doing an overestimation of these.

FogTorch is the tool implemented in Java described in this work that aims to solve the constraints described in the problem. The inputs that it takes are separated in two sections: the infrastructure and the application.

The infrastructure specification is formed by four main components: *things* (IoT-like devices), fog nodes, cloud data centers, and the links between the components. A *thing* has an identifier, a location, and a type. A fog node is characterized by an identifier, location, hardware, software, and all the *things* directly reachable from this node. The cloud data centers have an identifier, a location, and the software available. Finally, the links have the two entities that they connect and a QoS profile, where this profile specifies the latency and the upload and download bandwidths.

An application specification is composed by three components: software components, interactions and the *things* it needs to reach. The software components have an identifier, and the software and hardware requirements. An interaction is characterized by having a tuple with the different software components it connects and the QoS it needs. Lastly the *things* an application needs to reach have a minimum QoS profile, the software component that aims to reach the *thing*, and its type.

The main limitation of this work, is in its very exhaustive specification of both the infrastructure and the application. Even though the authors use a small use case with two cloud data centers, three fog nodes and three application components, the set of viable deployment configurations has more than fifty possibilities. It does not seem feasible to specify such information for hundreds or thousands of fog nodes and applications. A dynamic environment where nodes and applications are being added and removed from the system, does not seem ideal, since the problem to be solved described as *components deployment problem* (CDP) is proved as being an NP-hard problem with a worst-case complexity of $O(N^s)$ with N the sum of fog nodes and cloud data centers and s being the number of software components of a given application.

2.4.5 Scheduling Latency-Sensitive Applications in Edge Computing

In this work [51] the authors propose a framework that aims to reduce the latency in applications between the user and the server. The main challenge, is the distance from the server deployed in the cloud to the user, so the authors propose a framework that takes advantage of the edge nodes and deploy the server closer to the user. This work also takes into account techniques such as CDNs that aim to reduce such latency. However, as pointed out by the authors, CDNs are used as static content delivery points, whereas the kind of challenges this work aims to tackle are not solvable by simple caching mechanisms.

The framework has two main components, VM evaluation and task scheduling. The VM evaluation is firstly done by assigning each VM a quality score, with this being the combination of three main factors: connectivity, bandwidth, and resources (e.g., CPU, storage, memory).

The connectivity score takes into account a set of possible paths between a group of users and a node, and the delay for each of those paths. The bandwidth score is an average of the bandwidth available on each path, taking into account the amount of users on each path. Finally, the resources score is a function defined by the provider that estimates how many requests a given machine can execute, and how many it expects to receive. The combination of these three scores yields the quality score for a given node. The framework uses a harmonic mean, thus favoring smaller values when there is a big discrepancy in the three values. This assures that VMs with very high and very low scores are penalized since a good VM score should translate into a good score across all components.

The framework then aims to maximize the quality of the final scheduling. This is the sum of the individual qualities of each service assigned to a VM.

In the experiments conducted by the authors the edge-based scenario was compared with two others: a CDN-based one and a cloud-based one. As expected, the processing delay was much lower in the cloud scenario, since the amount of computational resources is much higher there. Even in the CDN the processing time was lower than in the edge solution, but the network delay was significantly higher between the edge and the remaining solutions. The overall service delay (processing delay plus network delay) was still lower in the edge, with it being around 1.3x faster in comparison with the CDN and 2x faster than the cloud. These improvements start to crumble when dealing with too many users (around 1000, changing according to the VM resources), since the processing capabilities of the VMs become a bottleneck in the system. An important note to take from this work is that implementing a good scheduling algorithm goes a long way to reduce the network delay on the edge, as presented in the results when comparing a cloud scheduler versus the proposed scheduler solution.

The downside of the work presented is that it only tackles a small part of the problem that a cloud-edge deployment solution should take into account. Namely, it does not address any kind of service migration, and the virtualization technique used (VMs) has very high resource requirements for deployments on edge nodes.

2.4.6 Discussion

Considering the solutions presented in this section we note that all of them have their own advantages and drawbacks, since they target different aspects. These are the main observations that we can take from the presented works.

Live Service Migration in Mobile Edge Clouds targets migration between Mobile Edge Clouds, presenting a very complete work, from which we can retain the *checkpointing* technique that they used to migrate the containers. However, the software used for container management (LXC) would probably be switched for a more recent alternative (e.g., Docker).

ABEONA presents a preliminary work on energy-aware edge to cloud migration, where, even though there was lack of detail on how the system was implemented, the architecture and the cloud-fog-edge structuring presented possible hierarchical relations to be used in our solution.

SEGUE presents a very well detailed MDP-based migration system, aiming to achieve high QoS. Even though it is not expected that our solution model will tackle optimal migration regarding QoS with a mathematical problem-solving approach, it is still important to take into account this work, mainly since throughout the literature, multiple works use MDP to achieve optimal migration taking into account a specific metric (QoS in this case).

FogTorch tackles task deployment in a resource constrained environment, which makes the work particularly interesting since this challenge is also present in our scenario. Nevertheless, the solution proposed was too exhaustive in the model definition, which is not feasible in a realistic environment.

Scheduling Latency-Sensitive Applications in Edge Computing proposes a simpler mathematical model for providing an optimal deployment resource wise which can be useful for our deployment scheduling. However, it uses VMs to achieve migration, which, as discussed before, is not feasible in the edge.

2.5 Deployment Frameworks

The interface provided by the service models described in Section 2.1.1 is generally agnostic to how resources are provisioned and how tasks are scheduled internally in the datacenter cluster. Next we will analyze systems that make such feat possible. Unfortunately, most of the cloud management solutions are considered trade-secret and normally when they are published, it implies that the company is no longer using it and there is a newer solution. Nevertheless, Google has made public some of their infrastructure management systems that we will discuss next, together with another solution (Mesos) that has an open-source implementation (Apache Mesos), the widely used Kubernetes and lastly ENORM, a cloud to edge deployment framework.

The following Sections 2.5.1, 2.5.2, 2.5.3, and 2.5.4 present deployment solutions in a cloud environment. These serve as a starting point for an edge solution since they address common challenges in deploying tasks on demand regardless the environment it deploys on. Section 2.5.5 presents a first approach on application deployments following an osmotic computing model.

2.5.1 Borg

Borg [56] is described as a *'large-scale cluster management system'* developed at Google to enable Google developers to deploy tasks throughout its huge fleet of machines. The system is composed by multiple *Borg cells*, where a cell is located in a single datacenter and on average has around ten thousand machines. These machines are connected through high-performance datacenter-scale network fabric.

A *Borg cell* is composed by a *Borgmaster* and multiple *Borglets* (one per each machine), whereas a *Borgmaster* is a logically centralized process that is responsible for handling client requests (job creation and lookups) and communicating with the *Borglets* . Even though it is a logically centralized process, the *Borgmaster* uses a Paxos-based store to replicate the cell state in each of the five replicas. The *Borgmaster* is also responsible for scheduling the tasks, where it takes two different approaches called the *worst fit* and the *best fit*. The first one aims to spread the load across all machines and leaving room for spikes. This technique translates to a bigger fragmentation, especially on bigger tasks. On the other hand, *best fit* aims to pack a machine as tightly as it can with tasks, leaving the others free to receive bigger jobs. This technique is more prone to penalizing miscalculations of a task requirements or jobs with bursty loads. In fact the scheduling technique employed on Borg is a hybrid between these two.

The *Borglet* present on each machine is responsible for performing the operations themselves, such as starting and stopping tasks, managing local resources, logging and reporting the state to *Borgmaster* .

In this work, one of the factors that stands out is the scale of the ecosystem that Borg is deployed on. Since it has so many resources available, Borg is very good at addressing most of the challenges, such as fault-tolerance, task scheduling and scalability. This has its disadvantages, for example, a busy *Borgmaster* can use around 12 CPU cores and 50GiB of RAM. This kind of requirements is something that only makes sense in a cloud infrastructure, since in an edge environment these would never be met. These high resource costs are not due to bad design or implementation of the system itself, but instead the magnitude of the number of machines that the system has to deal with.

2.5.2 Omega

Omega [50] was designed as a solution for cluster management following Borg. It aimed to achieve better performance from the beginning by employing a more optimistic approach on concurrency, more specifically on its scheduling policy, while still resorting to most of the patterns present in Borg such as the centralized Paxos-based store. Omega studies four kinds of schedulers: monolithic, statically partitioned, two-level and shared-state.

The monolithic scheduler is usually a single process that does not support any kind of parallelism and should have full knowledge about the system and the characterization of the workload. This is clearly not a viable solution for Google since it simply would not be able to cope with the amount of task requests it would get.

A statically partitioned scheduler assumes that the cluster it will manage is statically partitioned, so that different partitions support different behaviors. This would lead to unbalanced workloads per partition and therefore, suboptimal utilization of the resources.

A two-level scheduler is, as the name implies, comprised by two levels. First a centralized coordinator that partitions a big cluster in sub-clusters and decides the amount of resources per sub-cluster. The second level is comprised by multiple scheduler frameworks, where each of these has a set of resources (sub-cluster) assigned. We will discuss this scheduling technique in more detail when analyzing Mesos.

Lastly, the approach followed by Omega, is a shared-state scheduler. What makes this technique interesting is its philosophy of free for all, where all the schedulers have access to all resources. This optimistic approach on concurrency can have conflicts when multiple schedulers try to allocate the same resources. Omega keeps a master copy of the whole *cell state* (resource allocations) that frequently updates the local copies on each scheduler. A scheduler can lay a claim to any resources existing in the *cell*, even resources that have already been acquired by other scheduler without the first knowing of such acquisition. This operation is done in an atomic commit to the shared state. Eventually in case of a conflict, the scheduler will poll the shared copy and will realize if its commit was accepted or not, if it was not it will just try again. To prevent this kind of conflict from causing starvation usually, Omega schedulers use incremental transactions, thus allowing non-conflicting operations to be committed successfully even if there are ongoing conflicts.

The scheduling technique employed by Omega is the core reason of why it was developed even with Google having already a cluster management solution (Borg). Eventually most of the advantages that Omega has over Borg were integrated into Borg. Due to this, Borg is still the most used cluster management system in Google. Unfortunately it still inherits most of the problems of Borg when considering our deployment scenario. Using a system tailor made to handle massive amounts of resources and be as generic as it can task-wise, might incur in an overhead for unnecessary features or guarantees.

2.5.3 Mesos

Mesos [27] is a resource-sharing system for datacenters. Even though the resource managing component is not that interesting for this thesis, the scheduling policy employed is an example to take into consideration when designing a deployment solution.

As previously mentioned in the Omega analysis, Mesos employs a two-level scheduler that consists of a master and the frameworks themselves. The master knows all the resources present in the system and decides how many resources to offer to a framework, employing its own policy (e.g. fair sharing, priority). The frameworks are comprised of a scheduler and an executor. Instead of forcing the framework to specify the task resource constraints, which can end up being far from the resources utilized, the Mesos master makes offers to the framework schedulers which can be rejected. This can lead to unnecessary communication for resource offers that the master sends to the schedulers and just get rejected. To mitigate this, Mesos supplies a filter mechanism that tells the master that a framework will always reject certain resource offers that do not meet the criteria.

The Mesos framework only by itself is not expressive enough to specify multi environment deployments. Even though it could be used as an underlying resource management layer (by deploying instances in the different environments), an environment-aware layer that migrated the tasks between the different instances would have to be built on top of it. This is due to Mesos being designed to be employed in a confined environment such as a datacenter. Nevertheless, some techniques used in Mesos can be translated to an eventual edge-focused solution, such as the resource offering methodology and the two layer scheduling system.

2.5.4 Kubernetes

Kubernetes [33] is a deployment framework that automates scaling and management for containerized applications in a multi node cluster. It has gained traction in the industry recently, enabling applications to be cloud native through its community and its ease of use. In Kubernetes, the basic unit is a *pod*, which can contain one or more containers. *Pods* can be managed as a group through a *stateful set* or a *replica set*. Both these sets aim to guarantee scalability and availability by watching failure events and saturation criteria and acting accordingly, by deploying pods to replace the failed ones or increase quality of service.

In order to handle the difficult aspects of managing multiple nodes such as information consistency, Kubernetes uses *etcd* [13] as its backbone to disseminate the cluster state and configuration metadata. *etcd* is a strongly consistent distributed key-value store that uses the Raft [44] consensus algorithm to ensure data store consistency across all nodes, ensuring fault tolerance and high availability.

Kubernetes also has networking solutions that can handle communication between *pods* in different *nodes*, apply traffic shaping or add routing rules. For this, Kubernetes specifies a Container Networking Interface, that can be implemented through plugins with different plugins suiting different needs.

Scheduling is also handled automatically by Kubernetes, more specifically by the Kubernetes Scheduler. It ensures that when a pod is scheduled to run on a node, the latter satisfies requirements such as hardware, software, data locality or inter-workload interference.

It's the flexibility of design that we see in components such as the CNI, the Scheduler and others, that made Kubernetes ubiquitous in the cloud environment. Nevertheless, in order to keep its availability and consistency properties, Kubernetes is highly dependent on *etcd*, and with *etcd* using the Raft consensus algorithm, the performance quickly degrades with variable latencies, limited bandwidth and high churn. Due to the latter, Kubernetes makes a great solution for a stable environment with low latencies, high throughput and low churn, such as a cluster in the cloud, but not so much for the nodes present in the edge.

2.5.5 ENORM

In [60] the authors propose ENORM, a framework that aims to scale applications from the cloud to the edge aiming to decrease the latency from the clients to the servers. ENORM is composed by five different components: *Resource Allocator*, *Edge Manager*, *Monitor*, *Auto Scaler* and *Application Edge Server*. *Resource Allocator* is responsible for tracking the available resources in an edge node, *Edge Manager* takes care of accepting provisioning requests and instantiating the services' containers, *Monitor* tracks the communication latency and other metrics, *Auto Scaler* dynamically scales the resources used by the services according to the metrics from *Monitor*, and lastly *Application Edge Server* is the application server that is partitioned from the cloud.

ENORM manages all these components to build a framework that can scale applications to nodes closer to the users and achieve a lower latency and decrease the amount of data transferred to the cloud by up to 95%.

Another contribution from [60] is the iPokeMon benchmark, which is a Pokemon Go like game used to validate the framework's performance. This server has to be manually partitioned at the function level in order to create the edge server version, which can be cumbersome to do in a real life application. iPokeMon served as an inspiration for the benchmarked proposed in this thesis, even though iPokeMon is a fully built application with UI, whereas the proposed benchmark focuses on the backend aspects of such applications such as tracking application metrics, portability and deployment.

One big disadvantage of the work conducted in this paper is that the middle tier that contains the edge nodes only contains one edge node effectively, which may hide unexpected drawbacks or resource consumption when dealing with multiple edge nodes. Furthermore, in this work the cloud is always responsible for making the decisions of provisioning edge node resources, which ultimately one might argue that in a real scenario with multiple edge nodes this model may not be able to diverge as much traffic as expected from the cloud. We believe that a model that allows edge nodes to take local decisions poses as a more scalable alternative. The lack of fault handling may also be indicative of a static scenario that does not change over time, which is not ideal since availability is expected to be lower in the edge computing paradigm when comparing with cloud computing.

Even though ENORM also achieves lower client latencies, its greater achievement is in reducing the data transferred. Taking into account how the solution partitions the servers and the databases, it is clear that ENORM's main focus is on data partition and data transferred which is an important topic, yet orthogonal to this thesis.

2.5.6 Discussion

Most of the solutions that we analyzed in this section, lack awareness of properties that are present at the edge, such as resource scarcity and frequent membership changes (the solutions usually use some sort of Paxos-based solution which has performance impacts when the system membership is very volatile).

The lack of edge awareness is due to the solutions being environment agnostic, which in this case is not ideal, since a cloud/edge deployment solution should understand what are the benefits between both environments, and leverage the migration to one or another. Moreover, these solutions were designed for managing datacenter level resources, which means that the managing system has very high resource usage, and since the datacenter is considered stable, the master nodes are centralized, tolerating faults by synchronizing multiple replicas. This kind of solution is not ideal in an edge environment, where using a centralized master with replicas might incur in a lot of communication to keep the replicas synchronized. A smarter approach might be to fully decentralize the system, with every node being simultaneously a worker and a master. It might also be worth building a hierarchical structure that can delegate deployments from the cloud down to the edge, with a node keeping track of its child nodes' deployments.

2.5.7 Summary

Throughout this chapter we discussed the two computing models that comprise our system model and possible techniques that could be used in a deployment solution at the edge, such as schedulers and resource management systems. Given the lack of edge-focused decentralized deployment frameworks, we analyzed three deployment solutions used in the cloud model.

In the next chapter we will elaborate on the assumptions made by our system model, and describe some features and requirements that the future work will have, providing hints on how to do and validate these features.

Chapter 3

LowNimbus

Even though, throughout this document, the cloud may be presented by mostly pointing out its downsides or limitations and thus, making it look like an unsuitable computing paradigm, it also has very positive properties such as its amount of resources and availability. Therefore, our solution should not aim to replace the cloud paradigm, but instead cooperate with it making the most out of its benefits and minimizing its drawbacks by empowering developers to easily extract benefits from edge computing.

Taking the previous into account, we envisioned a system where applications start by being deployed in the cloud, and then progressively spread throughout the world (i.e., the edge) to other nodes closer to clients. This led us to create a tree building protocol that uses the cloud as the starting point (tree root) and builds "paths" closing up on the clients.

In order to make the most of the cloud, our approach relies on it as the *fallback* node, where in this case, it is twofold a fallback: first when the clients can't find a deployment in a nearby node, they will go back to use the one on the cloud and second when there is a fault in the tree ultimately we might have to repair it with a top-bottom approach, starting from the cloud down (this is not true in all faults, see Section 3.1.3 for more details).

This dynamic deployment system will also have a base assumption that a client connected to a closer node will have a better quality of service (*QoS*) than when connected to a more distant one. This assumption is not true in many scenarios such as Wide Area Networks (WANs), where the distance between the user and the server is not so much influenced by the physical distance but instead by the number of hops and the quality of the links that form the network path between the client and server. However, in cellular networks which are a particular case of a wireless scenario, the distance between the user and the server is the physical distance to the tower plus the distance to the server. If we can deploy the applications in these cell towers, the determinant factor for the latency between the client and the server is now only the physical distance to the tower.

To spread the applications throughout a region we devised a tree building strategy that has as the starting point (tree root) the cloud and progressively builds a "path" closer to the (client) requests origin. Alongside with shortening the path to the server as pointed out earlier, by distributing the load from a central cluster to widespread nodes it will avoid saturating the bandwidth from a single point, which as discussed before is a real challenge of the cloud model.

A strong case that supports our solution are the recent advances in 5G networking and the existence of computational resources with some computation closer to the user. This reality has long been supported by research in edge computing that has increasingly deployed computation resources closer to the users, with cases such as CDNs [30], Mobile Edge Clouds [40], Fog Computing [41], etc.

The following sections discuss the system model and the separation of concerns between each service (Section 3.1), followed by a description of how the clients interact with our system (Section 3.2), and finally the implementation details that describe how the system was materialized (Section 3.3).

Throughout this chapter we will use terms that may not be obvious to the reader, so in order to provide a clear comprehension of the system we present a small list with terms and their respective definition:

node a physical machine that takes part in the system.

deployment the set of configurations uniquely identified by an ID that specify an application setting.

instance a container running on a node, uniquely identified by an ID and specified in a deployment configuration.

3.1 System Model

The proposed solution is a deployment platform that launches application containers throughout the system progressively closer to the clients. The system was designed with a microservice architecture in mind, mostly to separate the concerns and logic of the solution.

The solution consists of four microservices that run on every node, with these being: *Archimedes*, *Autonomic*, *Deployer*, and *Scheduler*.

Archimedes is a name resolution service that resolves identifiers (deployment or instance IDs) into addresses and the entry point for end users.

Autonomic is responsible for collecting metrics and deciding which actions to execute, aiming to maximize a strategy.

Deployer is the main component that executes the deployment tree protocol and the entry point for applications.

Scheduler is responsible for adding/removing the service containers, by interacting with the Docker daemon.

This solution assumes that all the nodes have a location tag associated with them (statically configured) and whenever a client makes a request, the request also has a location tag for the client's location, since as we have stated before, a closer node to the client would mean an improvement on *QoS*.

In order to understand how the system evolves we will present a broad overview of the interactions between the different microservices and the client from initializing the deployment at the *fallback* node until the deployment is extended to the closest node to the client.

The system user starts by adding the deployments that it wants running on our system by sending the configuration files describing them to the *fallback* node *Deployer* API. The *Deployer* will then send three requests. First it sends a request to *Archimedes* registering the deployment (without any instances). Then it sends a second request to *Scheduler* with the information needed to start a container from the image provided. Lastly, it sends one to *Autonomic* to register the deployment and start tracking its performance metrics (which are collected by a companion system that was developed in parallel to this solution, and hence, not the focus of this work).

Upon starting, the instance will send a heartbeat to *Scheduler* reporting as being alive. After receiving the first heartbeat *Scheduler* registers the instance in *Archimedes*, thus allowing *Archimedes* to resolve deployment requests to that instance. At this point the initial setup is finished and the deployment is now being served at the *fallback* node.

When a client wants to access an application being managed by our proposal, it starts by connecting to the *Archimedes* on the *fallback* node (the *fallback* node has to be known *a priori*). Then when it requests a URL, this will be intercepted by the *Archimedes Client* middleware (see Section 3.1.1.4) which will issue a resolve request to *Archimedes* containing the host and port of the URL. *Archimedes* will then execute its resolving mechanism (see Section 3.1.1.1) and respond with an address and port to a deployment instance. Next, the middleware layer will update the original request with the received host and port and execute it (see Figure 3.5). At this point the interaction between the application client and *Archimedes* ends, and the system now executes internal mechanisms to improve the *QoS* to the client.

First *Archimedes* will emit the updated deployment load as well as the clients request origins to the underlying overlay metrics propagation layer. Then, *Autonomic* will reevaluate the deployment with a strategy that aims to minimize the distance to the clients and in case it needs, it will readjust the deployment of the managed (and used) application to contain a node closer to the client. For now, lets assume the reevaluation will decide that the best action to take next is to extend the deployment to the closest node to the clients' location.

For this, the *Autonomic* sends a message to *Deployer* requesting to extend the deployment to the closest node. The *fallback Deployer* will now register the deployment in the other node *Deployer* (similar to registering deployment in the first step) and will send its information alongside the request. The closer node *Deployer* will execute the same procedure as mentioned previously upon registering a deployment, with the difference that it will also store the parent information for that deployment.

When the client issues a new request to *Archimedes* at *fallback*, it will now respond with a redirection to the *Archimedes* at the node it just added to the deployment of the managed application, which will make the client start using the new *Archimedes* to resolve deployments. At this point the process repeats and will progressively get closer to the clients.

3.1.1 Archimedes

Archimedes is a name resolution service that resolves IDs into deployment instance addresses while having request locality as its main focus. In this section we will detail the internal mechanisms that allows us to achieve the aforementioned.

First we explain in detail the resolving process in Section 3.1.1.1, followed by the table propagation that ensures deployment locality in *Archimedes* in Section 3.1.1.2.

Next we explain how *Archimedes* calculates the number of clients and the location of the client requests in Section 3.1.1.3.

Lastly we present the client and the server integrations with *Archimedes* in Sections 3.1.1.4 and 3.1.1.5 respectively.

3.1.1.1 Resolving Mechanism

Archimedes resolution mechanism uses a table called *Deployments Table* which associates every deployment with their respective instances and these instances with their address (see Figure 3.1). The *Deployments Table* is divided into a local and a remote table, where the local one stores the deployments and instances running on this node, and the remote stores the ones received from other nodes.

Archimedes evaluates a set of conditions that aim to maximize the availability of a managed application deployment and the quality of service provided to the clients of that application. These conditions comprise the ID resolving mechanism, and they operate as follows.

When a deployment is being heavily used by *Archimedes* clients, *Autonomic* marks it as needing to redirect the deployment requests to a node closer to the client (see Section 3.1.2). Therefore, the first step is to verify if this deployment is being redirected and if so redirect it to the target.

In case the request was not being redirected due to the need to load balance, *Archimedes* also keeps a set of nodes that it uses to lookup for a closer node regarding the location where the client request was originated. This set is filled by mechanisms in the tree building protocol (see Section 3.1.3). If there is a closer node in the set, it then redirects the resolve request to that node.

If *Archimedes* doesn't redirect to another node, it then searches for the requested service with the solicited ID in the local part of the *Deployments Table*, if it finds it, it then gets a random instance from that deployment and resolves the port that the client requested, to the exported port in the Docker daemon. In case there is no deployment with the given ID, it will next search for an instance and if it matches any, resolve it in the same fashion as before.

After attempting to resolve in the local table and failing, *Archimedes* will repeat the ID search but this time in the remote component of the *Deployments Table*. In case it finds it will resolve the port as described previously.

Lastly, in case all the resolving methods fail, *Archimedes* redirects to the *fallback* node, thus guaranteeing that in the worst case if this application exists, the client will be able to find it in the cloud. It is important to remember that in case a managed application exists in the system, it will at all time exist in the *fallback*.

Table 3.1: Example of *Archimedes Deployments Table* of node1.

Deployment	**Instance**	**Address**	**Type**
deployment_A	deployment_A-00001-node1	163.83.103.56:5094	local
deployment_B	deployment_B-00001-node1	163.83.103.56:4800	
deployment_A	deployment_A-00001-node14	163.83.103.70:5937	remote
	deployment_A-00001-node23	163.83.103.79:1273	
deployment_C	deployment_C-00001-node6	163.83.103.62:2384	

3.1.1.2 Table Propagation

To avoid falling back to the cloud as much as possible *Archimedes* assumes that there is an overlay that provides message broadcasts to a limited horizon and that the overlay distance (number of hops) is correlated to the physical distance between any two given nodes. The better the correlation between number of hops and geographical distance is in the overlay, the better will be the geographical locality of the deployments generated by our proposed solution (see Section 3.1.5).

Archimedes makes the most out of this property by propagating its local component of *Deployments Table* to its neighbors within a configurable horizon. This means that whenever a deployment is extended to a certain node, other nodes in its surroundings will also be able to resolve requests that point to that service, even though they are not running it.

For better comprehension, we can look at Figure 3.1 where we have a partial view of an example system. Nodes are represented as circles and clients as triangles. The blue

F node is the *fallback* node, node *A* is running a *red* deployment instance, whereas node *B* is running a *yellow* deployment instance. There are two clients *X* and *Y*, where *X* is interacting with the *red* deployment at *A* and later on will interact with *yellow*, whereas *Y* is only interacting with the *yellow* deployment at *B*. The arrows represent the reachable nodes within a 2 hop horizon from node *B* (we omit unnecessary arrows for the sake of clarity).

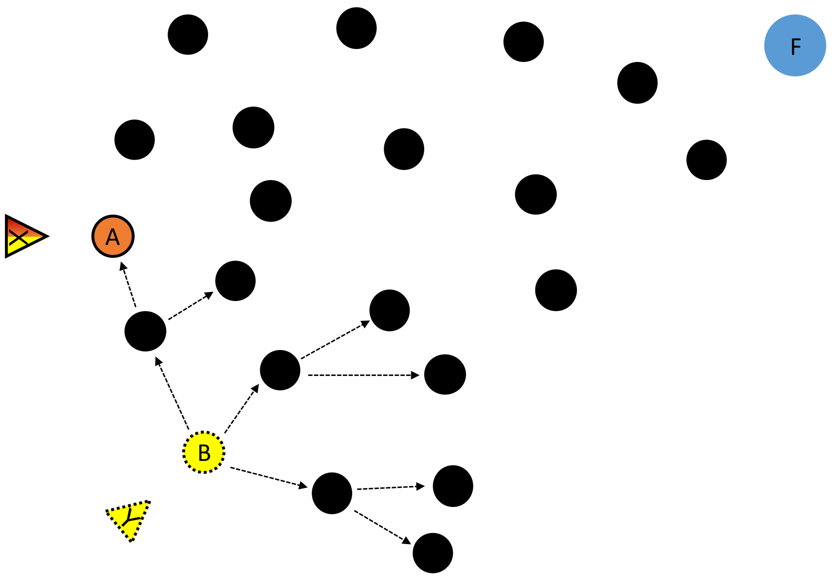


Figure 3.1: Archimedes overlay broadcast example.

Deployment	**Instance**	**Address**	**Type**
red	red-A	82.2.12.38:7283	local

(a) Node A *Deployments Table.*

Deployment	**Instance**	**Address**	**Type**
yellow	yellow-B	74.48.92.18:1284	local

(b) Node B *Deployments Table.*

Figure 3.2: Node *Deployment Tables* **before** *B* broadcast.

Deployment	**Instance**	**Address**	**Type**
red	red-A	82.2.12.38:7283	local
yellow	yellow-B	74.48.92.18:1284	remote

(a) Node A *Deployments Table.*

Deployment	**Instance**	**Address**	**Type**
yellow	yellow-B	74.48.92.18:1284	local

(b) Node B *Deployments Table.*

Figure 3.3: Node *Deployment Tables* **after** *B* broadcast.

Before the broadcast, both nodes only contain their local deployments (see Table 3.2). In a scenario where there was no local propagation, client *X* would issue a request for the *yellow* deployment, node *A* would then redirect it to fallback, and client *X* would have to be redirected down the tree until ending up on node *B*. This would greatly increase the latency to the client due to the unnecessary communication with the cloud when there is a node by its side, plus in a scenario where this was the default behavior, every client that could not find an instance in the closest node would have to contact the *fallback* node, which would incur in additional latency and bandwidth usage ultimately overloading it.

In order to decrease the chance of resorting to the *fallback* node, node *B* propagates the local component of its *Deployments Table* to the nodes within the horizon, with every node adding the *yellow* deployment to their table. Now, when client *X* decides to access the *yellow* deployment, node *A* resolves the request to the address of the instance present on node *B*, redirecting the client to that instance.

3.1.1.3 Requests Tracking

As mentioned previously, *Archimedes* is the entry point for the clients of managed applications into our system, therefore it is also responsible for tracking the amount of requests and their origin.

First, in order to keep track of how many clients are connected to *Archimedes* at a given node, we devised a strategy that registered the number of clients that issued a request in a recent time frame. This strategy is a batch based system that groups requests in a time window and stores *N* recent time windows, allowing us to not have to maintain a persistent connection to every client, which would incur in extra overhead. This also allows our system to not take immediate actions in cases such as peaks of high or low usage by taking into account multiple time windows.

This means that the system when querying the load sees a recent history and not an instant in time, thus mitigating actions of removing deployments from a node due to fewer clients in a given instant.

In addition to the batch based system that measures the amount of clients using *Archimedes*, we also devised a system that monitors the requests' origin. This system uses S2 cells to represent locations [49], since these provide a hierarchical scheme where we can characterize a location with variable precision (see Figure 3.4).

The hierarchy in S2 cells establishes that a cell at level *X* is composed by four different smaller cells at level *X+1*. Another big advantage is that every cell can be represented by a unique ID and most of its computations such as getting its children or a parent at a given level is very cheap from the point of view of computation.

Another feature that *Archimedes* needs to provide is the locations from where the requests are originating. A first approach would be to store the S2 cell that the client sends in the request, which would incur in having to track the locations for every client for every deployment. Alternatively we could store a cell at a lower level (less precise)

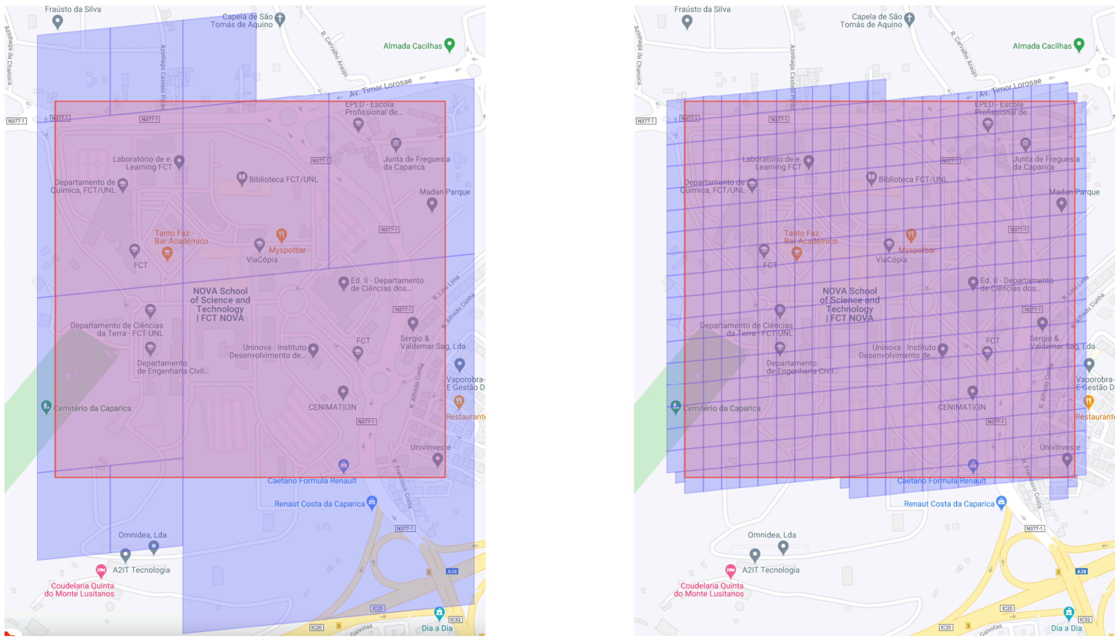

Fraústo da Silva
Capela de São Tomás de Aquino
Almada Cacilhas
Av. Timor Lorosae
EPED - Escola Profissional de...
Laboratório de e-Learning FCT
Biblioteca FCT/UNL
Junta de Freguesia da Caparica
Departamento de Química, FCT/UNL
Madan Parque
Tanto Faz - Bar Académico
Myspotbar
ViaCópia
FCT
NOVA School of Science and Technology | FCT NOVA
Ed. II - Departamento de Ciências dos...
Departamento de Ciências da Terra - FCT-UNL
Sergio & Valdemar Sag, Lda
Uninova - Instituto Desenvolvimento de...
Departamento de Engenharia Civil...
Vaporobra-E Gestão D
Cemitério da Caparica
CENIMAT|i3N
Restaurant
Univinveste
Caetano Formula Renault
Renault Costa da Caparica
Omnidea, Lda
A2IT Tecnologia
Coudelaria Quinta do Monte Lusitanos
Dia a Dia

Figure 3.4: Representation of a region using S2 cells with different accuracies.

that encompassed multiple clients. Ultimately, these alternatives pose a trade-off between either having additional precision with a lot of high level cells, or a bad precision with fewer lower level cells.

In our system we aimed to get the best of both by implementing a merge mechanism where we start by storing a lower level cell (less precise) and when the number of clients in that cell increases past a certain point, we split the cell and start considering the four cells within (with higher level). This allows us to adapt our model closer to the real world, since in high density regions it might be useful to consider smaller cells, whereas in less dense regions such additional precision would not bring significant advantages. The cells representing the clients location are called *centroids*.

Both the load of the deployments and the requests' origin are made available to other services through dissemination (with limited horizon) over the overlay (see Section 3.1.5).

3.1.1.4 Archimedes Client

When designing (and implementing) our proposal we tried to minimize the effort needed to integrate applications with our solution. On the server side, this was mainly achieved by wrapping the application's *backend* in containerized images and writing deployment configuration files, however in the client side such mechanisms cannot be employed, since we can't encapsulate the requests issued, instead we intercept them. When intercepting the request we can resolve the host in *Archimedes* and then rebuild the request with the address and port of the closest instance of the service being accessed by the client provided by *Archimedes*.

We found that the path that led to the least amount of change on the client *codebase* was to override the protocol implementations with wrapped versions of these. The wrapped versions are copies of the basis protocol, with resolving steps added when establishing connections. On protocols that are connectionless (such as HTTP) this meant resolving the host on every request, which will incur in two times as many requests as expected. This can easily be mitigated by implementing a cache. On the other hand, protocols such as WebSocket which maintain a connection throughout the interaction, the resolving step only happens once per connection when attempting to establish it.

In our use case we implemented an HTTP library that is a copy of the default one (in our case in Go), that provided the same interface but internally executed the process described. We can look at the example present in Figure 3.5, where the client is requesting a URL describing the login path for the *authentication* deployment. The middleware will then get the URL from the request, extract the deployment and the port and send a request to the *Archimedes* server that it is connected to containing these two fields. At this point *Archimedes* resolves the deployment and the port (see Section 3.1.1.1) and responds with the address and port that the instance is listening on. *Archimedes Client* now can issue the original request with the altered address and port. This interaction can be visualized in Figure 3.5.

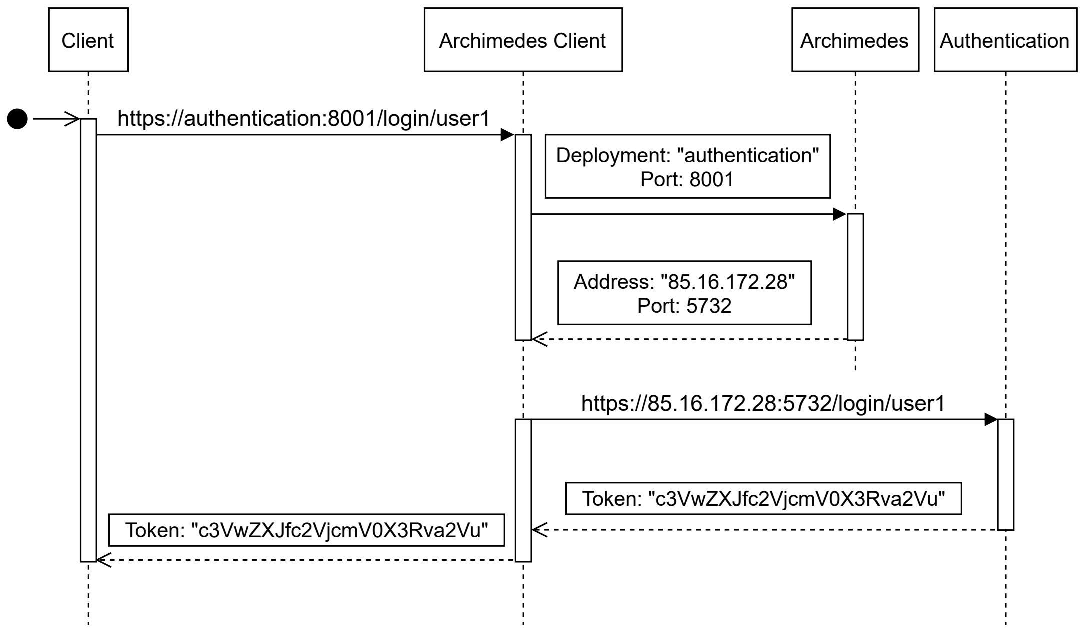

Client
Archimedes Client
Archimedes
Authentication
https://authentication:8001/login/user1
Deployment: "authentication"
Port: 8001
Address: "85.16.172.28"
Port: 5732
https://85.16.172.28:5732/login/user1
Token: "c3VwZXJfc2VjcmV0X3Rva2Vu"
Token: "c3VwZXJfc2VjcmV0X3Rva2Vu"

Figure 3.5: *Archimedes Client* resolution sequence diagram.

From here, when adapting other protocols to work with *Archimedes* we can either proceed in the same fashion and add the resolving steps into the libraries or simply edit the application client *codebase* to resolve the host before using the address.

The first option allows us to provide a drop-in replacement library that not only requires as few code modifications as possible to the application client, but also can be reused across different application clients that use the same interaction protocol. The cost of this alternative is the one of having to implement the drop-in replacement library.

The second alternative however, is more extensible since it extracts the *Archimedes* resolve step, thus separating the concerns of having to resolve the host and the communication protocol used to interact with the application. However, this last option requires more changes in the clients code by preceding every connection establishment with a resolve step and this not only has to be repeated throughout the client code but also across different client implementations since this is on a per use case basis.

3.1.1.5 Archimedes Server

Application servers also take part in integrating with *Archimedes*, more specifically in informing *Archimedes* that they are alive and that are ready to receive requests. Once again, as described in Section 3.1.1.4, we can implement this in two different ways, by reimplementing the protocol library that the server uses to listen for incoming connections, or by extracting the logic to a separate library. An extra benefit that comes with the first alternative is that when reimplementing the library, we can both implement the client and the server communication protocols with the required changes.

The way this middleware interacts with the server is by intercepting the listening action and start a separate routine that registers the deployment instance in *Archimedes*. This routine also keeps sending heartbeats to *Archimedes* periodically to assure it that it is still alive.

3.1.2 Autonomic

In order to avoid falling back to a cloud-centric model, we needed our system to evolve in a decentralized fashion. For this we devised *Autonomic*, which is the system component responsible for analyzing the environment and executing actions. This allows nodes to adapt independently according to the usage of the clients, meaning that when a managed application reports a metric (e.g. the distance from the node to the clients that is serving), *Autonomic* generates a list of alternative nodes, orders them according to the metric and lastly triggers *Deployer* to execute an action (e.g. extend the managed application to a closer node). Each of these metrics is considered a *goal*, and each managed application can specify which strategy (list of goals) wants to execute. This will lead to every node that is running the specified managed application running the strategy specified, thus collecting the metrics and executing actions independently. Each goal has four main stages: *generate*, *filter*, *order*, and *cutoff*.

The first phase is responsible for generating a set with valid nodes that fit the goal needs (such as considering only nodes which do not have a specific managed application running). Alongside the nodes, it also generates the needed criteria (e.g. distance to clients) to later evaluate their ranking.

Next, the *filter* phase discards the nodes from the *generate* phase, according to the output of the previous goal in the pipeline. By default, this phase is a set intersection between the output of the first phase and the output of the previous goal.

The *order* phase orders the nodes considered so far according to the criteria generated in the first step.

Lastly, *cutoff* will remove the candidate nodes that are not suitable. The suitability of a given node is related to the criteria that they were ordered previously.

This kind of pipeline enables for a goal to reject certain candidates that can't be chosen in the next goal, while guaranteeing that each goal has impact in the final result (see Figure 3.6).

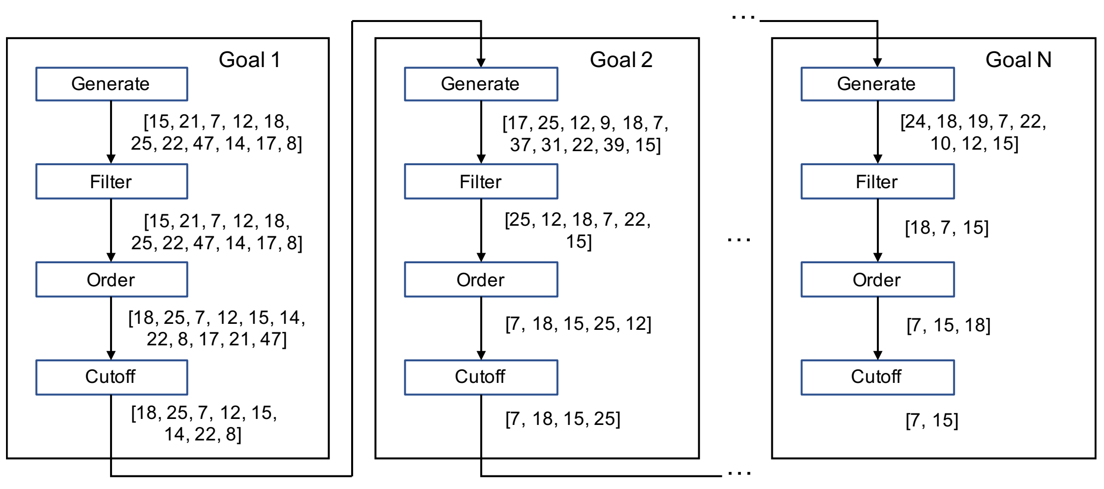


Figure 3.6: Autonomic goal pipeline.

Ideally the pipeline ends with a set of candidates that are valid for every goal in the pipeline however, there are scenarios where it is not possible to satisfy all the constraints. This can happen when there are at least two conflicting goals and one eliminates all the nodes that the other would deem valid, or when the set of nodes has been reduced to nodes that can not satisfy the minimum value of the criteria being considered (such as being too far from the client to be considered a viable alternative). The first scenario comes from an unsatisfiable set of goals provided by the user, and it will remain conflicting throughout the system life, whereas the second one can be transient and resolved automatically by the periodic changes in the overlay view, which will offer new candidates.

Whenever the pipeline reaches the previous scenarios where the set of candidates for the next goal is empty, it considers the pipeline up until this unsatisfiable goal, and therefore ignores the result. For example, assuming goal *X* returns an empty set it means that the pipeline from the first goal up until *X* is unsatisfiable and therefore goal *X+1* ignores the output from goal *X* and considers all candidates for the current goal as valid by not intersecting with the candidates from the previous goal. The main objective by creating a new pipeline and ignoring the previous goals is to avoid that two unsatisfiable goals impair the pipeline and therefore the rest of the goals as well.

Another important idea to keep in mind is that, the goals throughout the pipeline should not consider only candidates that would result in a guaranteed optimization for the goal in question. This would greatly hinder the remaining goals in the pipeline, so instead, goals should be willing to accept candidates that are worse choices than the current node by a small factor. This can greatly improve the goals further ahead since the candidates that are slightly worse for a given goal can translate in a considerable improvement in a goal later down the pipeline.

Such is the case when considering the distance of a managed application to the clients and the load distribution across multiple nodes as two goals. If the goal responsible for minimizing the distance only considers nodes that are closer to the client, it can ignore nodes that where further from the clients by a very small factor, yet have no load and would be suitable candidates for load balancing.

Upon reaching the end of the pipeline, the resulting value is passed as input to an action. The action is decided by the first goal that is not maximized. The actions contemplated in our system for now are the following:

- **EXTEND** – extend a deployment to a node.
- **MULTIPLE_EXTEND** – extend a deployment to a set of nodes.
- **REMOVE** – remove a deployment from a node.

Our system implements a default strategy which has two main goals, *Ideal Latency* and *Load Balancing* (described below). In the default strategy the pipeline starts with the ideal latency goal and finishes with the load balancing goal. Even though the default strategy is quite simple, our strategy model allows for the user to compose much longer pipelines. We will also see that a strategy this simple can be fine-tuned, more specifically in the *cutoff* phase, to achieve great results.

Next we describe in depth both goals that comprise the default strategy, *Ideal Latency* and *Load Balancing*, in Sections 3.1.2.1 and 3.1.2.2 respectively.

3.1.2.1 Ideal Latency

The *Ideal Latency* goal is responsible for minimizing the distance between the client and the deployment instance. This is achieved by extending the deployments to nodes closer to the client. Next follows a brief description on how this goal handles each phase.

Generate This phase starts by querying the overlay layer for the neighbors, their respective location, and the centroids of the clients using the deployment. In order to avoid loops, the deployment parent is removed from this list since it would not be a valid choice. At this point we could filter by only the neighbors that don't have this deployment already, but since goals are executing on different nodes independently , there would be no guarantees that the view of the deployments on each node at this point would be consistent with the view when executing the action later on.

From here, it calculates the distances from each neighbor to each centroid and the distances from the node to the centroids. Then it divides every neighbor to centroid distance by the distance from the node to that same centroid, resulting in what we call the distance factor. Assuming N as the set of neighbors, C as the set of centroids, d_n^c as the distance between neighbor n and centroid c and D^c as the distance between the current node and centroid c, we have that the distance factor of neighbor n to centroid c is described by the following equation:

$$F_n^c = \frac{d_n^c}{D^c}, n \in N, c \in C \tag{3.1}$$

This phase then ends by returning the candidates and the distance factors.

Filter In this stage we apply the default filter that ignores any candidate that was not in the output from the previous goal, however since the *Ideal Latency* goal is the first one in the pipeline for the default strategy this will mean that it will never remove any candidate.

Order After having the candidates filtered, we consider the lowest distance factor for each node and order them from lowest to highest. The result set R is described as follows:

$$R = [f_0, f_1, ..., f_n] : f_x = \min F_x^c, c \in C, n \in N \tag{3.2}$$

Cutoff The last phase is responsible for discarding candidates. In the *Ideal Latency* goal, all the candidates with a minimum distance factor bigger than 1.2 are discarded. It would be expected that the cutoff point would be a value smaller than 1.0, since any value equal or bigger than this would mean extending the deployment to a node further away from the centroid than the current one, but as we have explained previously a goal should not be selfish.

Upon having the resulting cutoff, *Autonomic* will issue a *MULTIPLE_EXTEND* action that will issue a request to *Deployer* with the nodes it should extend to.

3.1.2.2 Load Balancing

Another important goal in our autonomic system is *Load Balancing*. This goal is responsible for scaling deployments horizontally to avoid the same saturating pattern that would occur in a cloud based deployment. Next we describe how we achieve this in our pipeline model.

Generate In this phase we first query the overlay layer for which nodes are in our vicinity, the respective load for the deployment (e.g. number of clients) on each of those nodes, and the load on the running node. This set includes all the vicinity wether they contain or not the deployment already. How we calculate the load is described in detail in Section 3.1.1.3. At this point we have our initial domain that will be used in the next phase.

Filter For *Load Balancing* we also use the default filter that will ignore all the candidates that are not on the shortlist resulting of the previous goal.

Order This phase orders the candidates by load in ascending order.

Cutoff The final stage will ignore all the candidates that have a load (number of clients) above a given threshold (the default threshold is 50 clients).

Upon having the resulting cutoff, this goal will verify if the current node is overloaded (e.g. is serving too many clients) and if so increment the number of cycles that this node has been overloaded. Whenever this counter goes above a given threshold, either a *REDIRECT* action is issued to redirect clients to the node with the lowest load, or instead it issues an *EXTEND* action extending the deployment to an alternative that can help by handling some clients. After extending the deployment, the node will then have an alternative deployment to where it can redirect clients.

The number of cycles that are needed to consider a node overloaded is equal to one and a half times the time a client is considered connected to an *Archimedes* (*Archimedes* cache duration). This guarantees that most users will be counted for, since after the cache entry expires, clients send a request to *Archimedes* again, thus being accounted for.

This goal is also responsible for verifying if the current deployment is now stale, this meaning that no clients have accessed it. In order for a deployment to be eligible for removal, much like when overloaded, it has to go through a number of cycles of the autonomic goal optimizer to be considered as stale. This requirement serves to avoid rushed (and potentially incorrect) decisions. A deployment can only be removed from a node, if such node has no children for the given deployment in the deployment tree (see Section 3.1.3) and the number of cycles without any load is above a given threshold. By default, this threshold is equal to one and a half times the timeout to reset to the fallback in the *Archimedes* client, as to guarantee that if any client was using that deployment it will reset to the fallback and be redirected there, thus making the load bigger than zero and invalidate the removal of such deployment. Whenever a deployment needs to be removed, a *REMOVE* action is issued by *Autonomic* to *Deployer*.

3.1.3 Deployer

Deployer is the entry point for our system users, where one can register the applications that will be running and managed by the system. It is the central component that triggers the remaining microservices to either instantiate a deployment (*Scheduler*) or start monitoring the deployment and optimizing its goals (*Autonomic*). An important piece of this system is the deployment tree building protocol that is built starting at the *fallback* node (i.e., cloud) till the closest node to the clients. This tree is built according to the tree building protocol, which should evolve in a decentralized fashion as to not overload the cloud and furthermore, it should also have a fault tolerance mechanism that allows for some nodes to fail simultaneously without disrupting the operation of the managed application (s).

The tree that spans from the cloud to the client's closest node starts as a stationary managed application in the *fallback* node (*F*). After the application is used by the clients, *Autonomic* sends a request to *Deployer* to extend the application to a specific node, for now let's call this node *X*. The *Deployer* at *F* will send a request to register the application in the *Deployer* at *X*. *X*'s *Deployer* will add *F* as its parent and subsequently *F*'s *Deployer* will add *X* as a child. Assuming *X* is not the closest node to the clients, the *Autonomic* present at this node will then issue a request to extend the application to a closer node that we will call *Y* (similar to when *F* extended to *X*). Now, *Y*'s *Deployer* will add *X* as his parent and *F* as his grandparent. *X*'s *Deployer* will also add *Y* as his child. The hierarchy levels known by any given node, are at most two upwards (parent and grandparent), and one downwards (children). Figure 3.7 shows a visualization of the hierarchical relations described.

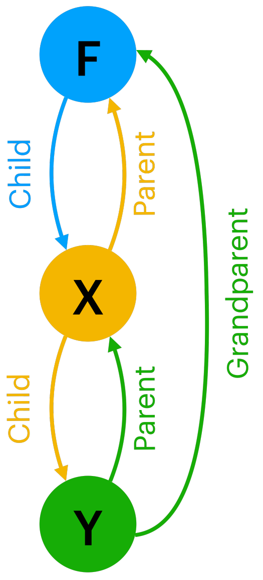


Figure 3.7: Example of all hierarchial relations.

This hierarchical relationships are particularly useful for monitoring nodes in the fault tolerance mechanism, as we will see in the next section. Furthermore, in Section 3.1.3.2, we will explain how the tree structure enables a deployment propagation mechanism that allows *Archimedes* to quickly redirect the client to its closest node.

3.1.3.1 Fault Tolerance

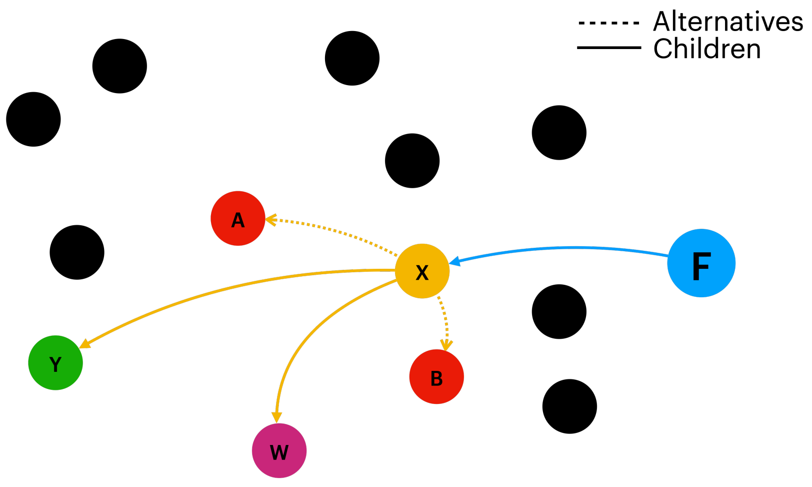
Our fault tolerance mechanism has, as main goals, simplicity, fast reaction time, and small number of messages (i.e., low overhead). This is mainly due to the context we are in, where edge nodes are not expected to have an availability as high as the cloud, yet they should not be as unstable as end clients devices, and therefore the probability of failure is not high. Furthermore, in this scenario, acting fast allows the users to maintain a similar QoS and not having to fallback to the cloud overloading the *fallback* node.

The way our protocol works is that every parent keeps sending periodic heartbeats to their children indicating that they are alive. In case a parent does not send a heartbeat for a given amount of time, his children will then tell their grandparent that his son is dead, so in our previous scenario with F, X, and Y, in case X stopped sending heartbeats, Y would report it as being down to F. After the grandchildren report that their parent is dead, it is the grandparent's responsibility to find a new suitable candidate to replace his lost child. For this, every node sends its alternatives to their children as to, in case of a failure, leave the orphan nodes with a subset of alternative parents that they can propose to their grandparent. Alongside the alternatives, the children also send the client centroids that they are serving as to allow the grandparents to target a specific location in case no alternative is alive or the children dies amidst the repairing process.

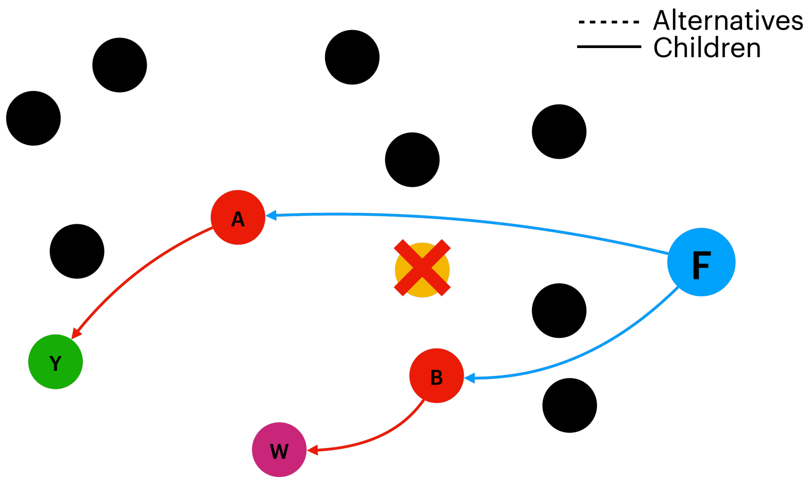
In Figure 3.8(a) we can see that X has A and B as its alternatives. These alternatives are being sent periodically alongside with the heartbeats to all node X's children, which in this case are Y and W. In a scenario where X goes down (see Figure 3.8(b)), both Y and W send a warning to their grandparent, F, containing the alternatives and the clients' location that the *Archimedes* in the respective child was serving. Even though in this scenario F extended to two different alternatives, it is not mandatory to extend to all alternatives, in fact, in this case, F receives two requests, one from each child, to which he will then extend the deployment to the closest alternative to the centroids (this assumes that the centroids were very close to X and W), so it happens that the alternatives closest to X and Y are different.

3.1.3.2 Deployment Propagation

As mentioned previously in Section 3.1.1.1, in the second step of request resolution, we left unexplained how *Archimedes* searches for a node closer to the origin of the request. This is where the tree protocol is particularly useful. When we add a deployment in a node, it will then broadcast this update N levels up the tree, informing the nodes of the deployment alongside with his location. When a *Deployer* receives this information it then adds the deployment to the local *Archimedes* instance and sends the information to



(a) Before *X* failing.



(b) After *X* failing.

Figure 3.8: Example nodes with child relations and node *X* alternatives.

its parent. The number of levels *N* is configurable, and it does not have to be the same as the number of hierarchy levels in the tree, with this meaning that even though we only know the parent and the grandparent (two levels upwards) regarding hierarchy, we can propagate deployment updates further up the tree to decrease the number of jumps it takes for the client to reach the closest node.

In order to better understand how *Deployer* interacts with the other microservices, we will now go through the lifetime of a deployment from the viewpoint of the *Deployer* component of out solution.

A deployment first starts by being registered in the *Deployer* present in the *fallback* node. Each *Deployer* has a *hierarchy table* that contains the grandparent, parent, children and the deployment information. When registering a new deployment, this will be added to the *hierarchy table*, it will also register it in *Autonomic* to start tracking the clients load and location, and lastly in *Scheduler* to deploy the service. It is important to know that at this point the service is not yet accessible to the clients since it is not registered in *Archimedes*. In fact, it will not be the *Deployer* component to register it in *Archimedes*, since the *Deployer* does not know when the service is ready, this responsibility is left to *Scheduler* since it is the one starting it.

Eventually, after the service has been added to *Archimedes* and clients start using it, *Autonomic* will analyze the centroids and chose the best node to extend the deployment of that managed application service (i.e., create a new instance). When extending, *Autonomic* issues a request to *Deployer* telling it to extend to the best node, which for now we will call node *X*. The *Deployer* present in the *fallback* node will then issue a request to node *X* to register the deployment, indicating that he is the parent to that deployment. Node *X* adds this deployment to his *hierarchy table* indicating that the parent is the *fallback* node.

3.1.4 Scheduler

Scheduler is responsible for interacting with the *Docker* daemon to start and stop instance containers. In order to start the containers the *Scheduler* has to resolve all the environment variables setup in the deployment configuration file. Besides the environment variables specified by the deployment, *Scheduler* also injects default environment variables to the instance container. Next follows the default environment variables:

DEPLOYMENT_ID Unique identifier for the deployment that the instance belongs to.

INSTANCE_ID Unique identifier for the instance.

FALLBACK_URL *Fallback* node IP.

NODE_IP Node IP that *Scheduler* is running on.

PORTS String representing the port translation from the container ports to the exposed ports in the node. Each port translation is represented by "**container_port->external_port**" and the string is a group of port translations separated by a semi colon. Example: "**80->3485;22->2874**".

REPLICA_NUM Number of the instance replica for a given deployment in this node.

LOCATION S2 cell ID representing the node location.

NODE_ID *Hostname* of the node *Scheduler* is running on.

Lastly, *Scheduler* is also responsible for translating the ports inside the container to node ports, thus allowing different instance containers to listen on the same port internally since the exposed ports will be different.

3.1.5 Overlay

Besides the four main components, our system also has an overlay component that is responsible for providing a partial view of the system, detecting changes in the membership, implementing a metrics dissemination mechanism and basic broadcast primitives to a limited horizon.

This overlay would ideally correlate the geographical distance between two nodes with the number of hops between them, therefore biasing the partial view of the system that a node sees to be filled with nodes that are geographically close.

Nevertheless, due to this overlay being out of this thesis scope, we used one that explores latency biased overlays, where the partial view contains nodes to which the current node has low latencies. Even though latency does not have a direct correlation with the geographical distance (particularly in small distances) due to other factors such as number of devices throughout the routing path (routers, switches, etc.), it still plays a key factor and therefore it can be considered as a viable alternative. In some cases, the latency biased view provided outliers (due to the aforementioned imperfect relation between distance and latency) and in order to minimize the effect of these outliers and increase the chance of reaching nodes that are effectively close to desired geographical location, we consider the nodes within a limited horizon bigger than one hop. This increases the number of nodes in the view, and therefore the probability of finding better nodes.

It is important to notice that this is not achieved without a cost since in order to consider a bigger horizon, we must send discovery messages using the broadcast primitives with a limited horizon over the overlay (i.e., messages are disseminated with a maximum time to live (TTL)).

The partial view and the reactive changes to it, are especially useful to the operation of the *Deployer*, where the fault tolerance mechanism depends on a node knowing a set of replacement nodes that can take its place when it fails (see Section 3.1.3.1). In this particular case, having nodes that are close by is a plus and not so much a critical property since in the worst case where we select an alternative that is far away, we want to replace the faulty node fast and let the system later converge to an optimal state.

Ideally, the overlay view and the reactive changes would also fit the needs for *Archimedes* table propagation, but since in our case the overlay was not in the scope of this thesis, the one used was not geographical aware and therefore, in order to guarantee that the *Deployments Table* reaches nodes geographically close to the current one, *Archimedes* uses the broadcast primitives to do a flood with a limited horizon.

3.2 Clients

There are two kinds of clients in our system, applications that develop containerized images and deploy (and manage) them leveraging our proposed solution (see Section 3.2.1) and the end clients that use the managed applications (see Section 3.2.2). Both of these clients and their interactions will be described in the following sections.

3.2.1 Applications

Applications are described in a YAML configuration file heavily inspired in the configuration files used in Kubernetes to minimize the effort in translating a deployment from Kubernetes to our solution:

- **replicas**: Number of replicas to deploy on a given node for this deployment.
- **deploymentName**: Name of the deployment that will be used to identify it.
- **containers**: Container to deploy for this service:
  - **image**: Repository for the container image.
  - **command**: Command to run when launching the image.
  - **ports**: List of ports to expose.
  - **env**: Key-Value list of environment variables.
- **static**: Determines if this is a static deployment or not.
- **depthFactor**: Number from 0 to 1 to determine how close to the clients should the deployment be extended (0 meaning cloud-only, 1 as close to the users as possible).

Next, follows an example that describes the *authentication* microservice of the benchmark PouchBeasts:

```
replicas: 1
deploymentName: authentication
containers:
  - image: docker.io/brunoanjos/authentication:latest
```

```
    command: [ "sh", "-c", "./executab{}le -l -d -a" ]
    ports:
      - containerPort: 8001
    env:
      - name: MONGODB_URL
        value: "mongodb://novapokemon-usersdb:27017"
      - name: TRAINERS_URL
        value: "trainers:8009"
static: False
depthFactor: 1
```

Listing 3.1: Example deployment configuration file

3.2.2 End Users

End clients interact with our system through the *Archimedes Client* (see Section 3.1.1.4). As mentioned previously, this middleware has many ways of being integrated in the client application (as drop-in replacement for the protocol used or as a standalone HTTP library) but whichever solutions is implemented, its sole requirement is the URL for the *fallback* node.

Furthermore, the application clients have to provide their location and keep it updated so as to improve the client's QoS through extending the deployments to closer nodes.

3.3 Implementation

Our system is implemented using the Go [18] programming language. This was mainly a choice motivated by personal preference, but a valid argument would also be the development speed and the runtime performance that this language offers. All the services were implemented using HTTP backends. Docker was used as the container runtime due to its simplicity, proven track record and the good SDK made available by the team for the Go language.

The framework is implemented in 12500 lines of code.

3.4 Summary

In this chapter we presented our proposal for an edge aware deployment framework that aims to make the most out of the colocation of edge nodes and the managed application clients. We presented the four main components of our solution (*Archimedes*, *Autonomic*, *Deployer*, and *Scheduler*) and their respective role in the framework. The integration of the framework with the managed application clients was also described in detail alongside the different possibilities of it can be done according to the nature of different applications and the communication protocols used.

Next, in Chapter 4 we will present a benchmark that can measure the performance of different deployment scenarios, namely a cloud and an edge scenario and allow us to draw conclusions related to the advantages and disadvantages of both scenarios.

Chapter 4

PouchBeasts Benchmark

In this chapter we will present PouchBeasts, a benchmark that aims to emulate the different interaction patterns between a client and a service, while providing a network model that can apply location aware delays to fit the needs of both cloud and cloud-edge scenarios. First, in Section 4.1 we present the microservices that compose the benchmark and the client. Next, in Section 4.2 we address the implementation details of the benchmark, and finally, in Section 4.3 we discuss the network model implemented in the benchmark that allows to emulate both a cloud scenario and a cloud-edge scenario, imposing delays with location awareness regarding the source and the target.

4.1 Design

In order to understand how the proposed solution in Section 3 performs when compared to other alternatives we need a benchmark that replicates a real life scenario where an application backend is deployed in the cloud and a client issues requests to such service. At first sight, any application with a client issuing requests and measuring the latency would seem as a good fit since, in our system, this latency would be expected to decrease when extending the deployment to nodes in close vicinity of end clients. As we will see, an important criterion to take into account when choosing the benchmark is the state handled by the application throughout its lifetime. Furthermore, applications can be considered either stateless or stateful. Stateless applications are services that keep no state between different requests, whereas stateful applications keep state that can be mutated by the requests issued. In order to minimize the application footprint and increase its portability, developers usually offload as much state as possible to a database. This kind of applications are highly scalable and can be deployed closer to the client with two different approaches: either deploy the application closer to the client with a cache and aim for a high cache-hit rate to improve latency, or instead apply a more complex partition scheme of the data in the central database and replicate it to closer nodes. Since data replication and partitioning schemes are not part of the scope of this work, this would imply either using stateless applications and ignore the data management component, or instead use stateful applications. In order to keep a fair comparison between both scenarios, we decided to choose stateful applications as the benchmark for both scenarios.

In stateful applications the state can be shared between multiple clients, in which case, the time that it takes to reflect the changes on all clients can be crucial and a determinant factor for the user experience. An example of such applications are mobile video games where two or more players are competing against each other in real time. A player that has a higher latency to the server can be at a disadvantage since it will see the action at a later time and therefore react later than the other player. Even though this latency can be very small and in some cases negligible, in the cloud computing environment such is not the case due to the client's distance to the server. Furthermore, high usage peaks can easily saturate the connections and the computing resources, increasing the latency above the expected and ultimately rendering the application unusable. Such was the case with applications such as Pokémon Go where the service was several times down due to high usage peaks [15] [21] [45].

These applications usually have a locality pattern associated, since the user is most likely to play with other users that are nearby. In order to replicate the real life scenario of such applications we devised a benchmark that is heavily inspired on Pokémon Go called PouchBeasts. PouchBeasts is a mock application that is composed by nine microservices and a client that interacts with each other. In the following, we provide a description of all these components:

Authentication is responsible for registering and authenticating users. This microservice is the first microservice that the client contacts when entering the system. First, the client registers itself with a username and a password. It then issues a login request to which the server replies with an *authentication token* that the client can send to the remaining microservices to verify its identity.

Authentication also has a database used to store the user credentials.

Battles is the microservice that implements battling capabilities between two different trainers (i.e., end users). Users can start a battle by either challenging a specific user, similar to how a user would battle his friend, or by joining the matchmaking queue and either wait in a new lobby for another player, or join an existing queue lobby.

In order to challenge someone, the user issues a request to the *Battles* service indicating which user it wants to challenge, upon which the *Battles* service sends a message to the *Notifications* service indicating the user it wants to send the notification to and to which lobby the notified user should join. At this point when the challenged user receives the notification it may either accept or reject the challenge. The user either accepts or rejects the challenge by issuing a request to either the join lobby or the reject endpoint, respectively. The latter case leads the *Battles* service to send a *REJECT* message to the waiting user, thus informing him that the other user has rejected the challenge and that no battle will occur.

When joining a battle the server verifies the *authentication token* provided by the user. Alongside the *authentication token* the user also provides an *items token* containing all the items that the user has bought, a *stats token* with information on the trainer such as the experience points, and also a *pokemon token* for each pokemon that the user has. All these tokens are verified with the trainers microservice for their validity.

Upon both users joining the lobby, the battle server sends a *START* message to both users informing that both users have joined the lobby and that the battle can start. Both players then select a pokemon to use. The users then issue periodic actions such as *ATTACK* or *DEFEND*. If one player attacks the opponent pokemon and it is successful in dealing damage, the server replies with an *UPDATE POKEMON* message containing the updated pokemon info, whereas if the pokemon deals no damage due to the opponent currently being defending, the server sends a *STATUS* message indicating that the opponent defended.

Besides these two actions clients can also issue are *USE ITEM* and *SELECT POKEMON*. When using an item if the item is valid, the server will reply with a *REMOVE ITEM* message to the user who issued the original message indicating that the item was used and an *UPDATE POKEMON* message to both users with the updated status of the pokemon. The user may also change pokemon during the battle by sending a *SELECT POKEMON* message containing the desired pokemon to which the server replies with an *UPDATE POKEMON* message updating the pokemon in the battle. This message is also mandatory whenever the current pokemon has no health points and the user still has elligible pokemons for the battle. These four messages (*ATTACK*, *DEFEND*, *USE ITEM* and *SELECT POKEMON*) make the core communication of the interaction.

Whenever a user has no more elligible pokemons, the other user is declared the winner and the results of the battle are committed to the trainers service. The trainers service then issues new tokens with updated trainer stats, pokemon stats and items for each user, which are then sent to the respective user through a *SET TOKEN* message.

Gyms is a service that handles pokemon gyms spread throughout the world, with which the users can interact. The players interact with gyms by forming a raid party that aims to defeat the boss present in the gym. This service implements a player versus environment (PvE) mechanism. Like *Battles*, *Gyms* is a microservice where the quality of service experienced by the client is highly influenced by the latency, furthermore in this particular case *Gyms* implements a many-to-one battling system, and therefore the amount of messages exchanged throughout the raid is much higher when comparing with *Battles*, and therefore the service QoS can deteriorate rapidly if too many users are connected.

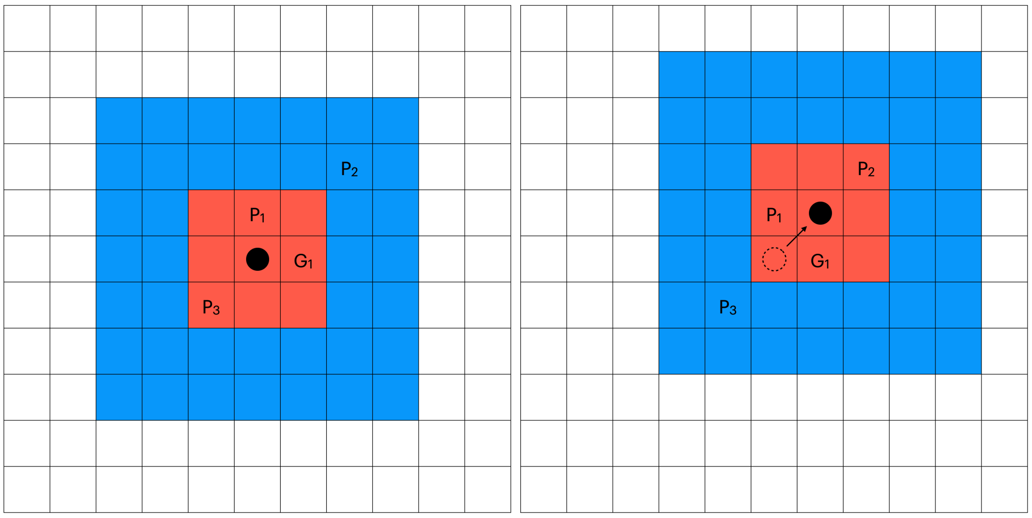
A user starts by creating a raid for a given gym. This raid lobby remains open for a specified amount of time to let other players join the ongoing raid. The lobby is then closed and the raid starts by sending a *START* message to all the participants. From this point on the battle between the boss and the participants is very similar to the player vs player battles already described, with the small difference that the boss is controlled by the server.

The *Gyms* service has a database used to store the existing gyms and their respective server. The server first starts by reading the gyms from a file to the database. This is particularly useful when there are more than one server, so the first loads the gyms from a configuration file to the database and from here on out all servers read the gyms from the database and start serving them for raids. Gyms is also responsible for adding the gyms to the location microservice that employs a filter and only shows gyms to the users according to their position and proximity to the gym.

Location is a key component of our benchmark that is responsible for keeping track of where the players are located and according to that location sending gyms and wild pokemons in the vicinity. As mentioned before gyms are added to the *Location* service by the *Gyms* service.

Users location is represented by the S2 Cells [49] within a given radius from the user position. The user cells are grouped into the entry cells and the exit cells. The entry cells delimit the area where a cell and its content (pokemons and gyms) are considered accessible, whereas the exit cells are a buffer that hold cells that were once in the entry boundary and have information that it is not worth to lose yet. In Figure 4.1 we can see an example of a user (black dot), its entry cells (red), and exit cells (blue). When the user receives the cells for the first time (see Figure 4.1(a)), it can see P_1, G_1, and P_3 in its vicinity, after moving (see Figure 4.1(b)) P_3 is no longer visible, but the information stored about this pokemon will not be lost until P_3 leaves the exit boundary (blue cells). This mechanism creates a buffer of cells that the user recently left and therefore decreases the information transferred between the server and the client.

The interaction between users and the *Location* service starts with a request to get the respective server for their current position being issued by the user. The user then connects to that location server and starts to send *UPDATE LOCATION* messages periodically to which the server replies with a *SERVERS* message, indicating which servers the user should be connected to (this takes into account the user



(a) User entry and exit cells (b) User entry and exit cells after moving

Figure 4.1: S2 cells entry and exit regions.

movement and tells the user to migrate from one server to another), a *CELLS PER SERVER* message that contains the cells that the user is in, a *GYMS IN VICINITY* message containing the gyms present in the user cells and lastly a *POKEMON IN VICINITY* message containing the pokemons present in the user cells.

To allow this service to scale, each server is responsible for a fraction of the world. This allows to dinamically adjust the number of clients connected to a given server by assigning a different amount of cells according to their popularity. The different servers synchronize the cells that they are responsible for through a database. This database is consulted when the user issues the first request asking which server it should connect to, where the server fetches the information from the database and answers with the respective server for the user's S2 cell.

Another important edge case to take into account is that a user might have to connect to multiple servers at once since it may be in a border zone between regions managed by different servers. For this purpose the server may also send a *DISCONNECT* message telling the user to disconnect since there are no remaining cells being managed by that specific server. In order to minimize unnecessary computation, when the user receives the *CELLS PER SERVER* response from the contact server, instead of sending an *UPDATE LOCATION* to the remaining servers that is connected to, it sends a *UPDATE LOCATION WITH CELLS* containing the precomputed cells and therefore saving the other location servers from computing the surrounding area around the user.

Microtransactions is, as the name implies, a service responsible for the game microtransactions, more specifically purchasing in-game currency with real world currencies.

This service acts like an intermediary between the game and real life APIs such as banks or digital payment websites. When testing this service only acts as a mockup, since no bank system is actually contacted.

This service also has a database associated that records every transaction performed.

Notifications is the microservice responsible for notifying the users when they are invited for either a battle or a trade. Upon starting, all the users connect to the *Notifications* service and wait for messages in the background. Like the users, both the *Battles* and the *Trades* service open a connection to the *Notifications* service to use when a user challenges another user. When this happens the respective service tags the notification with the name of the challenger and the address of the server that the lobby is in. The server identifies where the lobby is since both services (*Battles* and *Trades*) are stateful and when scaling to multiple instances of these services, identifying the one where the lobby was created is crucial.

Store offers a collection of items that can be bought with coins. A user can get more coins through the *Microtransactions* service.

Trades implements a trading mechanism that allows users to exchange items. The service is very similar to battles where it implements a lobby system where a user can challenge another one, the other user is notified through the *Notifications* service with the respective server and lobby, and then joins the lobby to start the trade.

Like *Battles* the challenged user can reject the challenge to which the *Trades* service will send a *REJECT* message to the waiting user. The user can also accept the challenge by joining the lobby. After both users are in the lobby the trade starts by sending a *START* message to both users. From this point on the users send *TRADE* messages with an item that they wish to trade, adding it to their trade pool. After each *TRADE* message the *Trades* service sends an *UPDATE* message containing the current state of both players trade pools and their accepted status.

When a user has finished adding the items he whishes to trade, he sends an *ACCEPT* message, thus closing his pool and updating his status to accepted. After both users have accepted, the trade is considered finished. At this point the *Trades* service commits the changes to the *Trainers* service, requests new *items token* and sends a *SET TOKEN* message to each user with their new *items token*.

A key difference between the *Trades* service and the *Battles* is that trades are short lived interactions that take, on average, much less time than battles.

Trainers is the microservice responsible for storing the users in-game information, such as their pokemons, items and stats. It is also responsible for validating and generating the tokens associated with these (*authentication token*, *items token*, *stats token*, and *pokemon token*). This service is used by *Authentication* to create the trainer when a new user registers, by *Battles* and *Gyms* to verify and update their items, pokemons and stats, by *Location* to add new pokemons when the user catches a wild one (i.e., a pokemon without an owner spawned by a server close to the user location), by *Microtransactions* to verify and update the trainer stats, by *Store* to add items and update the trainer stats, and by *Trades* to verify *items token* and add/remove items.

Since this service is used by so many other services, it is a great candidate to be used to model a system bottleneck and to hinder the performance of the overall user experience indirectly, allowing to measure the impact of different management strategies and solutions.

Client performs actions continuously that interact with the microservices previously described. These actions are challenging another player for a battle (*CB*), queue to battle another player (*QB*), make a transaction (*MT*), invite another player to trade (*T*), buy an item at the store (*S*), catch a wild pokemon (*CP*) and lastly raid a gym (*R*). In order to control which action the client executes there is a probabilities matrix that given the last action executed encodes the probabilities for the next action. In Table 4.1 we can see an example of such matrix, and the way it works is if a client just challenged someone for a battle (*CB*) the probabilities for the next operation are the ones present in the first row.

Table 4.1: Example of a probabilities matrix for player actions.

	CB	**QB**	**MT**	**T**	**S**	**CP**	**R**
CB	0.05	0.10	0.10	0.20	0.10	0.30	0.15
QB	0.15	0.05	0.10	0.15	0.10	0.30	0.15
MT	0.15	0.10	0.05	0.15	0.10	0.30	0.15
T	0.20	0.10	0.10	0.05	0.10	0.30	0.15
S	0.15	0.10	0.10	0.15	0.05	0.30	0.15
CP	0.20	0.10	0.10	0.20	0.10	0.15	0.15
R	0.15	0.10	0.10	0.15	0.15	0.30	0.05

This allows to define more complex usage patterns for the clients, similar to the real world applications usage.

With these nine microservices, PouchBeasts becomes a rather complex application, which is a consequence of implementing a real world like application very close to what would exist in production.

4.2 Implementation

PouchBeasts is fully implemented in Go [18] with the services APIs being available through HTTP. For the interactions that need server side pushing (such as notifications) or bi-directional communication patterns (such as battles, trades and location), we used

Table 4.2: Example of latencies between regions.

Regions	**A**	**B**
A	$15ms$	$50ms$
B	$55ms$	$10ms$

WebSockets. In order to provide an easy way of deploying the benchmark, all the services are containerized and, as we will see in Section 5.1.2, we also provide Kubernetes deployment files for the cloud scenario.

The benchmark is implemented in 18000 lines of code.

4.3 Network Model

PouchBeasts provides a network model that can simulate latencies on a coarse grained region basis. This model emulates what kind of latency a client in a given region should experience when contacting a node on another region, e.g. a user in Western Europe communicating with a server in Central Europe should have a higher latency than a user located in the same server region. This model is very coarse grained, and it does not model the latency continuously, so two users in the same region will have the same latency even though one might be at the edge of the region and one in the central point. Nevertheless, this model is accurate on average since, the latency from the closest user is being rounded up, whereas the latency from the furthest is being rounded down.

This model is achieved by tagging each request with a location tag (S2 Cell) and the closest node. Communication between services is not delayed since in a cloud scenario the latency is negligible and in a cloud-edge scenario the latency between the nodes should be handled on a system level and not by the benchmark. The client delay has two main factors: the delay from the client to the closest node and the delay from the closest node to the target node. The delay from the client to the closest node is always present whereas the delay from the closest node to the target node only exists when these two differ. Both of these delays are calculated leveraging the regions model previously presented. For a clearer comprehension, we present an example. Table 4.2 presents the latencies between different regions. If we have a client in region A communicating with a server in region B, the latency applied will be from region A to region A ($15ms$) and then from region A to region B ($50ms$), thus totaling $65ms$.

The entry latency is particularly important since the latencies used in PouchBeasts are between datacenters as we will see next, and the latency from a client to a datacenter should not be equivalent to the latency between two datacenters. This is even more pronounced when talking about mobile clients, which is the case for PouchBeasts clients, since there is an additional cost when an end device contacts the entry point of the network and from there on the request is routed to the target node.

As stated previously, our latency measurements for our regions model are based on inter region datacenter measurements, more specifically, from the CloudPing [9] project that extracts information from Amazon's AWS regions.

4.4 Summary

Throughout this chapter we presented the design, implementation and the network model that comprise our benchmark. We also discussed how this benchmark allows modelling different usage patterns of real life applications across its multiple services. Next, in Chapter 5 we will evaluate the deployment of the proposed benchmark in the framework proposed in Chapter 3.

Chapter 5

Evaluation

This chapter reports on the experiments ran to analyze the performance of both the basic cloud only scenario and our solution in a cloud-edge scenario. First, in Section 5.1 we will describe the setups of both scenarios, how they were emulated and what technologies were used. Then, in Section 5.2 we will present and analyze the experiments conducted to validate and evaluate both scenarios, and lastly we will discuss the results obtained in Section 5.3.

All the experiments presented in this chapter were conducted in the Grid´5000 [4] testbed, more specifically with 10 nodes in the *gros* cluster, each one with an Intel Xeon Gold 5220 CPU with 18 cores, and 96 GiB of RAM.

5.1 Setup

In this section we will explain the experimental setups used to conduct the experiments in the cloud (Section 5.1.1) and cloud-edge scenarios (Section 5.1.2). We will also detail how PouchBeasts is deployed in both solutions.

5.1.1 Cloud Scenario

Since our solution is an innovative approach on cloud deployments, there are no direct alternatives that offer a decentralized framework to orchestrate deployments. Therefore, the closest alternatives are the cloud deployment frameworks that are extensively used in production such as Kubernetes [33], OpenShift [47] or Nomad [25].

The setup for the cloud-only deployments is a multi-node Kubernetes cluster, where each node is running a set of pods. The Kubernetes Container Network Interface (CNI) takes care of creating routes to make communication between pods in different nodes possible. This structure is depicted in Figure 5.1. In this scenario, each of the nine

PouchBeasts services will be running in at least one pod (a service can have more than one replicas). These pods will be spread throughout the cluster nodes except one, which will be reserved for running the clients.

Furthermore, in order to emulate a real life cloud scenario, this setup includes an ingress that serves as a *proxy* for the clients trying to access the services deployed. Such ingress features a load balancer that can distribute the load for a given service between different pods. In our experimental setup we use Voyager [59] as our Kubernetes ingress controller which internally uses HAProxy [24] as its proxy implementation. We chose this ingress controller since it supports the Kubernetes bandwidth plugin and therefore allows us to limit bandwidth. Furthermore, HAProxy also supports the WebSockets protocol used throughout PouchBeasts. Figure 5.2 depicts how clients and servers are linked.

In cloud datacenters, the networking between machines provides high bandwidth and low latency communication between servers, which make the delay between nodes negligible. For this reason we only limit the bandwidth between the clients and the servers at the ingress, and leave the communication between pods limited only by hardware limitations. This configuration is installed through Kubernetes bandwidth plugin which internally uses Linux's Traffic Control (TC) to apply inbound and outbound rules. Moreover, in order to simulate a real scenario, with the help of TC, bandwidth limits are applied in outbound and inbound interfaces, so a limit of 20Mbit applied in the Ingress translates into limiting both the inbound and outbound bandwidth at 10Mbit.

5.1.2 Cloud-Edge Scenario

Given that our solution is designed to be deployed at the edge of the network, the experimental setup greatly defers from the standard cloud deployments since there is a need to emulate nodes outside the cloud. For this, we need to add delays in the communication between nodes, limit the throughput of edge nodes, add location awareness, and isolate applications running in different nodes. In cloud deployments, the latter concerns are usually overlooked since the delay between nodes is negligible and the throughput is very high due to the high quality infrastructure used. Furthermore, all the nodes in a cloud deployment are colocated in the same datacenter, and therefore, location awareness is not useful.

Since the number of physical nodes that we have access to is considerably lower than the number of nodes that we want to simulate, we implemented a concept of *virtual nodes* using Docker [11]. This enables us to simulate multiple *virtual nodes* in a single physical machine. Each *virtual node* runs our solution stack (*Archimedes*, *Autonomic*, *Deployer*, and *Scheduler*) and the overlay manager (see Figure 5.3).

Considering our solution uses Docker to deploy the applications, this means that we have to have access to a Docker daemon inside the *virtual node*. For this, either we gave access to the outer Docker daemon to the stack running inside the *virtual node* or ran a separate Docker daemon inside each *virtual node*. Since sharing a single Docker

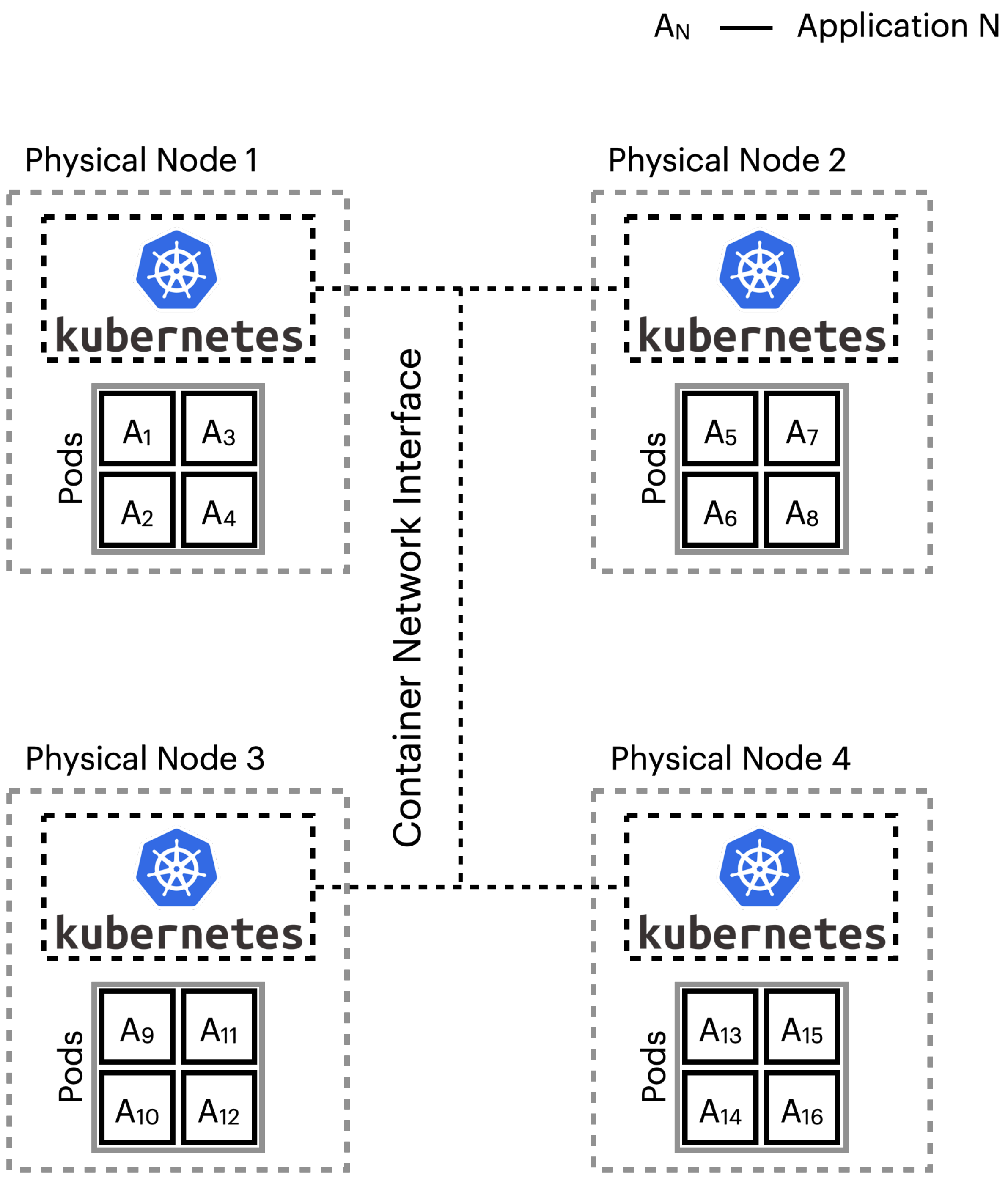


Figure 5.1: Kubernetes multi-node cluster structure.

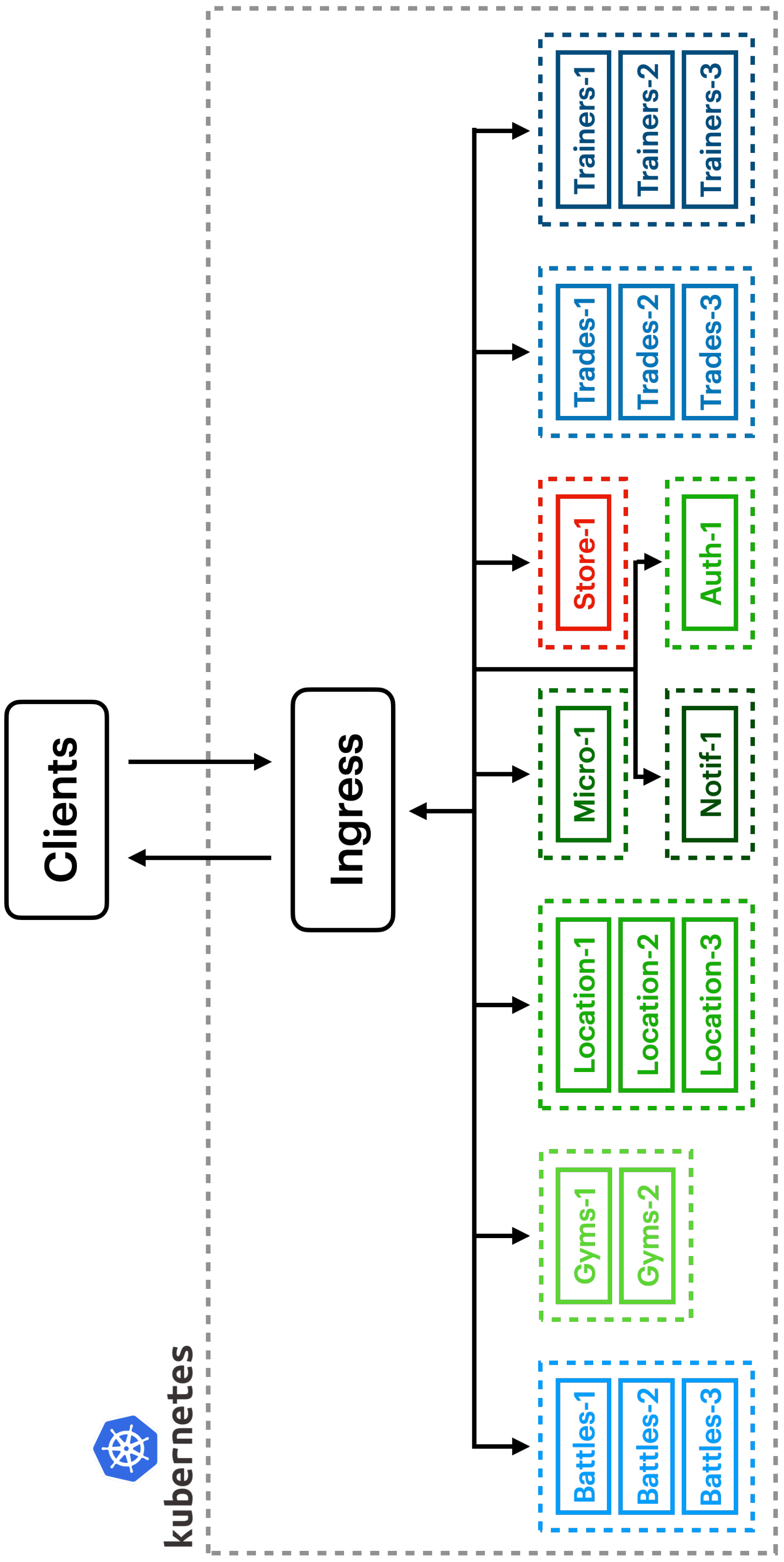


Figure 5.2: Kubernetes deployment architecture.

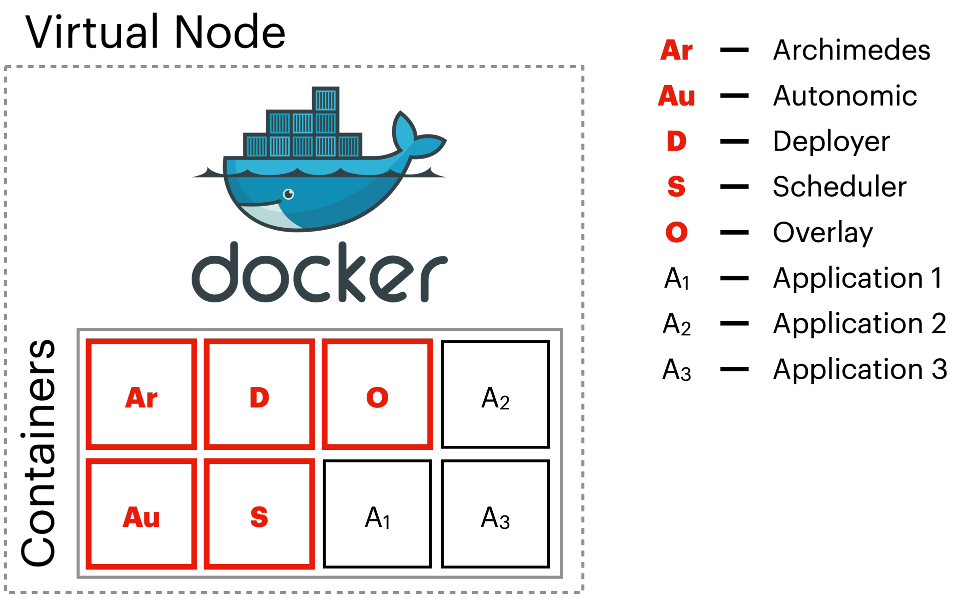


Figure 5.3: Virtual node architecture.

daemon across multiple *virtual nodes* would not provide the full isolation and could have unexpected behaviors regarding *Scheduler*, since this assumes that it has exclusive access over a Docker daemon, we chose to use a Docker daemon per *virtual node*. There is already a Docker image that suits our needs called Docker (previously known as Docker-in-Docker (dind)). Figure 5.4 depicts the architecture of a physical node running multiple nodes, both the outer and the inner Docker daemons as well as the solution stack and the applications running in each *virtual node*.

In order to connect the Docker daemons running the *virtual nodes* in each physical machine we used Docker Swarm. Docker Swarm creates an overlay network across the different physical machines which makes communication between *virtual nodes* in different machines transparent. Figure 5.5 presents the full architecture across multiple physical nodes.

Aiming to further emulate the real life scenario where nodes are spread throughout a region, we extracted latencies and locations from Wonder Network [63] and assigned each *virtual node* an identifier from such network, while applying the respective latencies and location. Wonder Network provides real world latency statistics between nodes in different cities, which can then be translated to a latitude/longitude format. In order to select the nodes, we do a random sample of N nodes within a specified boundary, which in our case we selected 50 nodes within Europe's boundaries, and we were handed the distribution depicted in Figure 5.6. In this figure we can see the edge nodes in black and the *fallback* node in red.

The scale we present here is not representative of all Edge Computing spectrum,

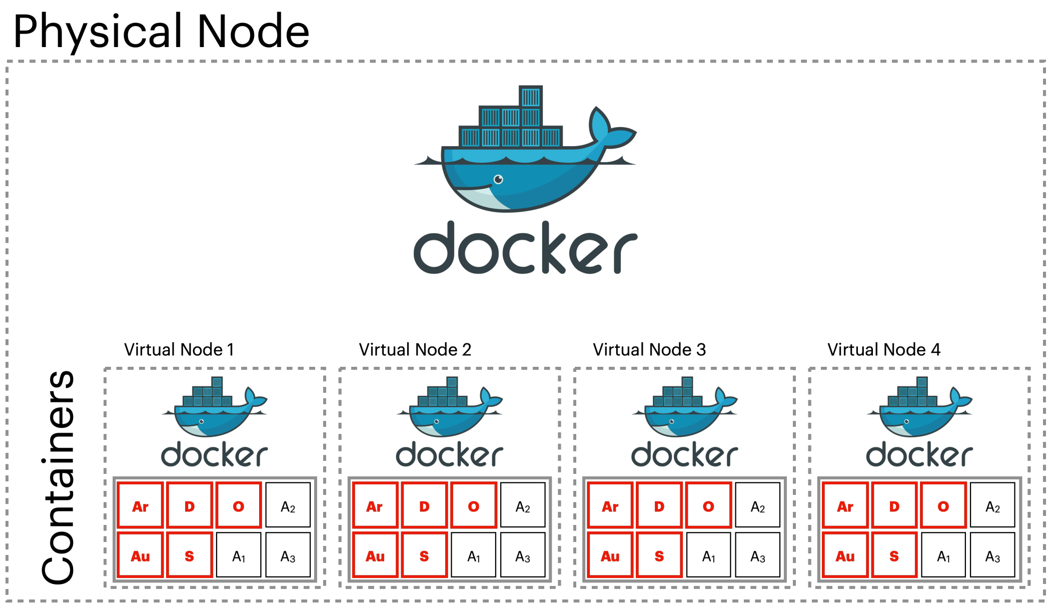


Figure 5.4: Physical node architecture.

nevertheless we argue that such distribution is the same despite the scale, this meaning that selecting a city as the *fallback* (cloud) and the remaining as edge nodes is equivalent to a smaller scale where we would select an Internet Service Provider (ISP) Point of Presence as the *fallback* node and the 5G towers as edge nodes. Our scenario is described as having a node that serves as *fallback* due to its higher availability, more resources and central location, and the remaining as edge nodes while having less resources, network links in a star-like graph, where there is a central point through which the communication goes through (*fallback*), therefore, our deployment solution can be instrumented at different levels of the Edge Computing spectrum due to this pattern being present throughout the Edge Computing model.

Another important aspect of our simulator is throughput limitations, and for this we decided to follow a simple strategy that we believe emulates the real scenario closely enough to draw conclusions from. Given the same experiment performed in our simulator and in a cloud scenario, the *fallback* node will have the same bandwidth limitations as the ingress in the cloud scenario, whereas the remaining nodes will be limited to a fraction of the *fallback* node, i.e. if the ingress in the cloud scenario is limited to 20 megabits, the *fallback* node will also be limited to 20 megabits and with a ratio of 0.25, the remaining edge nodes will be limited to 5 megabits. Even though this strategy is not fine-grained, we believe it mimics a real world scenario with enough detail to draw conclusions from.

Both bandwidth limits and delays are handled using Linux's Traffic Control (TC). Similarly to the bandwidth limits applied in the cloud ingress, the bandwidth limitations

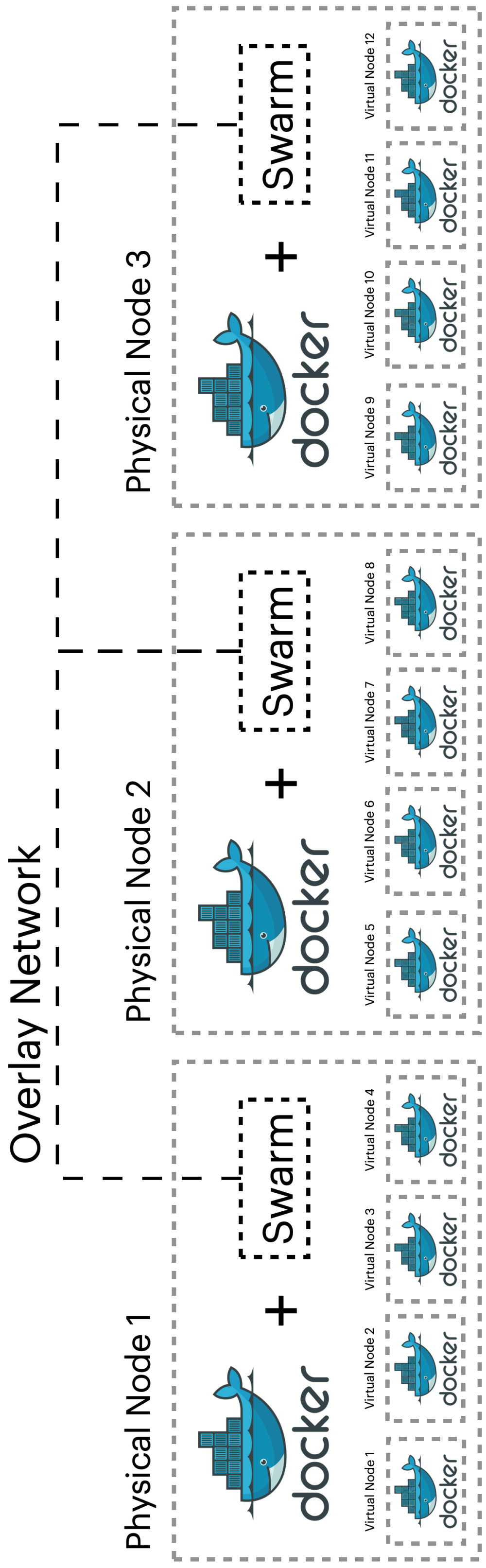


Figure 5.5: Multi node deployment architecture.

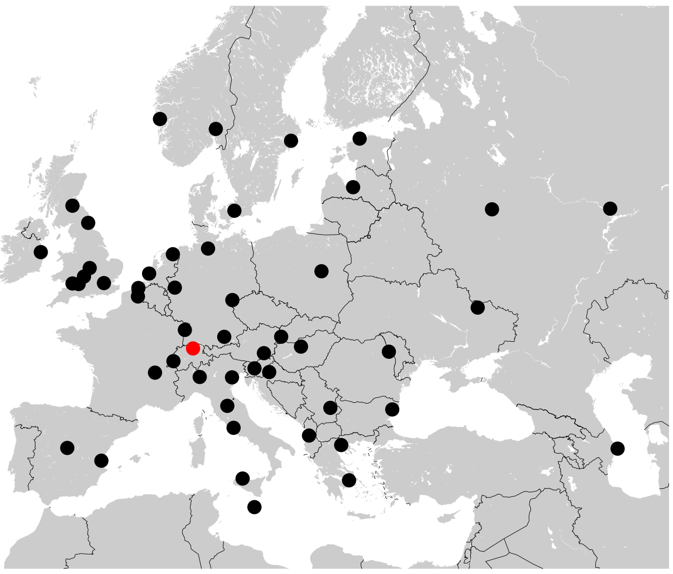


Figure 5.6: Random node distribution.

are applied both on the inbound and outbound interfaces, with the delay being only applied in the outbound interface. Figure 5.7 presents how bandwidth limits and delays are setup in each *virtual node*. It is important to note that the flow presented is not directly equivalent to how TC manages its internal token system, but for simplicity reasons we present the flow in this format.

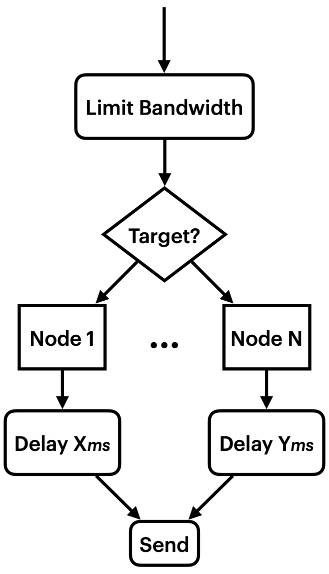


Figure 5.7: Random node distribution.

Lastly, there is a key difference on how PouchBeasts integrates in this scenario, more specifically regarding the *Notification* service and the databases. These are instantiated only in the *fallback* node, otherwise it would require implementing data partition schemes for the databases or add additional logic to coordinate multiple servers regarding *Notifications* service, which falls outside the scope of the work presented in this thesis.

Next, we will present the results of the conducted experiments in both scenarios previously described, analyze and draw conclusions from them.

5.2 Experimental Results

The comparison of both solutions (cloud only and cloud-edge) will focus on four main criteria: client latency, response ratio, number of retries and bandwidth usage. The different experiments will be characterized by number of clients (*N*), bandwidth limit at the *fallback* node (*BWF*) and bandwidth limit at the edge nodes (*BWE*). We will conduct 4 experiments that aim to consider 4 different scenarios: first a small scale scenario with few clients colocated in a single region far from the *fallback* node; next, the same scenario with additional clients aiming to saturate the bandwidth of the edge nodes; followed by the worst case scenario that overwhelms a region with even more clients colocated in a single region; and lastly a real life scenario with those clients spread throughout different regions.

5.2.1 Scenario 1 (*N*=50; *BWF*=40Mb/s; *BWE*=10Mb/s)

First, we present a scenario with few clients colocated in a region, in order to understand how the quality of service behaves when addressing a single cluster of users near an edge node. This experiment has the following characteristics: 50 clients; bandwidth in the *fallback* node limited at 40Mb/s; and bandwidth in edge nodes limited at 10Mb/s.

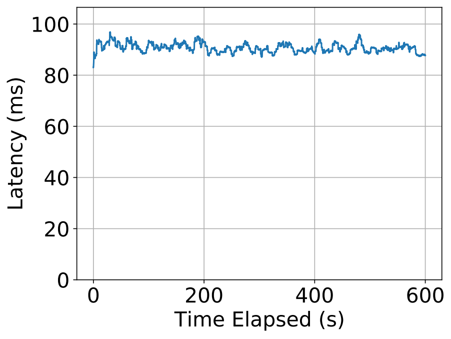
Figure 5.8, shows the latency experienced by the clients in the cloud scenario (Figure 5.8(a)) and in the cloud-edge scenario (Figure 5.8(b)).

In order to analyze the achieved latencies and understand how these compare to the expected results, we first calculated the expected latencies taking into account the regions of the clients and the *fallback* node, for the cloud scenario, as well as the clients and the closest node for the cloud-edge scenario (see Section 4.3). The expected latencies are 82*ms* and 16*ms* respectively.

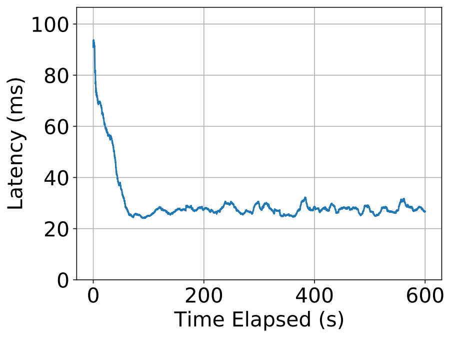
The average latency obtained in this experiment for the cloud scenario is 91 *ms*, whereas in the cloud-edge scenario is 30*ms*. These values translate into a 67% decrease when using the edge, which is a noticeable decrease in this metric.

Alongside the latency, Figure 5.8 also displays the response ratio for the cloud and the cloud-edge scenario (Figures 5.8(c) and 5.8(d) respectively). In both scenarios the response ratio is 100%, meaning all requests were fulfilled. Moreover, the number of retries in both scenarios was zero.

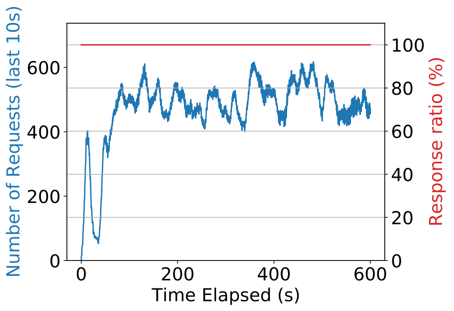
As the results show, the latency in both scenarios is over the expected, which is due to system contention, more specifically because PouchBeasts is designed to saturate the CPU by having as many interactions between clients as possible. This CPU saturation



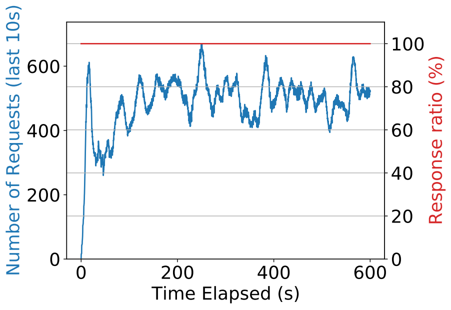
(a) Average client latency in the cloud scenario.



(b) Average client latency in the cloud-edge scenario.

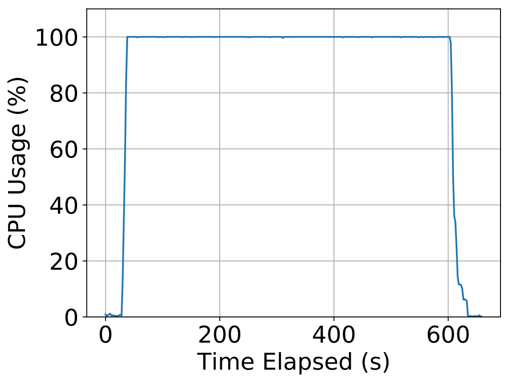


(c) Number of requests and response ratio in the cloud scenario.

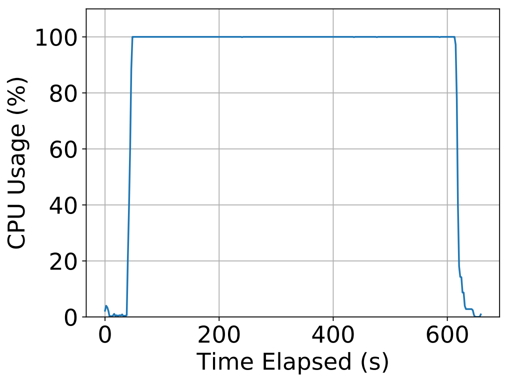


(d) Number of requests and response ratio in the cloud-edge scenario.

Figure 5.8: Quality of service measurements (*N*=50, *BWF*=40Mb/s, *BWE*=10Mb/s).



(a) Cloud scenario.



(b) Cloud-Edge scenario.

Figure 5.9: CPU usage in client nodes (*N*=50, *BWF*=40Mb/s, *BWE*=10Mb/s).

is evidenced in both scenarios depicted in Figure 5.9. In the cloud-edge scenario, it is also important to take into account that some interactions make the server at the edge node contact the cloud, and therefore it is predictable to increase the latency beyond the expected.

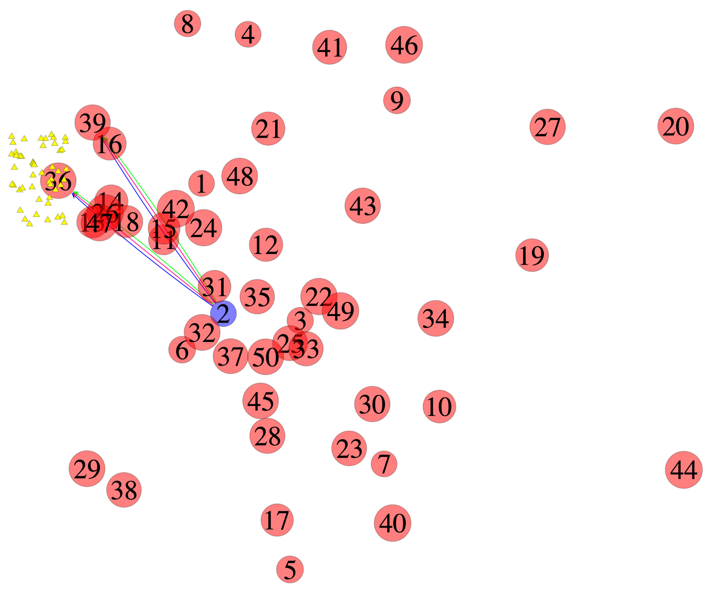


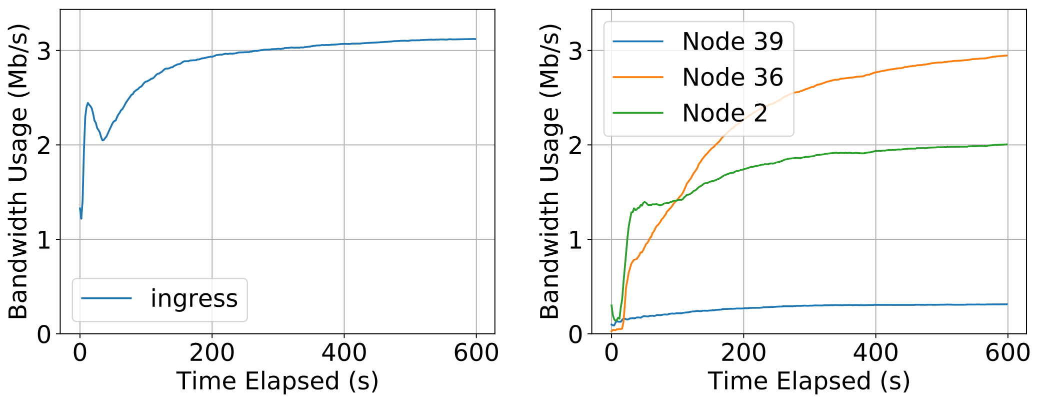
Figure 5.10: Deployment extensions from the *fallback* node to edge nodes (N=50, BWF=40Mb/s, BWE=10Mb/s).

We can understand how LowNimbus achieves a lower latency by analyzing Figure 5.10 which shows how different services (arrows) are being extend from the *fallback* node (Node 2) to two other nodes (Node 36 and 39) which are closer to the clients (triangles).

Figure 5.11 presents the bandwidth usage for the ingress in the cloud scenario and for the relevant nodes in the cloud-edge scenario which further validates the hypothesis that the additional delay is caused by CPU contention and not by bandwidth saturation. It also demonstrates how clients start using Node 36 and Node 39. One might argue that Node 2's bandwidth usage should decrease when the other nodes increase, but it is important to take into account that there are services that run on the *fallback* node only (see Section 5.1.2) and the metrics overlay uses a hierarchy that imposes a heavier load on the *fallback* node than the others.

5.2.2 Scenario 2 (*N*=100; *BWF*=40Mb/s; *BWE*=10Mb/s)

After understanding how both systems behave with a small number of clients in a single region, we decide to increase the number of clients in order to understand when the



(a) Cloud scenario.

(b) Cloud-Edge scenario.

Figure 5.11: Bandwidth usages in the cloud scenario and the cloud-edge scenario (N=50, BWF=40Mb/s, BWE=10Mb/s).

quality of service starts to deteriorate. This can be identified when the response ratio decreases, the number of retries increases or the latency increases.

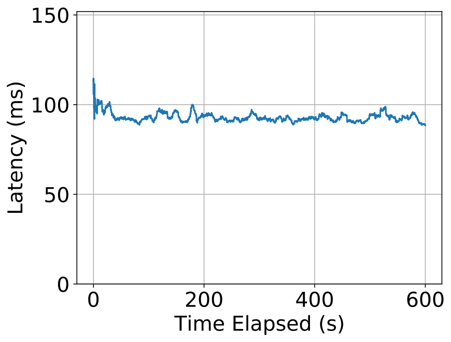
We then increased the number of clients to 100, while still keeping them in the same region, aiming to understand if the quality of service deteriorates when increasing the number of clients in a single region.

Figure 5.12(a) tells us a similar story as the previous experiment, indicating that the bandwidth usage by 100 clients is not enough to saturate the 40Mbits limit imposed in the ingress. On the other hand, 5.12(b) shows a clear deterioration in the quality of service experienced by the clients, with the average latency going from 30ms in the previous scenario to 89ms in the current one. Both scenarios present a close to 100% response ratio, with negligible number of retries.

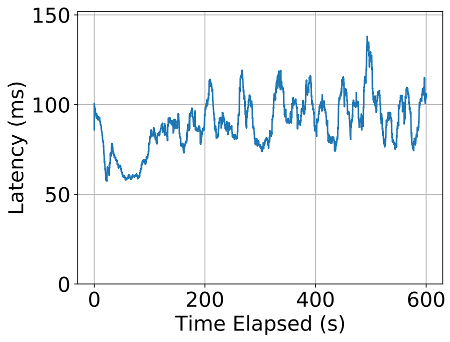
Analyzing Figure 5.13 we can not point out the reason for the deterioration in quality of service, since the bandwidth is still considerably lower than the 10Mb/s limit applied in Node 36 which serves as the point of presence in the edge for some services accessed by the client. This is due to how the limit is applied in the nodes, specifically both on incoming and outgoing connections at half the value of the total limit, meaning both incoming and outgoing are effectively limited at 5Mb/s.

Figure 5.14 shows how the incoming bandwidth is erratic and is very close to the limit occasionally exhibiting spikes that go over that limit. Furthermore, the bandwidth capture is done periodically and can therefore not have enough precision to show every moment the bandwidth is saturated. Nevertheless, the correlation is obvious since the latency increases around the 100 second mark, which is the exact moment bandwidth usage spikes.

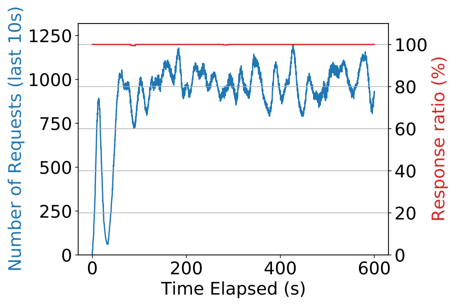
In order to further confirm that the previous experiment results were explained by the latency limit we kept the same scenario but increased the edge node bandwidth fraction to 0.5 (20Mb/s). The results, reported in Figure 5.15, validate our hypothesis by clearly



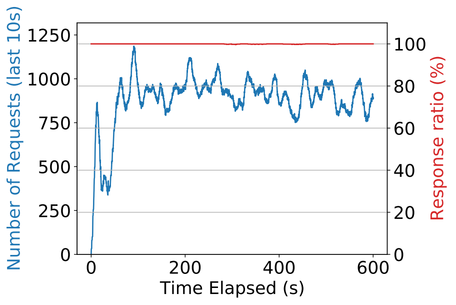
(a) Average client latency in the cloud scenario.



(b) Average client latency in the cloud-edge scenario.

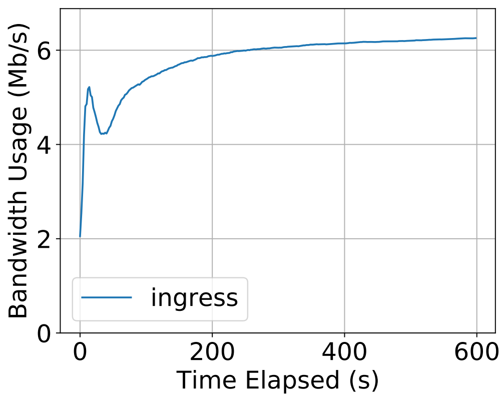


(c) Number of requests and response ratio in the cloud scenario.

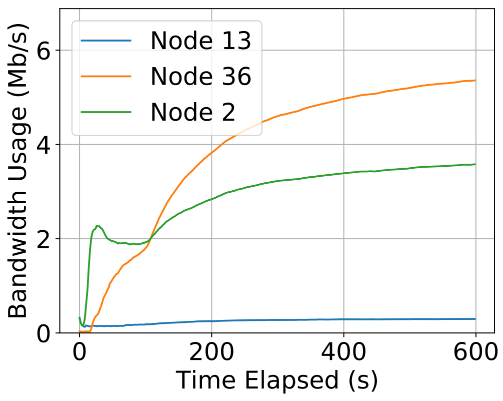


(d) Number of requests and response ratio in the cloud-edge scenario.

Figure 5.12: Quality of service measurements (*N*=100, *BWF*=40Mb/s, *BWE*=10Mb/s).



(a) Cloud scenario.



(b) Cloud-Edge scenario.

Figure 5.13: Bandwidth usages in the cloud scenario and the cloud-edge scenario (*N*=100, *BWF*=40Mb/s, *BWE*=10Mb/s).

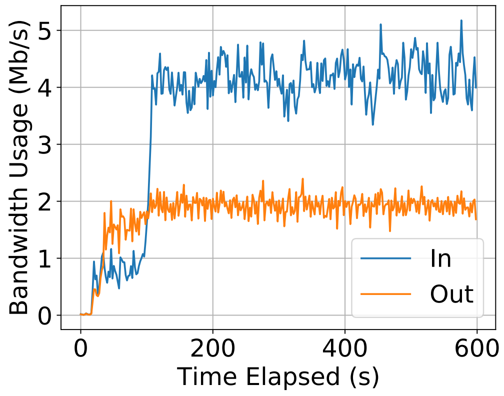


Figure 5.14: Bandwidth usage in Node 36. (*N*=100, *BWF*=40Mb/s, *BWE*=10Mb/s).

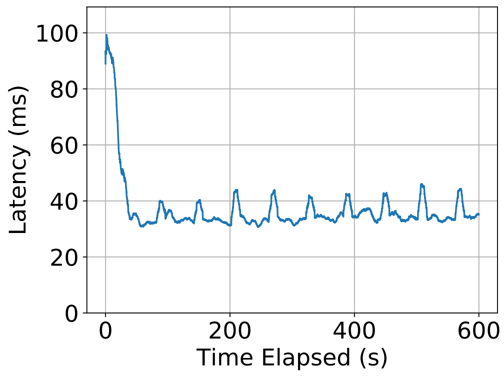


Figure 5.15: Average client latency in the cloud-edge scenario (*N*=100, *BWF*=40Mb/s, *BWE*=20Mb/s).

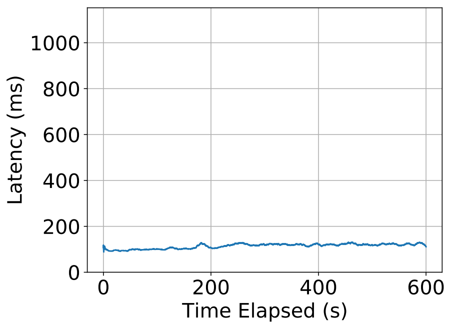
showing that when we increase the bandwidth on the edge nodes, the latency decreases to 37*ms*, similar to what was experienced in the first scenario.

5.2.3 Scenario 3 (*N*=400; *BWF*=40Mb/s; *BWE*=10Mb/s)

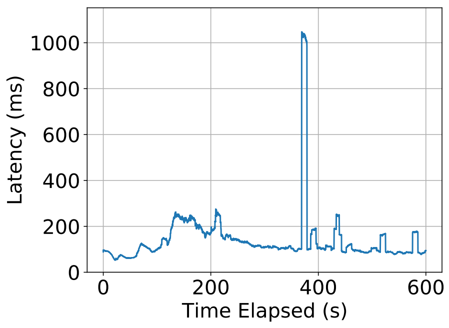
Aiming to simulate the worst possible scenario for our cloud-edge solution, we then devised an experiment that increased the number of clients further, all colocated in a single region. This scenario should put the load balancing features of our platform to the test, because as we can see in Figure 5.10 most clients will connect to Node 36 since this is the closest one. The scenario is described by the following properties: 400 clients; bandwidth in the *fallback* node limited at 40Mb/s; and bandwidth in the edge nodes limited at 10Mb/s.

Figure 5.16 presents the quality of service measurements, where we can see that the clients start with a low latency, but as more and more clients continue to issue requests, the response ratio starts to decrease and the latency increases. The average latency is 117*ms* for the cloud scenario and 146*ms* for the cloud-edge scenario.

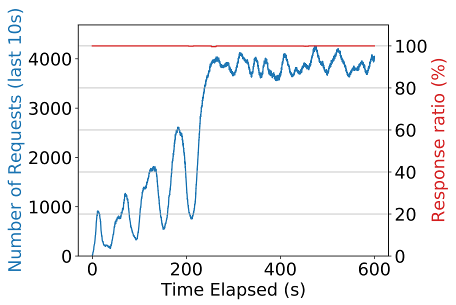
Around the 200 second mark both metrics start to improve. The results reported on



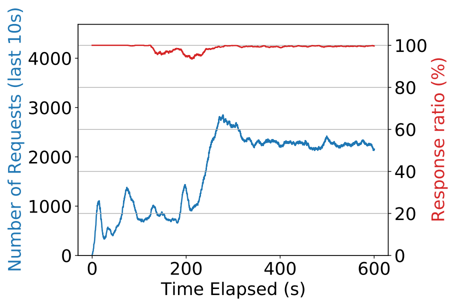
(a) Average client latency in the cloud scenario.



(b) Average client latency in the cloud-edge scenario.



(c) Number of requests and response ratio in the cloud scenario.



(d) Number of requests and response ratio in the cloud-edge scenario.

Figure 5.16: Quality of service measurements (*N*=400, *BWF*=40Mb/s, *BWE*=10Mb/s).

Figure 5.17 explain this behavior, since its around this mark that other nodes (Node 39 and Node 13) bandwidth usage starts increasing, due to offloading some clients from Node 36. Even though this offloading is enough to increase the response ratio to close to 100%, the number of requests issued between both scenarios is very different, with the cloud-edge scenario being penalized due to the amount of retries as denoted by the results on Figure 5.18.

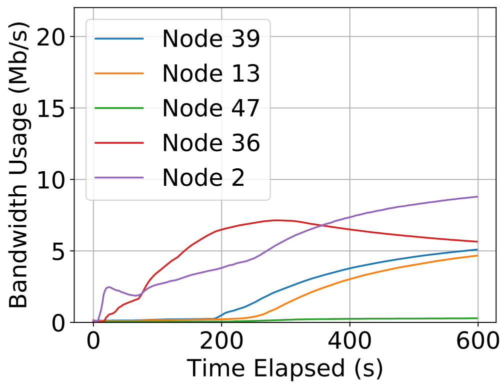


Figure 5.17: Bandwidth usage cloud-edge scenario. (*N*=400, *BWF*=40Mb/s, *BWE*=10Mb/s).

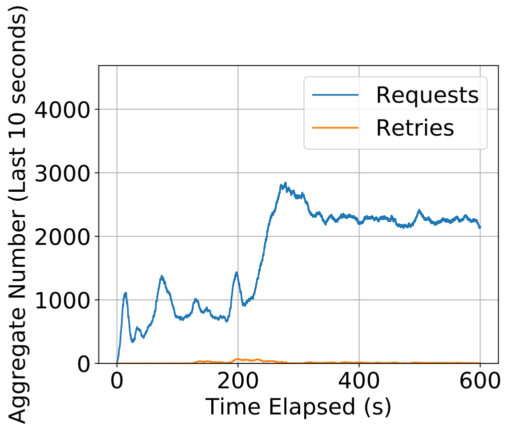
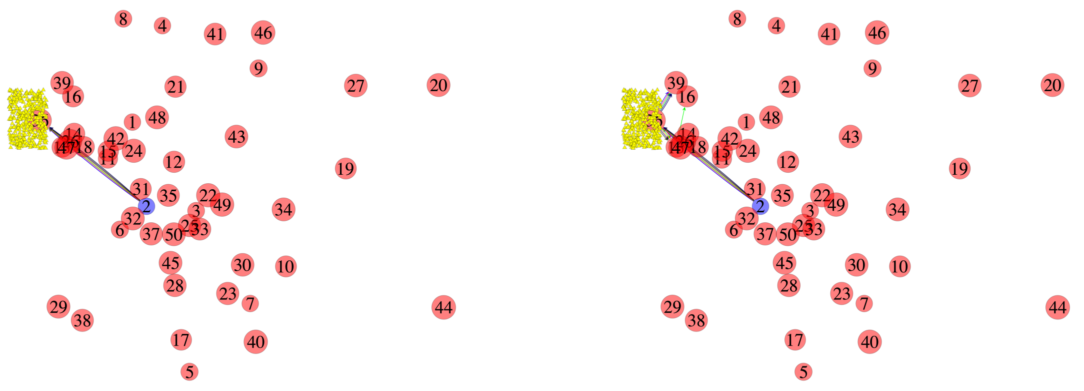


Figure 5.18: Number of requests and retries in the cloud-edge scenario. (*N*=400, *BWF*=40Mb/s, *BWE*=10Mb/s).

We can see how the services are distributed throughout the nodes in Figure 5.19. From Figure 5.19(a) to Figure 5.19(b) we can see how Node 36 extended the deployments to Node 39 and Node 13 offloading clients to these additional edge nodes.

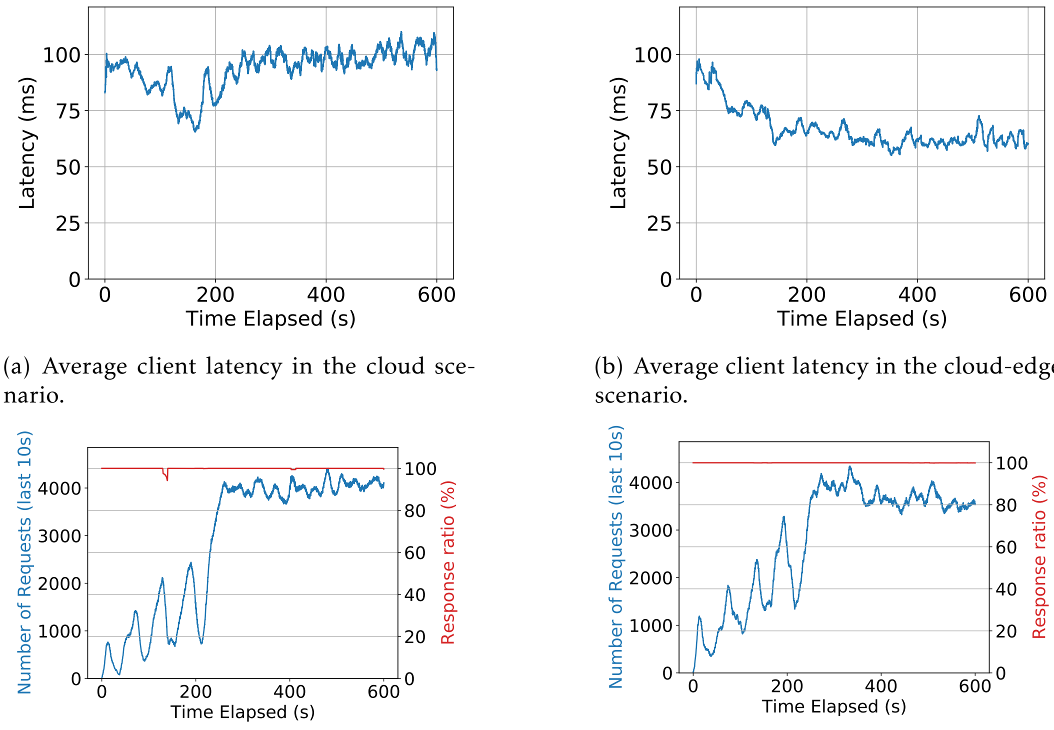


(a) Deployment extensions after 10 seconds. (b) Deployment extensions after 75 seconds.

Figure 5.19: Deployment extensions visualized. (*N*=400, *BWF*=40Mb/s, *BWE*=10Mb/s).

5.2.4 Scenario 4 (*N*=400 *spread*; *BWF*=40Mb/s; *BWE*=10Mb/s)

Lastly, since we believe the previous experiments are not representative of the edge computing paradigm due to all the clients being in a single cluster, we devised another experiment that spreads the 400 clients throughout Europe, since all the emulated nodes are in this region (400 clients *spread*, bandwidth in the *fallback* node limited at 40Mb/s, and bandwidth in the edge nodes limited at 10Mb/s). Figure 5.20 presents the quality of service metrics for this experiment.



(a) Average client latency in the cloud scenario.

(b) Average client latency in the cloud-edge scenario.

(c) Number of requests and response ratio in the cloud scenario.

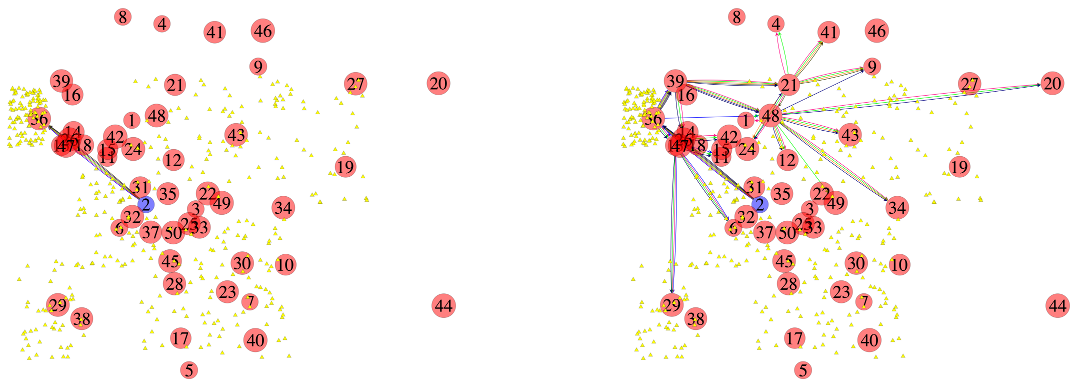
(d) Number of requests and response ratio in the cloud-edge scenario.

Figure 5.20: Quality of service measurements (*N*=400 *spread*, *BWF*=40Mb/s, *BWE*=10Mb/s).

Figure 5.20(a) depicts the average client latency in the cloud scenario. First, the clients start with a latency similar to the previous experiments, but then it drops due to the clients appearing in closer regions to the cloud. After more clients connect to the cloud, the latency inevitably rises due to clients being in further regions. In the cloud-edge scenario depicted in Figure 5.20(b), we can see that even though clients are being instantiated throughout different regions, our solution can cope with the load and act fast to deploy the services in different regions. The average latency in the cloud scenario is 95*ms* with the cloud-edge scenario decreasing to 65*ms*, translating into a 32% decrease in latency.

Figure 5.21 reports experimental results that capture how the different services first extend to the region on the left and then as clients appear in other regions, it spreads out to other nodes. This allows us to have a chain of nodes that can be used for the fault tolerance mechanism proposed as alternatives to when a node crashes. Furthermore, it takes additional pressure from the cloud by not extending only from the cloud outwards, and instead using the leaf nodes.

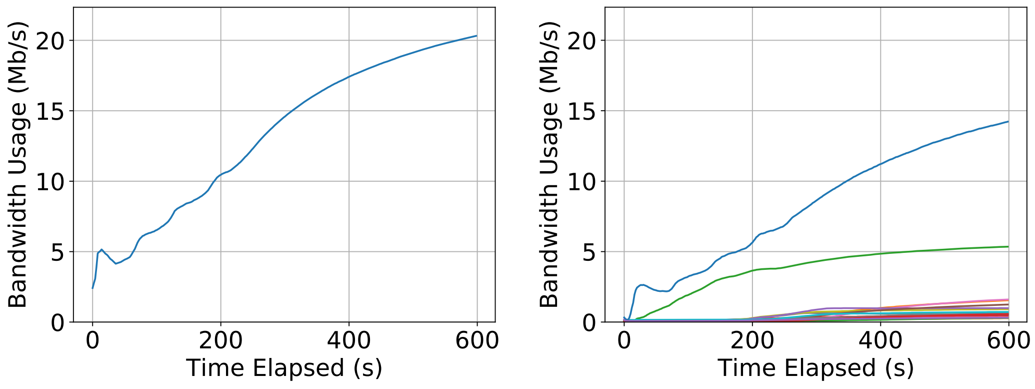
In Figure 5.22 we can see that not only many other nodes are being used (labels hidden due to space limitations), but the bandwidth usage of the *fallback* node (blue) is significantly smaller even with the additional overhead of our system, the number of



(a) Deployment extensions after 60 seconds. (b) Deployment extensions after 100 seconds.

Figure 5.21: Deployment extensions visualized. (*N*=400 *spread*, *BWF*=40Mb/s, *BWE*=10Mb/s).

clients offloaded to closer nodes has a positive impact. There is also a node that is more used than others (Node 36), with this being due to the higher density of players in that region.



(a) Cloud scenario. (b) Cloud-Edge scenario.

Figure 5.22: Bandwidth usages in the cloud scenario and the cloud-edge scenario (*N*=400 *spread*, *BWF*=40Mb/s, *BWE*=10Mb/s).

5.3 Discussion

Considering our experimental results that were obtained through several experiments that capture a multitude of different scenarios, it is evident that our solution is not a panacea for all cases. Scenarios such as the ones present in experiments for scenarios 2 and 3 show how client hot spots with few nodes in a given location can experience significant decrease of quality of service using our solution. Nevertheless, we show that even in the cases with quality of service degradation, our solution still manages to provide a minimum service through load balancing. In the last experiment, which is the closest to a real life scenario, we see how our solution thrives both in providing a better alternative for the clients regarding the latency to the services all around, and also by decreasing the bandwidth usage of the *fallback* node which ultimately solves the limitations of the centralized model of Cloud Computing and its bandwidth saturation.

The results obtained for scenarios 1 and 4 show that we can decrease the experienced latency by a considerable amount, 67% and 32% respectively, ultimately providing a better experience for the users with low intervention from system administrators. Evolving dynamically without any external intervention is one of the great advantages that our system has over other alternatives.

Chapter 6

Conclusion and Future Work

Conclusion

Computation at the edge of the network is on the rise as more and more cloud solutions offer service models that provide locations closer to the user, aiming to minimize some of the drawbacks that come from the centralized cloud model, such as high latency due to infrastructure saturation as well as long distance from the clients. With technologies such as 5G that promise computation capabilities even closer to the clients we researched how Cloud Computing and Edge Computing can work in a symbiotic manner across multiple subjects.

We first studied the key differences between the Cloud and the Edge computing paradigms aiming to design a framework that enhances the advantages and complement the drawbacks of both scenarios. For this we started on the topic of Serverless Computing which is the most recent paradigm established in Cloud computing that has variants implemented with an Edge Computing paradigm in mind, such as Lambda@Edge by Amazon [1]. After understanding that computations should flow from the Cloud to the Edge similar to the Osmotic Computing paradigm we researched different approaches on computation offloading and service migration. Finally, after having a bigger picture how the services would migrate, we studied different deployment frameworks and its techniques to manage deployments across multiple nodes in a large scale environment.

Next, we devised a solution that requires very low intervention, self-managing across multiple nodes that evolves in a decentralized manner, enabling the deployment of applications in a locality aware environment that strives for the best quality of service. We also proposed a novel benchmark that aims to provide future work with a real life like scenario and usage patterns, allowing for a reliable and trustworthy way of application's performance in different deployment scenarios.

Throughout this work, we aimed to minimize the disadvantages of building a decentralized system by centralizing crucial decision-making components such as the fault tolerance mechanism (fixed to the failing node's parent) and the entry point for the system (providing a consistent view of a managed application tree), components which would require coordination amongst multiple nodes for consistency purposes, ultimately increasing the bandwidth usage. These mechanisms could be fully implemented in a decentralized fashion but would incur in a more complex solution that would ultimately treat the cloud as a common node, not taking full advantage of its different characteristics such as low downtime and resource (computation and bandwidth) capabilities.

An important lesson that we can take from the work conducted in this thesis is that the decentralization/centralization of a system can be fine-tuned to take advantage of both models by making compromises on some components, as explained previously. Nevertheless, it is clear that the current cloud model on which applications are operating now is too centralized, and lacks scalability for highly interactive applications.

One important implementation decision that could have been done differently, is using an event-based paradigm for communication between the different components of the framework instead of the simpler HTTP handlers, which ultimately has a request-reply pattern, creating dependencies between messages across different components.

Future Work

There are some aspects discussed throughout this work that can be improved in the future, namely data partition and locality aware overlay. Both these topics were outside the work scope and therefore were overlooked. There is a need to tackle the data partition scheme that could reduce even further the latency in interactions that require communication with the *fallback* node. This could be managed similarly to ENORM (see Section 2.5.5) by caching data and operating over the local data avoiding going to the cloud for every operation. Furthermore, the locality aware overlay used in this project was not ideal, as it was explained in Section 3.1.5 and therefore further work to bias the overlay for locality metrics could be done as to improve and support the operation of systems as the one we devised in the context of this thesis.

Bibliography

[1] *Amazon Lambda@Edge*. https://aws.amazon.com/lambda/edge/. Accessed: 2020-01-13.

[2] *Apache OpenWhisk*. https://openwhisk.apache.org. Accessed: 2020-01-13.

[3] *AWS Lambda*. https://aws.amazon.com/lambda/. Accessed: 2020-01-13.

[4] D. Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by I. I. Ivanov et al. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1.

[5] D. Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* 1.3 (Sept. 2014), pp. 81–84. ISSN: 2372-2568. DOI: 10.1109/MCC.2014.51.

[6] A. Brogi and S. Forti. "QoS-Aware Deployment of IoT Applications Through the Fog". en. In: *IEEE Internet of Things Journal* 4.5 (Oct. 2017), pp. 1185–1192. ISSN: 2327-4662. DOI: 10.1109/JIOT.2017.2701408. URL: http://ieeexplore.ieee.org/document/7919155/ (visited on 01/28/2020).

[7] B. Burns et al. "Borg, omega, and kubernetes". In: *Communications of the ACM* 59.5 (2016), pp. 50–57.

[8] *ChakraCore: The core part of the Chakra JavaScript engine that powers Microsoft Edge*. https://github.com/microsoft/chakracore. Accessed: 2020-01-13.

[9] *Cloud Ping inter-region latencies matrix*. https://www.cloudping.co/. Accessed: 2021-06-14.

[10] E. Cuervo et al. "MAUI: making smartphones last longer with code offload". en. In: *Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys '10*. San Francisco, California, USA: ACM Press, 2010, p. 49. ISBN: 978-1-60558-985-5. DOI: 10.1145/1814433.1814441. URL: http://portal.acm.org/citation.cfm?doid=1814433.1814441 (visited on 01/14/2020).

[11] *Docker*. https://www.docker.com. Accessed: 2020-01-10.

[12] *Docker Hub*. https://hub.docker.com. Accessed: 2020-01-10.

[13] *etcd*. https://etcd.io/. Accessed: 2021-06-14.

[14] *Firecracker*. https://firecracker-microvm.github.io/. Accessed: 2020-01-13.

[15] *FORBES news on Pokemon Go outage*. https://www.forbes.com/sites/davidthier/2016/07/07/pokemon-go-servers-seem-to-be-struggling/#588a88b64958. Accessed: 2021-06-16.

[16] P. K. Gadepalli et al. "Challenges and Opportunities for Efficient Serverless Computing at the Edge". In: *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. 2019, pp. 261–2615. DOI: 10.1109/SRDS47363.2019.00036.

[17] P. Garcia Lopez et al. "Edge-centric Computing: Vision and Challenges". In: *SIGCOMM Comput. Commun. Rev.* 45.5 (Sept. 2015), pp. 37–42. ISSN: 0146-4833. DOI: 10.1145/2831347.2831354. URL: http://doi.acm.org/10.1145/2831347.2831354.

[18] *Golang Programming Language*. https://golang.org/. Accessed: 2021-06-14.

[19] *Google Cloud Functions*. https://cloud.google.com/functions/. Accessed: 2020-01-13.

[20] M. S. Gordon et al. "COMET: Code Offload by Migrating Execution Transparently". In: (Oct. 2012), pp. 93–106. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gordon.

[21] *Guardian news on Pokemon Go outage*. https://www.theguardian.com/technology/2016/jul/12/pokemon-go-australian-users-report-server-problems-due-to-high-demand. Accessed: 2021-06-16.

[22] A. Haas et al. "Bringing the Web up to Speed with WebAssembly". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363. URL: https://doi.org/10.1145/3140587.3062363.

[23] A. Hall and U. Ramachandran. "An Execution Model for Serverless Functions at the Edge". In: *Proceedings of the International Conference on Internet of Things Design and Implementation*. IoTDI '19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 225–236. ISBN: 9781450362832. DOI: 10.1145/3302505.3310084. URL: https://doi.org/10.1145/3302505.3310084.

[24] *HAProxy*. https://www.haproxy.org/. Accessed: 2021-06-14.

[25] *HashiCorp Nomad*. https://www.nomadproject.io/. Accessed: 2021-06-14.

[26] S. Hendrickson et al. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, June 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[27] B. Hindman et al. "Mesos: A platform for fine-grained resource sharing in the data center." In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[28] *IBM Cloud Functions*. https://cloud.ibm.com/functions/. Accessed: 2020-01-13.

[29] M. Iorga et al. *Fog computing conceptual model*. Tech. rep. 2018. DOI: 10.6028/NIST.SP.500-325.

[30] J. Kangasharju, J. Roberts, and K. W. Ross. "Object replication strategies in content distribution networks". In: *Computer Communications* 25.4 (2002), pp. 376–383. ISSN: 0140-3664. DOI: https://doi.org/10.1016/S0140-3664(01)00409-1. URL: http://www.sciencedirect.com/science/article/pii/S0140366401004091.

[31] *Kata Containers*. https://katacontainers.io/. Accessed: 2020-01-13.

[32] H.-S. Kim. "Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are". In: *International Journal of Cisco* (2016).

[33] *Kubernetes*. https://kubernetes.io. Accessed: 2020-01-13.

[34] *KVM: Kernel-based Virtual Machine*. https://www.linux-kvm.org/. Accessed: 2020-01-27.

[35] *Lean OpenWhisk*. https://github.com/kpavel/incubator-openwhisk/tree/lean. Accessed: 2020-02-18.

[36] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 2007, pp. 419–429. DOI: 10.1109/DSN.2007.56.

[37] J. Leitão et al. "Towards Enabling Novel Edge-Enabled Applications". In: *CoRR* abs/1805.06989 (2018). arXiv: 1805.06989. URL: http://arxiv.org/abs/1805.06989 .

[38] *Linux Containers*. https://linuxcontainers.org/. Accessed: 2020-01-13.

[39] F. Liu et al. "NIST cloud computing reference architecture". In: *NIST special publication* 500.2011 (2011), pp. 1–28.

[40] A. Machen et al. "Live Service Migration in Mobile Edge Clouds". en. In: *IEEE Wireless Communications* 25.1 (Feb. 2018), pp. 140–147. ISSN: 1536-1284. DOI: 10.1109/MWC.2017.1700011. URL: http://ieeexplore.ieee.org/document/8000803/ (visited on 01/14/2020).

[41] R. Mahmud, R. Kotagiri, and R. Buyya. "Fog computing: A taxonomy, survey and future directions". In: *Internet of everything*. Springer, 2018, pp. 103–130.

[42] D. Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[43] *Microsoft Azure Functions*. https://azure.microsoft.com/en-us/services/functions/. Accessed: 2020-01-13.

[44] D. Ongaro and J. Ousterhout. "In search of an understandable consensus algorithm". In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319.

[45] J. Paavilainen et al. “The Pokémon GO experience: A location-based augmented reality mobile game goes mainstream”. In: *Proceedings of the 2017 CHI conference on human factors in computing systems*. 2017, pp. 2493–2498.

[46] S. Rajan and A. Jairath. “Cloud Computing: The Fifth Generation of Computing”. In: *2011 International Conference on Communication Systems and Network Technologies*. June 2011, pp. 665–667. DOI: 10.1109/CSNT.2011.143.

[47] *Red Hat Openshift*. https://www.openshift.com/. Accessed: 2021-06-14.

[48] I. Rocha et al. “ABEONA: an Architecture for Energy-Aware Task Migrations from the Edge to the Cloud”. en. In: *arXiv:1910.03445 [cs]* (Oct. 2019). arXiv: 1910.03445. URL: http://arxiv.org/abs/1910.03445 (visited on 01/14/2020).

[49] *S2 Cells*. http://s2geometry.io/devguide/s2cell_hierarchy. Accessed: 2021-03-04.

[50] M. Schwarzkopf et al. “Omega: flexible, scalable schedulers for large compute clusters”. In: *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 2013, pp. 351–364. URL: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf.

[51] V. Scoca et al. “Scheduling Latency-Sensitive Applications in Edge Computing:” en. In: *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. Funchal, Madeira, Portugal: SCITEPRESS - Science and Technology Publications, 2018, pp. 158–168. ISBN: 978-989-758-295-0. DOI: 10.5220/0006706201580168. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006706201580168 (visited on 02/18/2020).

[52] W. Shi et al. “Edge Computing: Vision and Challenges”. In: *IEEE Internet of Things Journal* 3.5 (Oct. 2016), pp. 637–646. ISSN: 2372-2541. DOI: 10.1109/JIOT.2016.2579198.

[53] *SpiderMonkey: The Mozilla JavaScript runtime*. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. Accessed: 2020-01-13.

[54] S. Upadhya et al. “A State-of-Art Review of Docker Container Security Issues and Solutions”. In: *American International Journal of Research in Science, Technology, Engineering & Mathematics, ISSN (Print)* (2016), pp. 2328–3491.

[55] *V8 Javascript and WebAssembly Engine*. https://v8.dev. Accessed: 2020-01-13.

[56] A. Verma et al. “Large-Scale Cluster Management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741964. URL: https://doi.org/10.1145/2741948.2741964.

[57] M. Villari et al. “Osmotic computing: A new paradigm for edge/cloud integration”. In: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83.

[58] S. Voulgaris, D. Gavidia, and M. Van Steen. “Cyclon: Inexpensive membership management for unstructured p2p overlays”. In: *Journal of Network and systems Management* 13.2 (2005), pp. 197–217.

[59] *Voyager Mesh*. https://voyagermesh.com/. Accessed: 2021-06-14.

[60] N. Wang et al. “ENORM: A framework for edge node resource management”. In: *IEEE transactions on services computing* (2017).

[61] S. Wang et al. “Dynamic service migration in mobile edge-clouds”. en. In: *2015 IFIP Networking Conference (IFIP Networking)*. Toulouse, France: IEEE, May 2015, pp. 1–9. ISBN: 978-3-901882-68-5. DOI: 10.1109/IFIPNetworking.2015.7145316. URL: http://ieeexplore.ieee.org/document/7145316/ (visited on 01/14/2020).

[62] *Windows Containers*. https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/. Accessed: 2020-01-13.

[63] *Wonder Network*. https://wondernetwork.com/pings. Accessed: 2021-06-14.

[64] W. Zhang et al. “SEGUE: Quality of Service Aware Edge Cloud Service Migration”. en. In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Luxembourg, Luxembourg: IEEE, Dec. 2016, pp. 344–351. ISBN: 978-1-5090-1445-3. DOI: 10.1109/CloudCom.2016.0061. URL: http://ieeexplore.ieee.org/document/7830702/ (visited on 01/14/2020).