Bruno André Pitta Grós do Valle e Anjos

Degree in Computer Science and Engineering

A Distributed Deployment Solution for Cloud/Edge Environments

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

Master of Science in
Computer Science and Informatics Engineering

Adviser: João Leitão, Assistant Professor,
NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2020

The work reported in this document was partially supported by
Fundação para a Ciência e Tecnologia through the Project NG-STORAGE (PTDC/CCI-INF/32038/2017)

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 model.

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 can allow to operationalize efficiently the 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. Moreover, we discuss the performance metrics that will be taken into account to measure the quality of the solution, and how applications will integrate with the solution.

Throughout the document not only we will 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.

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 factor 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 infra-estruturas 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, discutimos as métricas de desempenho que serão usadas para medir o sucesso da mesma, e como os serviços se irão integrar com tal solução.

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

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

Contents

List of Figures

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 materialisation 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 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 deployment in the cloud that could 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 a lot of users.

1.2 Motivation

Even though cloud computing can fulfill the computational and storage needs of most applications, centralizing these resources has limitations. It can be easier to manage them, but it poses limitations on clients that are far from this central point. Cloud computing uses replication techniques to reduce the impact on these clients, by using multiple datacenters around the world, thus minimizing the distance from a 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 [13] (such as mobile interactive games similar to Pokemon Go), where the application has to wait for a message to contact the cloud and back just to interact with a user that can be located by its side.

In the recent years a new computing paradigm has appeared 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 [22], 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, deployment, 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 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.

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 deployment of services at the edge, in a decentralised manner, while aiming to achieve lower latencies when compared to cloud deployments. We will focus on service migration, scalability, fault tolerance and latency optimisation.

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 research topic. Resource monitoring will also be depreciated. 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.

1.4 Expected Contributions

The main expected contribution of this work is a framework that will meet previously mentioned the objectives. This main contribution can be partitioned into the following:

- Design of a 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 comparing performance metrics, such as latency, with centralized cloud solutions.

A further analysis on application components and the respective operations will be conducted as to understand which operations make a service eligible for edge deployment and that can improve the most.

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 and service migration.

Chapter 3 describes the research that we plan to conduct in the context of the thesis, discusses how we plan to evaluate our proposal, and presents the planning for the following months.

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 analyses 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 analysed 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 of the 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 analyse 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 Cloud Computing

Cloud computing has been concisely defined by NIST as follows [30]:

> *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 [30].

2.1.1 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 virtualised 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 virtualisation by VMs is one of its strengths since it enforces a hard barrier between the different systems. Nevertheless, this kind of lower level virtualisation translates into a bigger overhead from the VMM, since everything from the operating system upwards has to be ran 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 request 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 more popular is containerization. The use of containers aims to be a more lightweight virtualisation mechanism that execute directly on top of a host operating system. Even though ideally containers aim to provide the same level of isolation as virtual machines, in all fairness, software defined virtualisation is still more susceptible to allow possible unwanted interference from a container to the main system due to its architectural design [42].

The first container-based system was Linux containers (LXC) [29] that used *chroot*, *namespaces*, and *cgroups* to achieve OS level isolation. As mentioned in [6] 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 [9]. Docker would eventually popularize containers in industry, becoming one of the most popular container platforms [33] in part due to its ease of use and the existence of platforms like Docker Hub [10] 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 [4]. Google has been one of the main industry researchers on containerization technology throughout the years, with systems such as Borg [44], Omega [37] and Kubernetes [24]. 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 withing 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 2.6.1 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 [6].

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 [35] 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 2.3 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.

2.1.2 Discussion

Throughout this section we analysed 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.

2.2 Edge Computing

Edge computing is an emergent distributed computing model that aims at pushing the computation from the cloud datacenters 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 datacenters) 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 [13, 28, 39].

Other models such as Fog Computing [21, 32, 41] and Osmotic Computing [45], 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. P2P networks or mechanisms in the context of edge computing would make inter-user communication faster.

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).

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 this applications in edge computing.

2.2.1 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*"[32]. 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 materialised 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 on which these are deployed. According to [32] the literature uses different keywords to described these fog nodes, which include micro servers, micro datacenters, nanoservers, etc. Other works also characterizes 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 [45] 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.

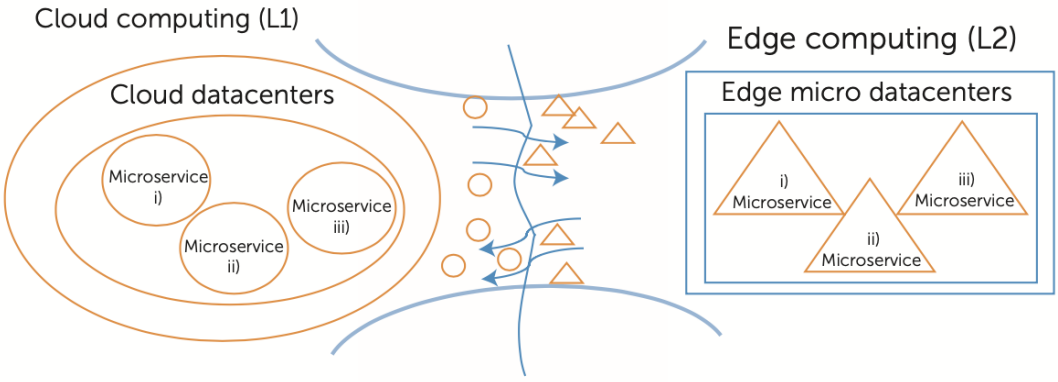


Figure 2.1: Osmotic Computing ideology. Image from *M. Villari, M. Fazio, S. Dustdar, O. Rana,and R. Ranjan. "Osmotic computing: A new paradigm for edge/cloud integration."*

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

2.2.2 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 of the traffic will be pre-processed 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 of the 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 guaranteeing message delivery and withstand node failures. Works such as [27, 46] 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 might 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 2.3 we present existing recent proposals that might be beneficial in devising our own solution.

2.3 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 [14], Microsoft Azure Functions [34], and IBM Cloud Functions [20] [12]. There are open source solutions for serverless computing, such as OpenLambda [18] and Apache OpenWhisk [2], with the latter being analysed 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.

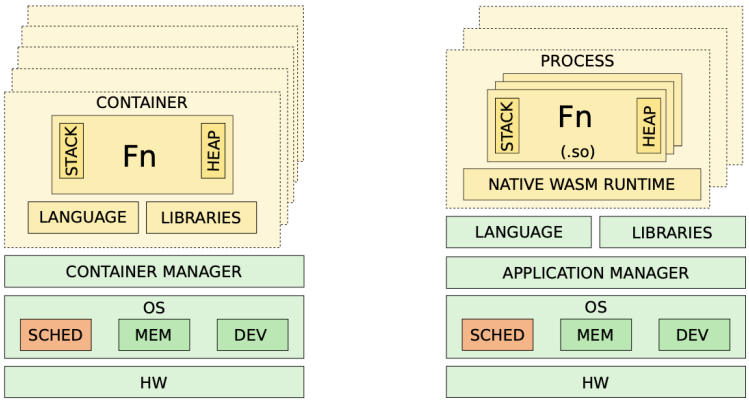
Flexible Programming Model In order to support as many services as it can, most of the platforms providing FaaS services usually support multiple programming languages.

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 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 [29], Docker [9], Windows Containers [48], and Kubernetes [24] [12]. Some cloud providers also have VM-based solutions such as Amazon's Firecracker [11] and Kata Containers [23].

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 we have discussed virtualisation 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 [16] 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 [12] [17]. Some examples of such runtime environments are Google's V8 [43], Mozilla's SpiderMonkey [40], and Microsoft's Chakra [7]. 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 lightweight than containers is that the language environment (such as libraries) is the same for all the tasks executing (see 2.2b). 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.



(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. Image from *P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer. "Challenges and Opportunities for Efficient Serverless Computing at the Edge."*

2.3.1 Solutions

In the following we will detail two existing solutions for serverless computing, one based on containers and another one that heavily relies on WebAssembly.

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 ran 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 [26].

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 [12] 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, multi core 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 39 μs depending on the task. There is a cold-start cost 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 virtualisation, 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.

2.3.2 Discussion

The biggest advantage of a Wasm-based solution is it lightweight environment, even when comparing with containers. In [17] 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 [17] 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 well 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.

2.4 Computation Offloading

In this section we will analyse existing solutions for task offloading. Computation offload can have different approaches such as the integration of RPCs (Remote Procedure Call) in the application that simply sends 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 (see section 2.4.1). 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 2.5) being a particular instance of computation offloading.

2.4.1 Solutions

The following solutions present two very distinct ways of offloading computation. Maui employs an RPC-based solution, whereas COMET implements a DSM model that enables execution transfer for any part of the application.

MAUI

MAUI [8] 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 the 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 focus of MAUI.

There are three main building blocks that make the MAUI system, both on the client side and the server side (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 transfered, 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.

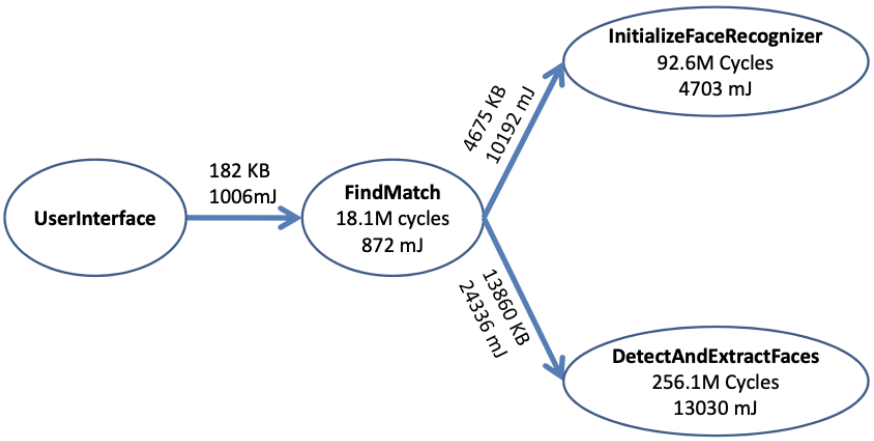


Figure 2.3: MAUI call graph of an example application. Image from *E. Cuervo , A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl . "MAUI: making smartphones last longer with code offload."*

The authors designed MAUI to handle failures having in mind that mobiles are, as the name say, 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 problem 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 offload or not, like in mobile offloading, but instead where it should be offload to.

COMET

COMET [15] (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 endpoints.

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 visualise the migration process.

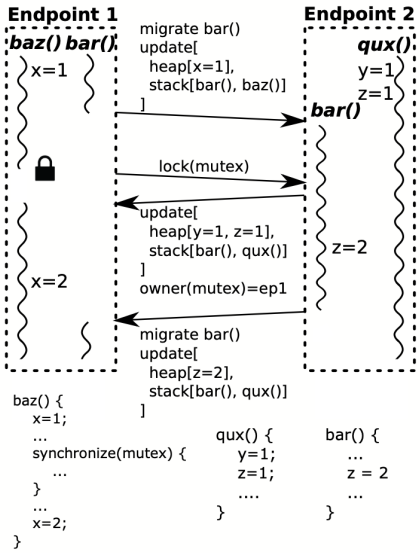


Figure 2.4: COMET migration mechanism. Image from *M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. "COMET: Code Offload by Migrating Execution Transparently."*

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 time.

By using 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.

2.4.2 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 a large number of requests is 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 analyse a different technique to achieve execution transfer to another machine called *service migration*.

2.5 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 transfered 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 analyses 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.

2.5.1 Solutions

The solutions presented here target different aspects of service migration and the environment the migration is done on. This can be migration technique (e.g., VM migration) or task scheduling, and it can focus on a specific metric such as QoS, consumption of energy, or optimal resource utilization.

Live Service Migration in Mobile Edge Clouds

In [31] the authors propose a layered framework for live service migration using two virtualisation 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) [29] and Kernel-based Virtual Machine (KVM) [25] 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 transfered 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 realised 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 transfered since ideally, the destination machine only needs the instance layer, assuming it had previously ran 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.

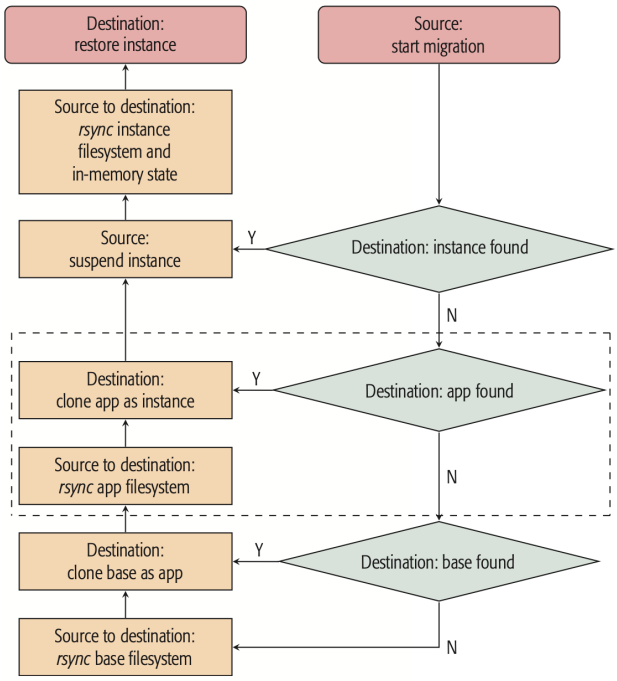


Figure 2.5: Migration mechanism in the proposed three layer model. Image from *A. Machen, S. Wang, K.K. Leung, B. J. Ko, and T. Salonidis. "Live Service Migration in Mobile Edge Clouds."*

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 filesystem such as overlayFS that creates different layers for each operation modifying the filesystem, 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 a 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.

ABEONA

ABEONA [36] is an edge to cloud task migration system that focus 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.

SEGUE

A very important factor in service migration is quality of service (QoS). SEGUE [49] 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 [47]), 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. More simple 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 ran again to find a better edge cloud. The work presents a QoS focused approach, where the MDP is ran 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) [47] 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.

FogTorch

In [5] the authors propose FogTorch that focus 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 the 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 then 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.

Scheduling Latency-Sensitive Applications in Edge Computing

In this work [38] the authors present a framework that aims to reduce the latency in applications between the user and the server. The main problem, 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 an 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 penalised since a good VM score should translate into a good score across all components.

The framework then aims to maximise 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 much 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 virtualisation technique used (VMs) has very high resource requirements for deployments on edge nodes.

2.5.2 Discussion

Considering the solutions presented in this section we note that all solutions have their own advantages and drawbacks, since they simply 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 the 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.

2.6 Deployment

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 analyse 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). Given that edge computing is a somewhat recent subject, to our best knowledge there are no solutions that focus on deployment at the edge, taking advantage of a migration mechanism.

2.6.1 Solutions

The following paragraphs 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.

Borg

Borg [44] is described as a *'large-scale cluster management system'* developed at Google to enable Google developers to deploy tasks throughout it's 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 penalising 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.

Omega

Omega [37] 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 analysing 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 realise 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.

Mesos

Mesos [19] 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 is comprised 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 of the 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.

2.6.2 Discussion

Most of the solutions that we analysed in greater detail in this section, have a lack of awareness for 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.

2.6.3 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 analysed 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 of the features and requirements that the future work will have, providing hints on how to do and validate these features.

Chapter 3

Planning

3.1 System model

Following the *Osmotic Computing* model presented in section 2.2.1, a solution that aims to migrate microservices between the cloud and the edge and make the most out of both environments, should take advantage of the proximity to the user and consider the resource limitations at the edge, while offloading the more computing intensive tasks to the cloud.

Leveraging the points of presence at the edge can have different final goals, such as decreasing power consumption, improving quality of service, preprocessing data, etc. Even though ideally a solution would perform the best at all these metrics, such solution might be infeasible, since some of these properties are inter-dependent, e.g. for a better QoS it might be needed more communication between the end device and the edge, thus impacting power consumption.

The system model considered in this work, focuses on applications that have human interaction at its core. In this kind of applications, latency is crucial, so decreasing it will be the main goal. Ideally the application follows a microservice-oriented model that decomposes it in multiple services, thus enabling fine grained migration. This fine grained control enables partial migration, which can offload only the services that have impact on human interaction.

More specifically the latency between the client and the server will be minimized by migrating components of the application from the cloud to the edge, where the proximity to the user can allow for lower RTTs in addition to having copies of these components serving a smaller number of users.

We assume that the membership of the system is known, at least partially and it may be at times inconsistent, due to nodes leaving and joining the network.

Upon deploying the application to the edge, the data on which the application operates needs to be locally available at the node, since issuing requests to the cloud to get the data for every request would incur in poor performance, rendering the edge deployment advantages meaningless. This will be treated independently of the proposed framework, hence we will assume that the data is present at the node on which we will deploy the application component.

3.2 Proposed Solution

The proposed solution will be packaged within a deployment framework that can migrate the services between the cloud and the edge, that can control the number of copies of computational components of managed applications, yet support a lightweight environment that can run in edge nodes.

One of the main challenges that any solution deployed at the edge needs to face is the volatility of the membership. It is important to take into account that a centralized solution is not feasible in this context since delegating all the responsibility to a node, which can leave the network, can compromise the availability of the system and that of applications managed by it. Therefore our solution will be fully decentralized enabling the use of localized fault-tolerance mechanisms based on replication and enabling fast recovery by avoiding a centralised control, even though at a higher cost for membership management.

This framework will take advantage of lowering the latency between the clients and the services by deploying the services at the edge nodes, and migrate or replicating them according to user usage. It will partition the users set according to their geographical position and the edge nodes available, using IP geolocation. Initially it can employ a basic solution by partitioning the set of users to a minimum number of nodes, but depending on resource usage it can spread out the application component throughout more nodes, or even decrease the number of nodes on which it is deployed.

The framework to be devised will be more suitable for deployments of stateless services, since there is an overhead to deal with state migration, where a stateful service would be paying the overhead without any benefits.

3.2.1 Applications

Applications that use this framework may have different levels of integration. Different applications might prefer to specify user partitions or leave to the framework to partition its components geographically. This is especially useful when for example there are users communicating/interacting with each other, that would end up in different partitions if it left the partitioning schema to the framework.

Applications will also have to provide an executor that will be the entry point of the application. A technique such as the one presented in Mesos, where the application also

provides a scheduler that is responsible for accepting or rejecting resource offers might also be required in our solution.

3.2.2 Fault-Tolerance

Since the set of edge resources available is not static, the nodes need to handle failures and optimization of deployment when new edge services become available. When a node fails another one should be able to identify which services were running on it, and start running these itself. In order to know which processes were running on each node, each node replicates to a small set of its neighbors control information regarding the tasks it is running. This minimises network congestion, and in case a service goes down, the best spot for the service to be redeployed is somewhere close of where it was previously since it will be closest to the clients that were accessing it.

An additional passive backup mechanism might be employed, that gives an extra guarantee when multiple nodes in a single location fail simultaneously (for instance due to power outages).

3.3 Evaluation

The evaluation of the proposed solution will take into account two scenarios: cloud deployment and edge deployment. In order to evaluate the cloud deployment we will use a cloud platform yet to be determined, whereas in the edge scenario the testbeds will be mainly comprised of clusters such as the Grid5000 and the computing cluster available at the department of Computer Science at NOVA School of Science and Technology of NOVA University Lisbon.

The solution evaluation will mainly focus in the following metrics:

Latency We will measure how fast a client receives a response after it has sent a request. There are already expectations regarding this metric, implying that the latency should be lower in the edge deployment scenario, since the edge is closer to the client. Nevertheless, the overhead that the management layer has in this scenario can have a penalty on the overall performance of the system.

Fault-Tolerance Relate how many nodes can fail with the degradation of quality of service (i.e. latency, requests served). The main performance impact in this case may come from the system reconfiguration, redeploying the services after it has failed. Ultimately there is also the possibility of failing too many nodes, and redeploying all the services lost on the remaining nodes saturating the nodes resources.

Overhead The impact that the decentralised management layer will have in the system. In the cloud deployment scenario this layer will not exist, whereas in the edge scenario this will be responsible for all the communication between nodes.

Reconfiguration Time After a node goes down, how long will it take for the system to redeploy the services that were running on the failed node. This metric is particularly interesting in the edge scenario where the system will have to manage failures and redeploy the services.

The solution will also be designed in a modular way, so that simpler models can be compared with the final solution by replacing the modules. A possible solution that will be considered as a baseline is deploying the application at the edge without any kind of distributed network, making all the edge nodes communicate through a centralized component present in the cloud. This is mostly due to the lack of similar solutions, therefore evaluating the different possibilities provides a view on how other possible solutions would perform, shedding light over the benefits of our own proposal.

3.4 Work Plan

The work will be developed throughout the following seven months. This will be split into the following tasks:

Task 1: Design

- **1.1** Determine the framework main layers/components
- **1.2** Define the framework modules

Task 2: Implementation

- **2.1** Implement the solution
- **2.2** Implement other modules that will provide a simple (cloud-based) solution to compare with
- **2.3** Fix eventual errors that will appear in the solution

Task 3: Evaluation

- **3.1** Test the solution for errors
- **3.2** Evaluate the solution and store metrics
- **3.3** Test the remaining solutions and store metrics

Task 4: Thesis writing

The chart in 3.1 illustrates the planned time frame for each task.

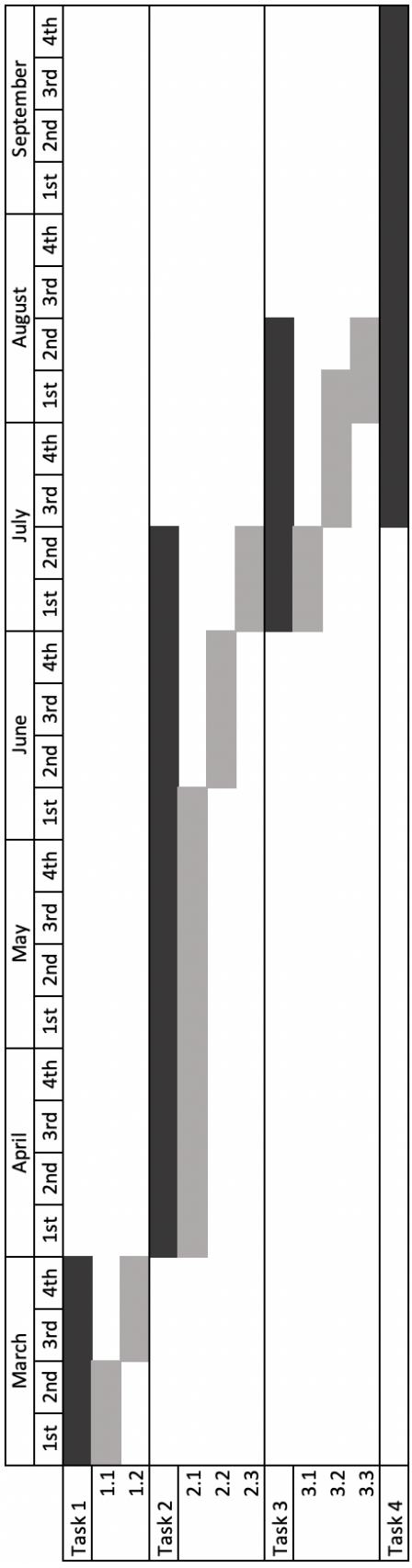


Figure 3.1: Gantt chart with task planning

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

[5] 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).

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. "Borg, omega, and kubernetes." In: (2016).

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

[8] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. "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).

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

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

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

[12] P. K. Gadepalli, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer. "Challenges and Opportunities for Efficient Serverless Computing at the Edge." In: ().

[13] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. "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.

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

[15] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. "COMET: Code Offload by Migrating Execution Transparently." en. In: (), p. 14.

[16] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. "Bringing the Web up to Speed with WebAssembly." In: *SIGPLAN Not.* 52.6 (June 2017), 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363. URL: https://doi.org/10.1145/3140587.3062363.

[17] 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, 225–236. ISBN: 9781450362832. DOI: 10.1145/3302505.3310084. URL: https://doi.org/10.1145/3302505.3310084.

[18] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "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.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." en. In: (), p. 14.

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

[21] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. S. Goren, and C. Mahmoudi. *Fog computing conceptual model*. Tech. rep. 2018.

[22] 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.

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

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

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

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

[27] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: a membership protocol for reliable gossip-based broadcast." en. In: (2007), p. 10.

[28] J. Leitão, P. Á. Costa, M. C. Gomes, and N. M. Preguiça. "Towards Enabling Novel Edge-Enabled Applications." In: *CoRR* abs/1805.06989 (2018). arXiv: 1805.06989. URL: http://arxiv.org/abs/1805.06989.

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

[30] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf. "NIST cloud computing reference architecture." In: *NIST special publication* 500.2011 (2011), pp. 1–28.

[31] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis. "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).

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

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

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

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

[36] I. Rocha, G. Vinha, A. Brito, P. Felber, M. Pasin, and V. Schiavoni. "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).

[37] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. "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.

[38] V. Scoca, A. Aral, I. Brandic, R. De Nicola, and R. B. Uriarte. "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).

[39] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. "Edge Computing: Vision and Challenges." In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. ISSN: 2372-2541. DOI: 10.1109/JIOT.2016.2579198.

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

[41] C. Systems. "Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are." In: (2015).

[42] S. Upadhya, J. Shetty, H. Raja Rajeshwari, and D. G. Shobha. "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.

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

[44] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. "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.

[45] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. "Osmotic computing: A new paradigm for edge/cloud integration." In: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83.

[46] S. Voulgaris, D. Gavidia, and M. van Steen. "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays." en. In: (), p. 21.

[47] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung. "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).

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

[49] W. Zhang, Y. Hu, Y. Zhang, and D. Raychaudhuri. "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).