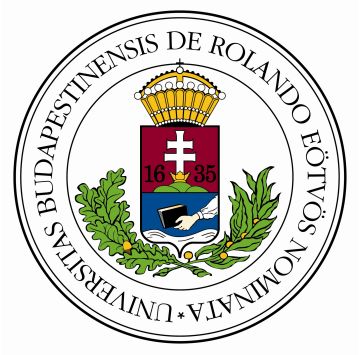
10.15476/ELTE.2021.123

Eabling 5G Edge Native Applications

Doctoral Dissertation
2021

Reale Anna
anna.reale@inf.elte.hu



Doctoral advisor: **Dr. Zoltán Horváth, PhD, habil.**
Doctoral advisor: **Dr. Melinda Tóth, PhD**
Industrial advisor: **Dr. Benedek Kovács, PhD**
Eötvös Loránd University, Faculty of Informatics,
1117 Budapest, Pázmány Péter sétány 1/C
Department of Programming Languages and Compilers

ELTE IK Doctoral School of Informatics
Doctoral Programme of Foundations and Methodologies of Informatics
Head of the doctoral school: Erzsébet Csuhaj-Varjú, DSc., habil
Head of the doctoral program: Zoltán Horváth, PhD, habil.

Contents

List of Figures

Acknowledgements

I would like to thank wholeheartedly my two supervisors Zoltan and Melinda for the support and guidance during these five years. Knowing that, despite your overwhelmed schedules, your doors were always open for me, really made the difference.

Thank you to my Ericsson supervisor Bence, for all the challenges, all the teachings and all the fun.

Thank you to my ever presents colleagues:

Peti, for all the time spent with me at our office, and for coping with my bad character, for all our little fights that still ended up into making things that worked.

Charles, for sharing anxieties and work, both in Ericsson and ELTE, and for being my fake almost-husband for all this years.

Michael, for the day and nights spent working and learning together, for the great company and ever presents smiles and good humor.

I am sure without all of you there would be no thesis!

Also an infinite thank you to Davide, for his never-ending patience, for always being ready to listen to my complaints and for being the awesome partner that he is.

Thank you to my flatmate Iti, for the awesome tea and chats covered in dark humor we shared at home after work.

Thanks to my favourite coffe-mate Akos, and all of the CLC stuff and the ELTE colleagues that made everyday life so much better!

And of course, thank you to my mum, sister and brother for giving me the opportunity and the guidance needed to start and move forward with my studies.

Chapter 1

Introduction

The research described in this thesis revolves around Edge-Native Applications. In fact, since the 5G Mobile Edge infrastructure is not entirely in the market yet, there are many questions about how to facilitate the new category of applications that it will host.

The focus points of this thesis will be deployment, mobility, network and cloud integration in 5G and Edge Computing (EC).

1. **Deployment** - An efficient and distributed allocation scheme is needed in EC. The optimization of resource allocation may be multi-objective and varies due to heterogeneity of applications, servers, user demands and connections. A centralized approach is not suitable for distributed MEC (Mobile Edge Computing) systems [14, 15] thus a distributed one should prevail.

2. **Mobility** - Edge servers will mostly be located near the RAN (Radio Access Network). Since 5G requires a dense network of small, and low-power cellular base stations (nano or femto cells), user mobility will cause frequent handovers. Service disruption will need to be avoided. Users may move during the computation offloading period and their data will have to travel with them. Mobility will create also a requirement for application portability. Finally, the variations in user requests, may affect uplink interference and performances of the computing resources [16].

3. **Network and Cloud integration** - EC should be interacting with the underlying network architecture, profiting from existing interfaces [11]. Furthermore, a distributed EC will not have all the computing resources needed to satisfy user requests. A solution will be transferring intensive and non-latency related operations to the cloud [17].

For this reason, our work concentrated at first in assisting developers in making a cloud or monolithic application into one that could fit the Edge. Secondly, we

tried to establish how to *deploy* edge-native applications and how to guarantee their *mobility*. Finally, we concentrated on building a full infrastructure, optimizing the scheduling and *integrating* cloud services and current 5G network standards.

The three main **research questions** we will try to answer in this work are the following:

1. How to facilitate deployment on the new Edge computing infrastructure? Can deployment be optimized in real-time? [1, 2, 3]

2. How to Migrate applications? What infrastructure works for migration and whether or not live migration can be achieved in real-time scenarios. [4, 5]

3. Creating a full infrastructure for distributed Edge Computing. There are two main reasons for migration: re-balancing and user movement. Can we reduce service redeployment, adding dynamic context awareness? How much this reflects on scheduling performances and service availability? [6, 7]

Three **thesis** follow our main questions:

1. **Facilitate Deployment at the Edge:** Adapting code from Cloud to Edge involves a great amount of work to refactor applications and services. Usually developers do this solely based on experience, simulating tools can be of support since the infrastructure is new and blind deployment may be expensive. Furthermore the extensive mobility of the applications represents a challenge for the choice of a correct separation of applications in micro-services. **I created a framework for partitioning and estimation of best deployment of a monolithic application. I implemented it in the form of a tool to assist into selecting the proper way to refactor an application, via graph partitioning schemes. I demonstrated this concept partitioning an AR application at a function level granularity.** Via non-intrusive online profiling of the application I created a call graph representing functions runtimes and number of calls between functions. The partition process applies min edge cut using a refinement algorithm, MLKL, that Coarsen, Partition and Uncorsen a graph multiple times. Any weighted graph may be used to represent the application in this step in our tool.

   This method aims at suggesting partitions that minimize their interaction, the idea is that the partitions should be easily moved around the network without compromising performances. Indeed more complex approaches may involve other parameters, such as the amount of allocated memory or the dependency to other code/services.

Finally the tool can suggest pareto-optimal places for deployment, based on Network resources capacities and existing links. I simulate the placement and routing issues into a single algorithm taking care to handle multiple user request at the same time. I show how it is possible to compute a deployment that satisfies SLA keeping in consideration cost of resources and benefit of requests. I proved that the approach does performs better than a random deployment, reducing deployment and networks failures.

2. **Migration in Edge Computing:** In Edge computing it will be necessary to move an edge-native application from one edge to the other, following the user, or, for recovery reasons, even from an Edge to the Cloud. The best approach to avoid this would be to have Stateless Apps, but there are cases in which this is not a possibility.

   I implemented a Java simulation of a Fog network, integrating live migration trough means of CRIU library calls and coordination via Docker Swarm. After testing the limits of this configuration I set up a better version based on Kubernetes. I constructed a docker in docker solution that is deployed as a daemon set on each node. This component is in charge of actuating iterative pre-copy migration of its hosted containers, if triggered by any other software running in the cluster. I have tested the migration against stop and restart of multiple container deployments and shown how, given a stable network connection (like a 5G coverage) we can use migration to achieve a lower Service Downtime. .

3. **Distributed Mobile Edge Scheduling:** In many Edge scenarios, especially those related to distributed Edge and IoT, it is for the best to reduce the need for application migration from the start. **I proved that we can reduce need for migration trough the improvement of the deployment strategies. I made a lightweight monitoring tool to feed real time telemetries to a scheduler component of the edge orchestrator. Thanks to awareness of cluster status resources are used more efficiently: 10 to 20 more ngnix containers can be scheduled on our cluster of raspberrypies compared to the default K8s scheduler. The devices do not crash due to overheat contrary to the default scheduling setup.**

   I selected a scoring system that no only increased life of the raspberrypies but that makes the scheduler up to 64% faster than the default. The score is based on a composition (Multiplicative Exponent Weighting) of nodes telemetries, where the minimum value correspond to the best candidate. Weights are using Coefficient of Variation: the more a parameter is variable

the higher its influence on the deployment.

1.1 Thesis Structure

This document is structured as illustrated in Table 1.1:

Chapter 1 contains a short background introduction to Edge Computing and 5G and includes our own classification and definition of Edge-native applications as we described in [8].

Chapter 2 concentrates on Thesis 1 and presents our efforts to facilitate partitioning and first deployment of applications on an Edge infrastructure.

Chapter 3 presents the issue of application mobility and our efforts towards integrating container live migration into networks relying on existing container orchestration technologies. This Chapter collects our Thesis 2. In particular we present an ad-hoc Fog network exploiting Docker Swarm and a pattern to apply live migration within a Kubernetes cluster.

Chapter 4 focuses on the improvement of scheduling approaches for Distributed Edge, illustrating our algorithm for a Kubernetes scheduler and comparing it with state of the art approaches;

Finally, in Chapter 5 we synthesize the scope, the problems and solutions, together with the domain and contribution of this work.

Chapter	Content
1	Background: context and focus
2	Theses 1: Facilitate Deployment at the Edge
3	Thesis 2: Migration in Edge Computing
4	Thesis 3 : Distributed Mobile Edge Scheduling
5	Summary: all theses and contributions

Table 1.1: Chapters and their content.

1.2 Background

As the number of mobile devices and services requiring computing or storage capabilities that significantly exceed their own capacities has exploded, the paradigm of Cloud Computing (CC) had arisen and gained a continuously increasing importance. The concept of CC is based on Data Centers (DC), which are capable of coping with storage and processing requirements of tasks involving large scale

data. Moreover, data centers are usually connected between each other over optical cable building up Data Center Networks (DCNs) that, due to meager internal communication costs, appear to the outside world as a single entity. When facing a problem that outruns available local resources, one can offload the code and the data to the Cloud, then - when computations are done - receive the results back.

The paradigm of CC has given a solution to the scalability problems related to inefficient resources. Code and data migration, however, may involve a significant latency or cause congestions in the network. The root of the problem can be summarized as location unawareness of the CC paradigm and is getting severe with time as more and more (semi-)intelligent devices will attempt to connect to DCNs.

The concept of Edge Computing has emerged to leverage the storage and computation capabilities of the edge devices that are connected to the Internet and meant to be an intermediate layer between the devices and the CLoud. They are able to handle a subset of requests that would be usually sent to the Cloud, but which, in fact, do not need its real involvement because of the diminished resource requirement or no need for the DC to be involved. Thus, thanks to the presence of this type of device, the computation load of DCs is reduced, similarly to the latency of responses, when an application needs to have real-time or almost real-time responses.

Moreover, due to its geo-distributed nature and high availability, the Edge layer is apt to handle the challenges of mobility. E.g., to serve requests of moving users as it is necessary in cases of autonomous cars or providing streaming and real-time gaming in a high-speed vehicle.

1.2.1 Edge Computing architectures and infrastructures

Edge Computing (EC) [18] in Mobile networks is an architecture that suggests deploying cloud servers in each base station of the mobile network. Such disposition will allow network owners to satisfy the requirements for the 5G era: acceleration of content, services, and applications, and increase in the responsiveness.

Edge computing has been proposed for mobile cloud computing and IoT cloud computing in different formats by different parties. A fundamental component of the edge computing services is the underlying cloud infrastructure (based on physical hardware) that hosts the applications and the supporting functions.

In the following paragraphs, we will present a distinction of Edge Computing paradigms into three main categories, as previously associated by [19]: Cloudlets, Fog computing, and Mobile Edge Computing .

These conceptual models are partially overlapping and complementary. The core idea to run applications and related processing tasks in proximity of mobile

users is common to all. We also will describe related industrial implementations such as Distributed Clouds, Mini Clouds, and Nano and Micro Datacenters.

Cloudlets

Cloudlets are an extension of the cloud or a "data center in a box" [20, 21]. They are self-managed, energy-efficient, and simple to deploy on-premises. The term Cloudlet [22] can be defined as a trusted set of computers having a good connection to the Internet and making their resources available to nearby mobile devices. The Cloudlet runs a virtual machine that is capable of provisioning resources to the connected users near real-time over a WLAN network in one-hop distance and with high bandwidth. Above provisioning, the infrastructure, Cloudlet architecture also provides a middleware framework support to component-based applications designed with a focus on applications with strict real-time requirements such as Augmented Reality.

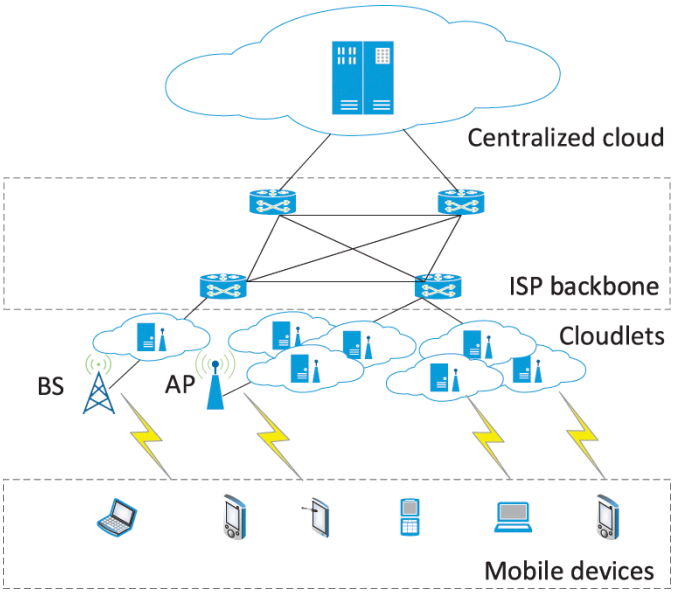


Figure 1.1: A cloudlet is a mobility-enhanced small-scale cloud data centre that is located at the edge of the Internet [9].

Fog Computing

Fog computing is another edge paradigm that mainly supports the Internet of things (IoT) applications [23]. The OpenFog Consortium was founded in 2015 by Cisco Systems, ARM Holdings, Dell, Intel, Microsoft, and Princeton University. In this paradigm, Fog Computing Nodes (FCN) can be placed at any point in the

architecture. FCNs are highly heterogeneous; they can be built on various devices like routers, switches, IoT gateways, set-top boxes, etc. The heterogeneity of devices leads to the ability to work with different protocols as well as with non-IP-based technologies in communication between FCNs and the end-devices. Since the inhomogeneity of the Edge naturally should stay hidden from the user devices, FC systems expose a uniform interface containing storage and computational services along with monitoring security and device management facilities. On top of this abstraction layer, an orchestration layer organizes resource allocations according to user requests.

With the term Fog computing, we refer thus to a highly virtualized platform that provides services related to computation, storage and networking, and that can be seen as a layer in between the Data Centers and the user equipment/devices. Since this original definition leaves the Fog paradigm very close to other Edge Computing ones as Mobile Edge Computing [24] and Cloudlets [21], we adopt Vaquero [25] views on Fog computing. The author distinguishes Fog from other Edge computing architecture by specifying that the involved nodes may be in a great quantity and extremely heterogeneous (wireless and sometimes autonomous). Vaquero envisions Fog nodes that can deploy decentralized communication and potentially cooperate with each other to perform storage and processing tasks in an autonomous fashion. Nodes tasks can range from basic network functions support to new services and applications running in a sandbox environment. Thus, the most interesting characteristic of this interpretation would be that users can become an active part of the Fog network, by not only being consumers but also providers: leasing part of their devices to host services and get incentives for doing so.

The main factors that will bring the Fog can be summarized as follows:

- *Nodes edge location and geographical distribution*: Fog nodes must be deployed at the Edge of the network and must be able to locate at least their neighbors in the local area. Due to the geographical distribution and the closeness to the final user, they will be able to communicate at low latency and reduce in data movement across the network significantly.

- *User mobility*: fog applications can communicate directly with mobile devices via mobility techniques and protocols such as LISP [26] (Cisco's Locator/ID Separation Protocol) that can decouple host identity from location identity [27].

- *Nodes heterogeneity*: Fog is a multi-layered hierarchical infrastructure, with dynamic and miscellaneous nodes deployed in a wide variety of environments, possibly in both physical and virtual form.

- *Nodes interoperability*: Fog nodes must be all-purpose and able to interoperate even if related to different providers' networks.
- *Nodes and users real-time interaction*: services with low latency, and involving continuous real-time interactions between users and system.

Mobile Edge Computing (MEC)

In the first declinations of MEC, the acronym referred to Mobile Edge Computing. In recent years the European Telecommunications Standards Institute (ETSI) shifted this paradigm toward Multi-access Edge Computing, whose parts are further investigated in Section 1.2.3 while current efforts in the literature and standardization are synthetized in Figure 1.2.

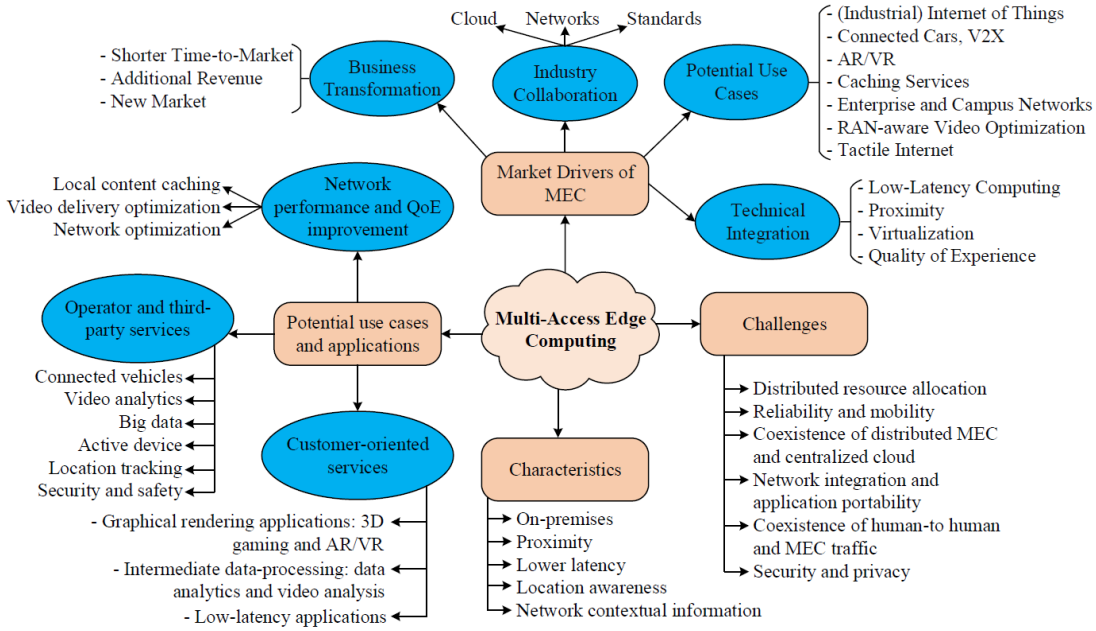


Figure 1.2: An overview on MEC [10]

MEC [28] is responsible for bringing storage and computational resources to the Edge to reduce latency and improve location awareness.

ETSI describes a MEC implementation fully reliant on 5G in [11]. In this architecture(Figure 1.3) the MEC orchestrator can interact with the Network Exposure Function (NEF), or in some scenarios directly with the target 5G NFs. The MEC host, i.e. the host level functional entities, are most often deployed in a data network in the 5G system. An instance of NEF can also be deployed in the edge to allow low latency, high throughput service access from a MEC host.

The distributed MEC host can accommodate, MEC apps, a message broker as a MEC platform service, and another MEC platform service to steer traffic to local accelerators.

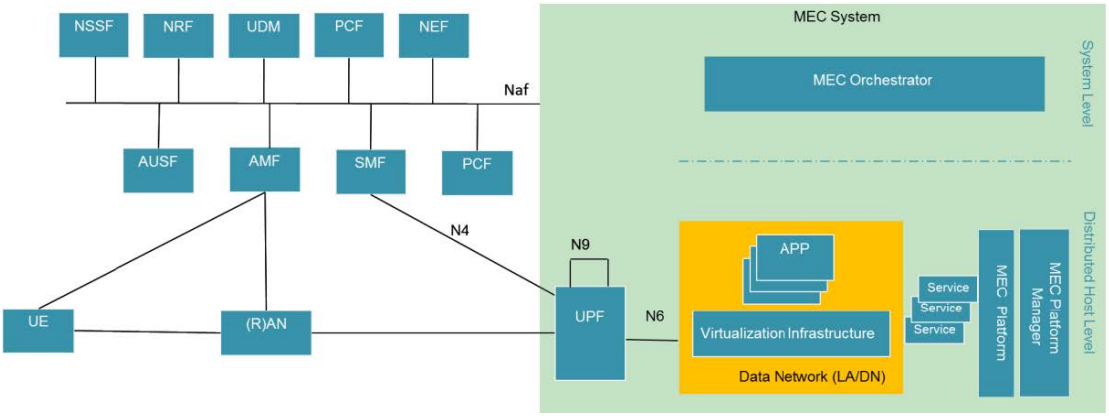


Figure 1.3: MEC infrastructure on 5G [11]

The User Plane Function (UPF) can be seen as a distributed and configurable data plane from the MEC system perspective.

A MEC application may belong to one or more network slices that have been configured in the 5G core network. In 5G is Network Slicing, that allows the allocation of the required features and resources to different services or to tenants, is handled by the Network Slice Selection Function (NSSF). NSSF also assists in the allocation of the necessary Access Management Functions (AMFs).

In a 5G system it is the AMF that handles mobility related procedures and it is responsible for the termination of RAN control plane. The Session Management Function (SMF) includes session management, IP address allocation and management, DHCP services, selection/re-selection and control of the UPF. As MEC services may be offered in both centralized and edge clouds, the SMF exposes service operations to allow MEC as a 5G AF to manage the PDU sessions, control the policy settings and traffic rules as well as to subscribe to notifications on session management events.

The procedures related to authentication are instead served by the Authentication Server Function (AUSF). Policies and rules in the 5G system are handled by the PCF. The PCF can be accessed either directly, or via the NEF.

Brief comparison: Among the conceptual models for Mobile Edge, Fog Computing offers more flexibility in the choice of devices. It leverages on legacy devices by adding storage and processing to them, however the overall capacities are usually lesser than that of Cloudlets or Mobile Edge servers [19]. Since dedicated

devices are used as Edge, MEC and Cloudlets require more time and effort to setup the infrastructure, but reduce heterogeneity of systems. MEC also augments context awareness compared to Cloudlets since it gives access to Network information. Implementation differences are summarized in Table 1.2 . In our work, when we refer to Edge Computing, we imply Fog Computing and MEC.

Table 1.2: Comparison of Edge Implementations

	Fog	**MEC**	**Cloudlet**
Devices	Routers, Switches, AP, Gateways	Servers on base stations	Data Centers "in a box"
Location	Varying between UE and Cloud	RN Controller/ Base Station	Local/Outdoor installation
SW Architecture	Fog Abstraction Layer	Mobile Orchestrator	Cloudlet Agent
Context awareness	Medium	High	Low
Network	Bluetooth, Wi-Fi, Mobile Networks	Mobile Networks	Wi-Fi
Node to Node	Can communicate	Partial comm.	Partial comm.

Industrial takes on Edge Computing

Edge Computing boxes can range from extremely mobile, low-powered devices (e.g. Tactical Edge Computing [29]) to racks of traditional servers (i.e. Cisco Flexpod Express [30]).

Mini clouds have been defined as one or more clusters of computers within the same LAN, able to provide services such as web pages and storage [31]. Authors envision an integration of instances owned by multiple institutions to form a self-managed virtualized Cloud.

Micro and nano data centers are rack of lights-out servers, relatively small in size(on the range of a tenth of square meters), and managed remotely by a larger data center. Disruptive startup companies such as Vapor IO, EdgeConneX, and DartPoints have been deploying some [32]. Dimensions vary from small, regional, and micro-regional data centers. Typical locations are parking lots, municipal areas, and at the base of cell towers.

This are the first industrial attempt towards Cloudlets. One of the main promoters of the technology is MobileEdgeX [33], a company created by a Communication Service Provider, Deutsche Telekom [34]. Other examples of Cloudlets in this cathegory are Network Attached Storage (NAS) units with Cloud synchronization (e.g. Synology's CloudSync [35]).

Distributed Cloud is promoted by Ericsson [36], one of the most prominent telecommunication vendors, as a distributed cloud infrastructure hosting telecommunication application runtime components as well as designed for 3rd party applications. Distributed Cloud is an execution environment for applications over multiple sites, including connectivity managed as one solution. Edge computing is here envisioned at enterprise premises, for example inside factory buildings, in homes and vehicles, including trains, planes and private cars. The infrastructure can be managed or hosted by communication service providers or other types of service providers.

1.2.2 5G Based Execution Infrastructures

The more frequent usage of mobile devices to perform and compute-intensive operations or storing a vast amount of data introduced the need to offload tasks or data to the clouds to achieve better performance and extend battery life [37]. The new generation of mobile networks (i.e., 5G) will be characterized by network densification in spatial, spectral, and back-haul dimensions. Mobile Edge Computing (MEC) [38] is a network architecture that suggests deploying cloud servers in each base station of the mobile network to reduce the distance between end-users and Cloud. Cellular networks will be using C-RAN [39], SDN [40], NFV [41] and MIMO [42] technologies and maybe even involve full duplex or device to device connections. As explained in the Europeans actions plans [43], 5G will be characterized by:

1. Economic fiber-like radio with data rates beyond 10 Gb/s, using higher frequency bands above 6 GHz.
2. Network Function Virtualization (NFV) to allow implementing specific network functions in software running on generic hardware, effectively reducing implementation, management, and operational costs and allowing reuse.
3. Software Defined Networking (SDN), to allow the control of network resources to be opened to third parties.

It was already stated in [44] that: "Future mobile terminals will have much improved context awareness, with knowledge of the user's requirements, the surrounding environment, and the network." Core 5G applications requirements that are out of the current 4G technology's abilities will be:

1. low latency of 1ms (10 to 20 ms for 4G),
2. serving 1 million of devices/km^2 (about 1000 device/km^2 for 4G)
3. fast deployment of new services in 1 hour time (done in days with current technology)

1.2.3 Edge computing Standardizations and architectures

Telecom operators are promising millisecond range latency for 5G networks. By the laws of physics, this is only feasible if the serving application requests are geographically close enough to the mobile equipment. However, as pointed out in [45], it is not only the low latency itself that drives network architecture towards edge computing but also the ability to offload the device and therefore save on battery.

In order to enable the portability of applications through countries and mobile operators, telecommunication standardization bodies are working on solutions to support edge computing in networks.

- **ETSI Multi-Access Edge Computing (MEC)** ETSI efforts towards standardization of edge computing are translated into a series of specifications. The standard proposes integration options to mobile networks and introduces a concept of a telecommunication specific edge computing platform that allows operators to open their Edge to authorized third-parties.

  ETSI MEC definitions In (GS MEC 012), the Radio Network Information Service (RNIS) is described as "a service that provides radio network-related information to mobile edge applications and to mobile edge platforms." It exposes information to applications per User Equipment (UE) or per cell or per period of time, such as radio network conditions, user plane statistics, and UE context.

  (GS MEC 013) Location Service API defines services for activities related to the location of the device (e.g., its geo-location or Cell ID), such as tracking or service recommendation.

  The location of single or multiple UEs associated with the "MEC host" is deducted based on the site of all radio nodes connected to that Edge.

  In later work, two API services were detailed:

  1. (GS MEC 014)UE Identity API tags the user's equipment (UE) to map it to enable enforcement of traffic rules. Example usage [46] is to apply traffic rules to program the data path and redirect the traffic to a corresponding service provider, whether it's local or remote, to lower the latency.
  2. (GS MEC 015)API for the Bandwidth Management Service (BWMS), allows applications to register for the bandwidth allocation for sharing and flexibility of the resource usage. This service also allows more complex policies such as prioritization of certain traffic.

- **Edge Computing in 3GPP** Being the main standardization body in the mobile telecommunication sector, 3GPP has its own tracks for standardizing edge computing since release 15 [47]. The work inherits some concepts from ETSI MEC but specifies the details in a way that the solution is optimized for 5G mobile networks. 3GPP defines the separation of the control plane and the data plane, network slicing, and so-called Local Breakout functionality to enable breakout points closer to the user devices and proposes methods describing how the user devices may found the application.

1.2.4 Edge-native Applications

In this thesis work, I am introducing the new concept of edge native applications. The term is a relatively new concept, I am defining it with a detailed characterization.

Initially, ETSI MEC [48] classified applications running at the Edge into three categories: device-centric (e.g., online gaming), RAN-centric (e.g., caching), and information-centric (e.g., content optimization). 3GPP does not specify applications for Edge, besides the network functions that are an integral part of the network, it differentiates between so-called trusted applications that are entitled to manipulate specific network configurations through Network Exposure Function and untrusted 3rd party applications that are running over the top.

The term *Edge-native* has been applied only later, in December 2019, by the *The Eclipse Foundation* to refer to their new vendor-neutral Working Group.

The definition is inspired by the term *Cloud-native*: those applications born segmented into microservices with the intent to increase agility and maintainability. Each part of the stack (applications, processes, etc.) is packaged in its own container, allowing for reproducibility, transparency, and resource isolation. Containers are actively scheduled and managed to optimize resource utilization (Orchestration).

The group intends to provide an end-to-end software stack to support deployments on top of Edge resources of applications related to the Internet of Things (IoT), Artificial Intelligence (AI), autonomous vehicles, and others.

In their announcement [49] Edge computing is stated to be not necessarily different from Cloud computing in terms of computational power or software stack (e.g., containers, Kubernetes, and microservices). The significant difference stands in the fact that, in Edge Applications, we care about the location of the used resources and the transparency of the orchestration.

In the recent works on Edge computing, few authors have also introduced this term [50, 51].

Authors in [50] define an *Edge-native* application as any software custom-designed to take advantage of one or more of the attributes of Edge comput-

ing: bandwidth scalability, low-latency offload, privacy-preserving denaturing, and WAN-failure resiliency. We do argue that privacy-preserving is not a characteristic intrinsic to the Edge. Even though adding an extra tier to the network allows data aggregation, and reduces the amount of data that gets to the Cloud. In fact, security and privacy could potentially benefit from the Edge. However, they are still highly dependent on the application implementation. Nothing forbids an application running on the Edge to log in clear every data sent to it. In fact, the security aspects of Edge Computing are out of the scope of this work.

In [51] an application is *Edge-native* not only because it is deeply dependent on services that are only available at the Edge; but also because it is written to adapt to scalability-relevant guidance. This implies that the application should be able to scale up and down, not only horizontally (ex. growing resource usage on the same machines/tiers) but also vertically: moving from a device/Tier-3 to Edge/Tier-2 to Cloud/Tier-3.

In the following text, we expand on this definitions identifying five common characteristics for *Edge-native* applications (as summarized in Figure 1.4): *Cloud-native, Edge Driven, Mobile, Geo-Localized* and *User Equipment Dependent*.

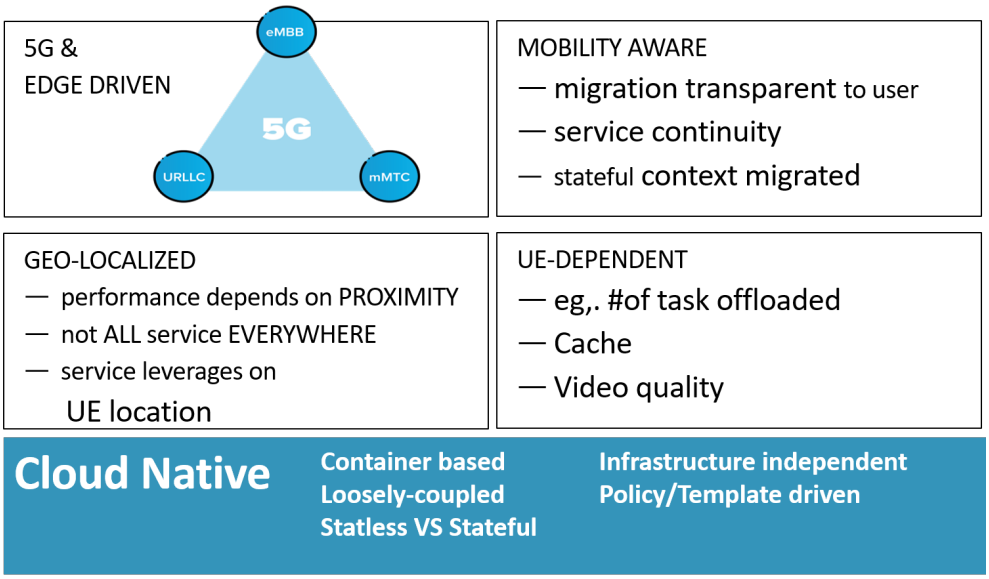


Figure 1.4: Characteristics of Edge-native applications

Cloud native

Edge-native applications are a subset of Cloud native applications and preserve their features.

Container based: self-contained, independent applications; packaged in lightweight containers.

Loosely-coupled: designed as Microservices. Where each exists independently from others but can discover them dynamically to allow for integration, in other words, each edge-native application should be easily represented in a dependency graph, where the nodes express a service and the edges the dependency with a different subset of services.

Strongly separated between stateless and stateful: state independent services are privileged. However, persistence can be enabled, relying on a separate service. (For example a shared database, a cache service, etc.)

Infrastructures independent: their abstraction level is higher than the OS, allowing smooth migration. However, they may specify some required system capability (e.g., GPU).

Policy driven: they can be configured to be highly automatable and elastic, thus easier to scale.

Edge Driven

Applications require at least one of the three vertices of the 5G triangle: greater data-bandwidth, ultralow latency (network proximity), or massive/continuous communication.

Greater data-bandwidth: they are immersive applications such as Premium HD, 360 degrees views, 4K video, or any multimedia that can benefit from cashing and trans-coding.

Ultralow Latency: Real-time application where communication is critical. Industrial IoT applications for monitoring and time-critical process control such as smart grid switching of power, triggering alternative energy supplies, or fault detection applications. Precision farming. Applications that are sensitive to latency variation (jitter): AR/VR and other forms of tactile Internet, smart vehicles, and their V2V or V2I communication, Industry 4.0. Also, smart cities, primarily emergency-related applications: traffic safety and control systems, hazard warning and cooperative autonomous driving, Healthcare applications like remote surgery or remote abdomen scans.

Massive communication: Those applications where sending billions of events or data to the Cloud would be expensive and inefficient. Examples are: analysis of raw video streams for video surveillance, face recognition or object recognition such as plates or pass stickers; IoT gateways, big-data analytic collections.

Mobility aware

Applications are designed to follow the user or to be subjected often to redeployment and scaling horizontally and vertically, meaning that, Edge Native applications have the ability to scale out to the Edge or scale back to the Cloud.

Mobility awareness also helps edge-native applications to prevent or react to catastrophic failures since Edge sites may suffer single points of failure in their network and power supplies.

Application mobility requires transferring and synchronization in order to guarantee service continuity to the final user—the added complexity to the infrastructure changes according to the type of application mobility. The simplest scenario is if the application is stateless so that there is no need to store and move its context.

Stateful applications increase complexity since they require moving and seamlessly synchronizing the context to make the migration transparent to the user.

Geo-localized

The software is highly dependent and reacting to the surrounding environment. Not only the performance of the application depends on its network proximity to the services, but also the service leverages on inputs dependent on the location of the user. An example is ad delivery and footprints analysis in shopping malls. The main priority for this applications is to provide high defined personalized content without burdening the upstream bandwidth. These are geographically self-contained services like those for stadiums, airports, concerts, universities, or any smart buildings. Applications that allow the viewers to perform the same action from different perspectives based on their personal preferences. For example, having multiple views of a soccer field during a match. Live camera signals are locally ingested and played out to visitors in real-time. Visitors can select between different cameras for an immersive experience.

User Equipment Dependent

Applications, being either human-centric or event-centric, have most of their interactions influenced by the UE. The Equipment characteristics alter the performance of the application (ex. video input and connection quality, amount of messages that can be handled in the time unit, memory available to perform tasks offloading or preprocessing, etc.).

An example of edge-native application covering all the presented properties are Augmented Reality based Massive Multiplayer Online Games.

These are Cloud-native applications that are dependent on all three vertexes of the 5G triangle. It is mandatory to enable UEs to interact with each other in real-time: time of all servers and clients should advance at the same rate to ensure as close as possible experiences (**ultralow latency**). They render very high-quality images and videos (**great data bandwith**) and involve continuous data transmission from the gamer controller (**massive communication**). Clients are **mobile** devices and need a **stateful** context: they should share the same application state (temporal and spatial consistency) .

The rendering of the backgrounds, maps, and other surrounding user avatars depends on the user location (**Geo-Localized**). Users may access the games from different equipment/mobiles but have to get the same chance of participation regardless (**UE Dependent**).

Chapter 2

Facilitate Deployment at the Edge

5G mobile networks promise high bandwidth and low latency on the radio interface for both downlink and uplink data [52], which capability will enable new types of applications and services. Nowadays, there exist very few edge-native applications, but their number will grow in the future. Such mobile applications include Augmented Reality (AR), Virtual Reality, Gaming, and many other bandwidth-heavy and latency-sensitive applications, potentially applied for critical use cases such as Intelligent Transportation Systems or Surveillance.

Deploying an application on a 5G network with distributed edge cloud capabilities involves, first of all, the choice of were to allocate what parts of the application. The decision depends both on the application itself and on the involved network.

In this chapter, we propose an offloading framework to allow components of edge-native applications to run not only on the user device but also on the Edge and Cloud sites. The aim is to give an overall impression of the problem, but we cannot cover the complexity of a working infrastructure solution for Edge-native applications. We will address the issues of scheduling, migration, and service provisioning in later chapters.

To support the change from Cloud to Edge, we also present a toolset to enable analysis and partition of an existing monolithic cloud application.

Finally, we provide a method and tool to facilitate the refactoring of an existing application into modules. To do so, we calculate the best placement of the modules on the network compute servers, given user profile and context conditions information. Common ones are the typical user request size, policy per type of user (SLA), available bandwidth, network node types, available computation power and cost (on the device, in the distributed Edge, and at the central Cloud).

We have chosen a resource-demanding AR application for our test. We assume that they can benefit both from involving low latency external computation power and significant sized, but affordable, storage capabilities. To validate our assumptions, we apply the mentioned tool and measure the application properties under

certain circumstances and network constraints.

Our main contribution is the proposal of a method to automatize application partitioning and placement in a 5G/Edge environment. We introduce a possible toolset implementing our approach, its experimental setup, and evaluation.

Most works on this topic focus only on task partitioning and placement, while they seldom address the question of handling multiple users and performing load balancing. For this purpose, in our work, we integrate an approach from network service placements and apply a variation of the approximation algorithm for the Path Computation and Function Placement Problem described in [53].

2.1 Motivation

Our first objective is to describe what characteristic the Edge infrastructure will have to enable the movement of applications trough different layers of the network. Secondly, we want to provide a first guide to enable the transformation from Cloud to Edge.

In most cases, developers rely on experience to make migration decisions from one technology to another, like what happened with micro-services and containers. However, Edge computing also involves new physical elements in the cloud infrastructure, for which experience only may not be enough. Thus the developer has to decide the best allocation and division of services without having previous references on the network and infrastructure influence on the overall performance.

While knowledge of the service category retains its importance in the migration process, a toolset to simulate such a process may save developers time and money. In fact, as for the Cloud, most of the Edge infrastructure will be pay-per-use, so that an unfortunate choice of deployment would translate in a significant waste of resources.

There is an extensive literature on Edge computing [54], partitioning [55] and code offloading [56]. However, from our knowledge, all the components of this complex system have been analyzed as standalone without considering too deeply their interactions.

The Chapter is structured as follows. After a deeper introduction to the objectives of modeling, partitioning and placing an application (Section 2.1.1), in Section 2.2 we present a classification of Application Partitioning and Offloading models. We also concentrate on the importance of context awareness and describe the requirements for partitioning of an AR application. Finally in Section 2.2.6 and Section 2.2.7 we present our first setup. Section 2.3 contains our model to partition an application and simulate multiple deployments and service requests. Together with modeling(Section 2.3.5) and network experiments(Section 2.3.6). In Section 2.4 we compare our work both with offloading and placement frameworks.

Section 2.5 summarizes our findings while 2.6 .

2.1.1 Problem Statement

To partition an application and deploy its modules in a 5G network, with edge computing resources, we need to calculate the (sub)optimal grouping of the components and their placement that maximizes network capacities in a given instant. Giving a flexible method to automate this process enables applications to adapt to environmental changes through dynamical reallocation of resources.

We can synthesize this task in three main steps:

a) Model the application through hybrid analysis (using both static analysis and heuristics from dynamic profiling of the given application);

b) Calculate a partition to divide the application in modules minimizing their interactions and communication cost while maximizing the responsiveness and perceived performances;

c) Decide best placements of the modules in the given network;

Model and Discovery The first step in finding the optimal way for executing a task should be profiling the context and the job itself. Profiling of a given task can be carried out in several ways. Fundamentally, this phase discovers locations of the code at which distribute or parallelize the execution. Having an enriched description or abstraction of the different resources available could help with integrating different vendors HW and SW and fastening this profiling. Developers may annotate an application to influence the result of the profiling and the consequent partitioning decision. An offline static analysis may help separate monolithic applications into candidates for tasks, such as identifying parallelizable or independent parts of the code and therefore, suggesting deployments on different machines. The scope of the context discovery is to build up a model or representation of an available network slice for which the subtasks' deployment is reasonable. In this analysis, we have to gain information on the costs of transmission, available resources, and the capabilities and current workload of reachable nodes in the network. A combination of static invariable knowledge and dynamic collection of this data through simulation and estimation models will be needed. For example, one could model measurements of the average consumption of the battery per instruction or task size, while total memory can be considered static data. Finally, the available resources on a node change during time and need constant monitoring. Based on previous works [55] we believe a graph representation of the task connections and the cost associated with running them in different

available nodes would be more performing and less resource-intensive than a linear programming (LP) model.

Placement Thus, the results of the Discovery phase should be applied to elaborate a sort of place-and-route graph, as a plan to deploy our subtasks in the available network. Making deployment plan is very similar to a *path computation and function placing* problem; already well-known task that must be solved by network manager in NFV/SDN settings. Even et al. in [53] described an approximation algorithm for addressing this problem efficiently and flexibly. Integrating this solution with methods for faster Pareto-optimal solutions may allow us to comply with a more strict real-time requirement. Further restriction to the slice of the network to consider may be posed by the application of clusters, as described in the previous section.

2.1.2 Continuous Interactive Applications

In this chapter, we will run experiments with a category of cloud applications that share some characteristics with the edge-native ones. In Continuous Interactive Applications (CIA) [57] the interaction is real-time and almost constant, so this time must be as close as possible to zero. Examples of such applications are augmented reality (AR) solutions or MMORPGs (Massive Multiplayer Online Role-Playing Games). The main requirement for such applications is to enable participants (even if in different locations) to interact with each other in real-time. All clients should share the same application state (temporal and spatial consistency) and have an equal chance of participation regardless of their network conditions (fairness). The simulation times of all servers and clients should advance at the same rate. The representation within a simulation should be as similar as possible to the real-world (fidelity). This general requirements are summarized in Figure 2.1

In these scenarios, latency is a known significant barrier. Furthermore, network latency has a lower theoretical limit imposed by the speed of light. We believe it is possible to reduce latency and to improve CIAs by applying computation offloading and by reducing physical distance between server and user through 5G and edge computing.

The emerging 5G network architecture, together with the concept of edge computing seems to be the perfect solution to the requirements of CIAs [58]. State of the art 4G networks can add from 10ms to more latency to CIAs, but the future 5G network can be expected to be able to support real-time and context-aware applications, being able to provide specific context information [59].

The mobile nature of CIA requires adaptive algorithms and context-aware architectures. Dynamic changes in the configuration at the Edge of the network can

happen frequently: devices may be physically moving, the network may need to balance resources or reallocate them. Therefore, our goal is to define a framework that takes into account the dynamic nature of CIA applications.

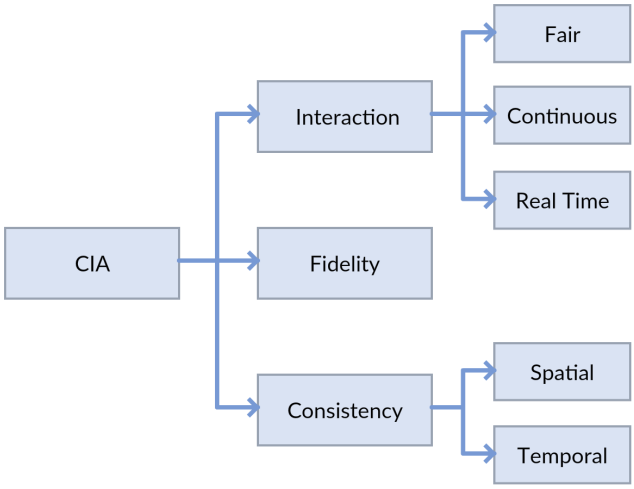


Figure 2.1: Continuous Interactive Applications General Requirements

2.2 Offloading in Edge Computing

The use of mobile devices, to perform intensive operations, or store a considerable amount of data, requires offloading to the clouds to achieve better performance and extend battery life [37]. These would be difficult and expensive without bringing the Cloud closer to the Edge of the network and the end-users.

One of the fundamental properties of an Edge Computing framework is computation offloading. In Edge offloading, resource-intensive tasks should be executed over the cloud infrastructure to overcome the resource limitation of mobile devices and reduce the total execution time [56].

Furthermore, at the Edge applications will travel trough all tiers: mobile device, Edge, and Cloud, according to the user and network.

Having at least two supported types of layers (the Edges and the Local clouds) and including context awareness to facilitate task or code offloading, increase the infrastructure complexity.

We describe our work towards a framework keeping in consideration such needs, together with the interactions of its components (see Figure 2.2).

The UE connects to an antenna site, where the Edge resources are co-located, and where it can get access to the internet and thus to Local Cloud resources. Generally we consider Local Cloud any data center close enough to the user, for

instance in the same area of his Region. The main idea behind the framework is to store multiple partitions of the same application, and switch among them based on the context.

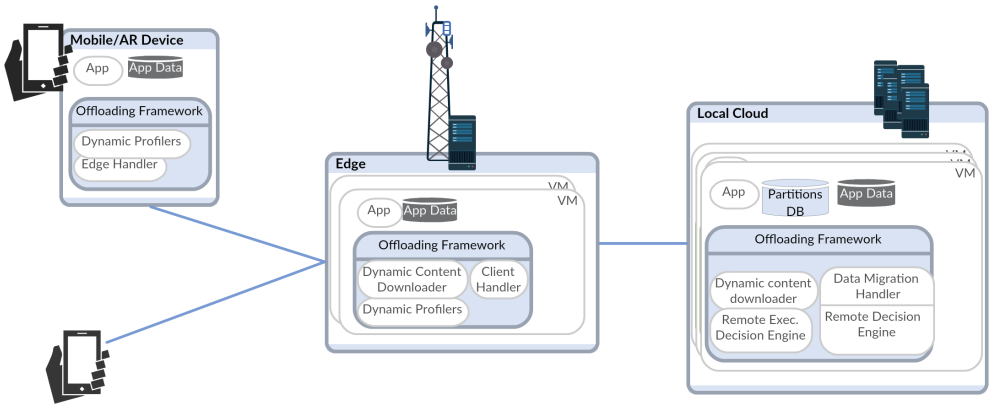


Figure 2.2: Context-Aware Framework for Application Offloading

In this sense, we also investigate how to provide a flexible partitioning of the application based on the location it has to be moved.

2.2.1 Application Partitioning

Partitioning an application means starting from a monolithic application and separate it in multiple communicating components.

Program partitioning has been used in application offloading for resource-constrained devices. Previous works propose computation offloading at different levels of granularity: Module level [60], Method level [61], Object level [62, 63], Thread level [64, 65], Component level [66, 67]. Various metrics can help to decide the right level of granularity for the partition of a graph. For instance, the critical path, being the longest directed path between any start and finish nodes, indicates the shortest time needed to execute. The time can be calculated from its length, computed by the sum of the traversed nodes' weights. The average degree of concurrency, that is, the total amount of work divided by critical path length, is also a standard metric. Related to the size of the partitions, we consider essential the size of the data associated with tasks because it helps to minimize the volume of data-exchange and maximize data locality. Also, the size of context is an indicator of how affordable or expensive the communication between tasks can be.

Another dimension that could be considered is the type of gathered information used by the partitioning. It can be a result of a static (based on the source code) or a dynamic (based on runtime information) analysis.

Applications may be partitioned using automatic or manual annotations.

The partitioning analysis can be performed using a graph-based or a linear programming (LP) model. According to [55] graph approaches perform better, consume fewer resources, in comparison with LP.

State-of-the-art Application Partitioning Algorithms (APAs), applied to distributed processing, still face many issues and challenges. An extensive summary concerning APAs in Mobile Cloud Computing is proposed in [55]. Based on the model used as an input to the partition, the authors identify three main categories of solutions: graph-based, Linear Programming (LP), or hybrid solutions between the two.

Solutions based on graph representations of the applications may use a data flow graph to represent data dependencies between operations [68, 60, 65],

while class dependency graphs can describe the structure of an application [69, 68].

Authors in [62] partition object-oriented programs by generating an Object Relation Graph (ORG) to estimate the runtime objects and their interactions, and then applying graph partitioning to this ORG. In [63], a two-layer graph structure is used, in which a second graph, the Target Graph (TG) accounts for the various target infrastructures and distribution objectives.

Graph-based APAs require efficient manual annotation techniques; it is up to the programmer to balance the metrics and specify metrics function. Besides, a tremendous resource overhead is generated in case of applications with a large number of components. Finally, the performance of graph-based solutions depends on the application characteristics: the analysis becomes easy if the application is already somehow modularized. On the other hand, LP-based solutions always produce optimal results for a particular objective function [66, 70, 71]. LP APAs need dynamic scheduling techniques, extra profiling, and resource monitoring; thus, causing high overheads.

Hybrid solutions extract the important features of graph-based APAs and LP-based APAs to improve the performance and mitigate overheads but, in most cases, at the expense of generating only a sub-optimal partition [61, 64, 65, 72, 67].

Application Modeling and graph partitioning

The NP-hard graph partitioning problem is a fundamental issue in many other domains of computer science, such as parallel processing [73] and load balancing [74]. In grid computing, the graph partitioning problem has been used to define parallel tasks to be deployed on heterogeneous infrastructures. As stated by [67], many proposed algorithms, such as MiniMax, VHEM, QM, PaGrid, and MinEX, use a multilevel paradigm, while others use simulated annealing [75].

In the literature, decomposition techniques based on graphs [76] involve three

macro steps: (1) Identify the level of granularity for the elements of the partition; (2) Analyze the application with task dependency and interaction graphs, (3) Map possible valid partitions.

Properties of tasks that affect the quality of mapping are feasibility of task generation, size of tasks, and volume of data handled by the task or passed between two of them.

In fact, one needs to take into consideration the interaction between the partitioned tasks: they often share data and may have a correct sequential order [77, 78].

In scheduling, the interaction graph represents the application divided into tasks. Nodes in the graph are the tasks while their weights denote the amount of work to be performed. Edges represent the interactions between tasks. Generally, edges are undirected; when directed, they are used to show the direction of the dataflow (if unidirectional). Weights on edges contain the cost of communication. Shared data may imply synchronization protocols (mutual exclusion, etc.) to ensure consistency.

In distributed systems theory, the interaction graph is also referred to as the Control Flow Graph (CFG). A CFG is a representation, using graph notation, of all traversable paths of a program state during its execution. The graph provides the whole structure of a program, among others, making explicit all of the paths that are induced by a conditional branch. A function dependency graph, for example, is a sub-graph of these graphs, having has partition granularity the function. Dependency between functions implies interaction (calls or data passing) between them.

A Call Graph (CG) is a dependency graph representing calling relationships between functions in a computer program. Each node denotes a procedure and each $edge(f, g)$ indicates that procedure f calls g. Thus, a cycle in the graph indicates recursive calls.

Call graphs are results of basic program analysis, that can be used for model programs, or as a basis for further investigations. Call graphs can be dynamic or static. A dynamic call graph is a record of one execution of the program, for example, as output by a profiler. Thus, a dynamic call graph can be exact but only describes one performance.

In object-oriented languages, the potential target method(s) of many calls cannot be precisely determined solely by an examination of the source code [79].

Thus, to build the call graph, it is necessary to have inter-procedural data and control-flow analysis of the program.

2.2.2 Application Offloading

The decision about whether to offload part of an application depends on a complex combination of factors: the size of the task, the available resources, their distance,

the actual latency, and the cost of the code, state and data migration itself in terms of energy consumption.

The decisions could be taken either online or offline. The difference being in the higher computational overhead of online approaches.

Though the technical problem of offloading is well addressed in the current efforts [56], the application partitioning is still challenging due to the variation of resource requirements in heterogeneous mobile devices. An ideal solution should consider which granularity to choose for different mobiles and edges. It should adapt to changes in the network, without creating overhead and interfering with the application behavior. Our solution aims to solve this, as well.

2.2.3 Context Awareness

Adapting an application behavior based on the continuous changes of certain context variables, such as user location and resources (processor, network bandwidth, computational power, etc.), has been of interest in multiple works in the past years. We may distinguish between application adaptation, traditionally concerned with variability in hardware resources, and context-aware applications, more related to the user and the physical environment in which he/she is [80].

Context-awareness works include context management, context modeling, context measurement, and context delivery. Which respectively refer to: how context and relationships are modeled, how context is obtained, and how it is delivered to the engine responsible for the adaptation [80].

Context-awareness and application adaptation are critical enablers for autonomous systems [81]. IBM autonomic computing architecture states that they require self-management, self-configuring, self-healing, self-optimizing, and self-protecting.

For [82], the key challenges for application-level automation include models, frameworks and middleware services that support the definition of autonomic elements, the development of autonomic applications as the dynamic and opportunistic composition of these autonomic elements, and the policy, content, and context-driven definition, execution and management of these applications.

Works on autonomic computing address some of these self-management capabilities applied in specific domains [83]. Software deployment, data storage, resource allocation, communication patterns are only a few examples.

Since for our work, we concentrate on enhancing the AR technologies through self-optimization of the application offloading, we will shortly review the work in this direction.

2.2.4 AR Requirements

A mobile device user may offload code or data to a dedicated instance in the Edge directly. The mobile device communicates with the instance using a cellular network via the Internet. Because of mobility, device users may lose connectivity to the Edge, but might later reconnect to the same or a different one. In these cases, the offloaded code and data should not be lost.

Mostly, the need for a latency lower than what the human eye can perceive (25ms), pointed us toward a hybrid partitioning, using a base-pool of partitions created offline, and an algorithm for the online selection based on the graph representation of the method level costs. The partitioning approach will imply running the application on a small set of inputs to create a database of possible partitions varying according to network and hardware resources availability. Measuring through light profiling techniques will help to reduce the overhead. When the setup threshold is passed, the new partitioning will be deployed. Logs that do not tamper with applications can be used to allow a periodical offline analysis of the partitioning performance. This way, the database can be updated. We decide to focus on a module level or a method level partitioning, and object-level partitioning was excluded because of the general dimensions of OpenCV objects in the AR applications.

2.2.5 Offloading Framework

Based on the previous assumptions, we propose an offloading architecture that takes inspiration from [84] and [85] (see Figure 2.3). It distinguishes two supported types of layers: the Edges and the local Clouds.

An *Edge* has virtual machines (VM) instances hosted by a public cloud service near the mobile devices. Every application is assigned to an instance for offloading and caching purposes. Instances can communicate with each other and with the local Cloud. A cloud is a VM hosted by a resource-rich machine placed far enough from the Edge.

The *Edge handler* on the user device queries the nearest mobile base station for the most adjacent Edge. If one is available for connection, connects using a public IP address.

The *Dynamic content downloader* downloads input data for tasks when a mobile device offloads computing tasks through the Internet, so that the mobile device does not have to transmit input data with the code, which saves energy.

The *Data migration handler* manages data transmitted between a mobile device and its instance on the Edge. The mobile device connects to the client handler on the Cloud, and the dynamic migration handler connects to the dynamic object input/output stream on the Edge and the mobile device. When the mobile device

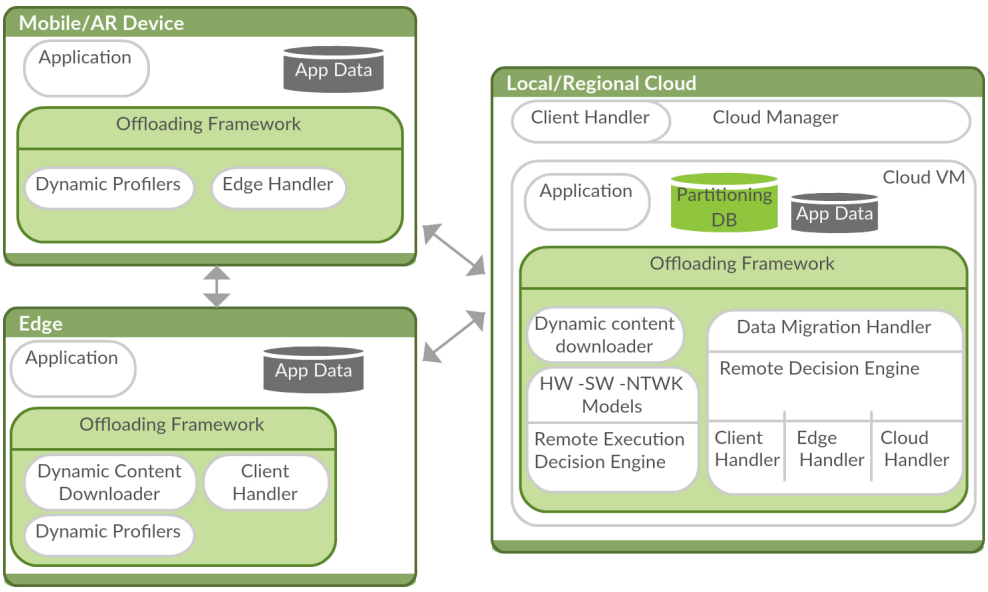


Figure 2.3: The proposed offloading framework

connects to the next nearest Edge, the new Edge handler attaches to the handler of the previous Cloud to retrieve the code/data. If the old Edge does not receive user data requests after a short time, it transfers the data back to the Cloud to prevent data loss.

The *Remote decision engine* decides on whether to migrate unfinished jobs with their data to the Edge or not.

The *App Data* caches frequently offloaded codes with application input data.

The offloading framework in the Cloud is placed in a VM hosted by a virtualization platform. This is the *Cloud VM*.

The *Remote execution decision engine* is the decision-making component of the offloading framework. Whether and where to offload data, it is based on the selected partitioning and optimization algorithm. This engine takes the measurements provided by the profilers as inputs. At execution time, active profiles will be monitoring: network bandwidth, user device energy and resources, Cloud, and edge resources. As stated by [61], it does not make sense to complicate network measures: a simple TCP packet is enough to judge latency. The main software measures are the time spent per method and CPU resources required. The main hardware ones are available battery and memory on the device and Edge.

The *Partition DB* leads us to a hybrid solution. A set of preferable partitions

based on thresholds of the profiled metrics can be stored in the DB. Every time the profilers register significant changes, the best fitting partition will be implemented on a new instance, and there will be a handover between the two.

2.2.6 Toolchain for Context Analysis

In our work, we plan to apply both static and dynamic tools. Static analysis can be used to discover dependencies on the application source code/bytecode. This analysis is quite lightweight and adds minimal overhead since it is done once and offline. The advantage of the dynamic techniques is to take into account any software and hardware and contextual network changes. The dynamic techniques use profilers, and they can optimize the application partitioning based on dynamic program analysis and context awareness. We used sample/log based profilers that are executed without modifying the binaries they measure.

The following tools are used: *Callgrind* [86], *OProfile* [87] to profile the system calls and CPU usage for a single process, and *CProfile* [88] to create graphical call graph.

- **Callgrind** is a static analysis and profiling tool. It was used to generate function dependency graphs and to record the call history among functions in a program's run (call-graph). The number of instructions executed, their relationships, the caller/callee, and the numbers of calls were registered.
- **OProfile** Was used to profile the system calls and CPU usage for a single process. The program offers a tool to check some information about the last profiling information, including CPU cycles and call graphs. The number of cycles can be combined with the processor information and provide hints about the energy consumption [89]. The profiler is composed of three main parts, (i) Kernel, responsible for the measurements itself, (ii) DataCollection, responsible for storing the measurement information and finally the (iii) Sample Database and Analysis that is responsible for computing and showing the profiling results.
- **CProfile** is a C extension of a Python module to profile software, it was used to generate graphical callgraph, profiling of CPU in nanoseconds grouped by Source File, cycle and by functions from called libraries. CProfile is mainly used for function-level performance analysis [88].

2.2.7 Preliminary Analysis

To support the proposed architecture and applying an offloading technique inspired from [67], we wanted to verify our toolchain and our ability to estimate the cost of

each method call (and thus possible partition) through hybrid static and dynamic analysis.

This experiments are taken from a collaboration work and are further detailed in [12].

Our simplified distributed 5G network consisted of two laptops simulating the Edge and the Cloud, connected via Ethernet cable to a Raspberry pi(as shown in Figure 2.4). The latency and speed are simulated in the application by software, where the minimal latency varies from half a millisecond to one and a half millisecond. This way, simulating the 5G environment mentioned by [18].

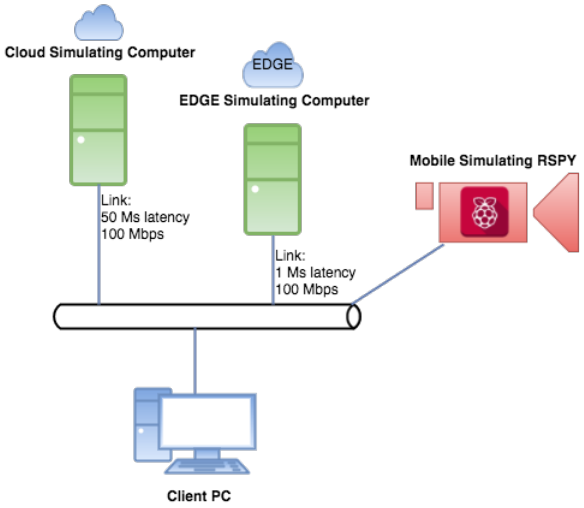


Figure 2.4: Environment overview [12]

We created a simple AR application based on OpenCV, a face recognition solution [90].

It recognizes all the faces appearing on a video stream and brackets them in real-time. The augmented video stream was rendered on the Raspberry pi, representing our augmented reality device.

The application was divided into four different modules, used to implement three different partitions, as explained bellows.

The *Video Decoder* receives a .jpg image from the streaming and decodes into an array readable by OpenCV. The *Face recognition server* receives a .jpg image from the streaming and does the full processing: decodes it into an array readable by OpenCV, performs the face pattern matching, draws a square around it, encodes back to jpg, and sends the image to stream. The *Video Encoder* converts an array of an OpenCV readable image into jpg. Finally, the *Face Recognition from encoded image* receives a decoded image, performs a pattern match, draws a square around these and sends the still decoded image for streaming.

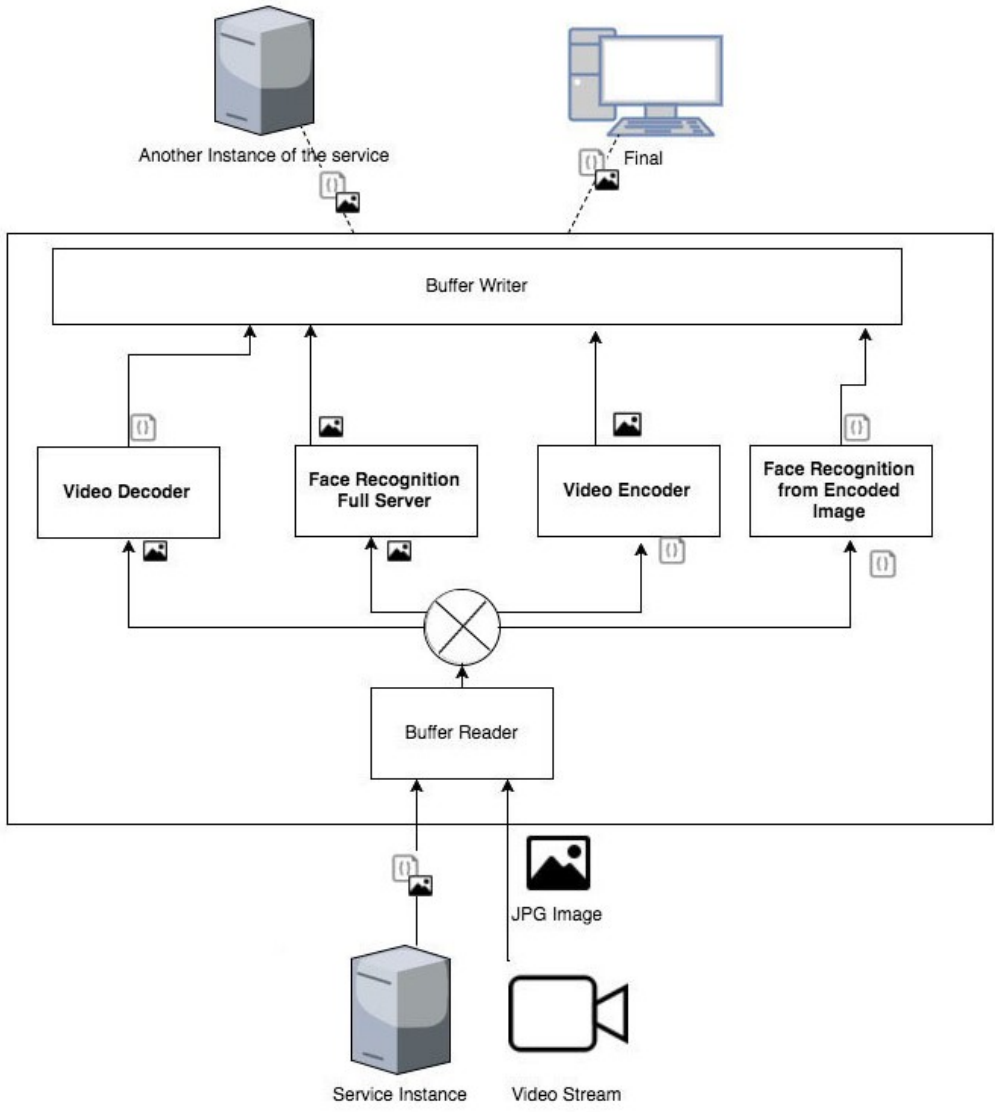


Figure 2.5: Server and services [12]

In this way, the application can be divided in more instances and different machines. An example of it can be seen in the Figure 2.5.

The possibility of splitting the library itself was not explored, but it is a strong possible further study, as some tasks may be done in the client side avoiding the data transmission.

The Client Application is able to read data from the server and display it, in two different fashions:

- **Encode and Show**: OpenCV decoded image are read from the buffer, and

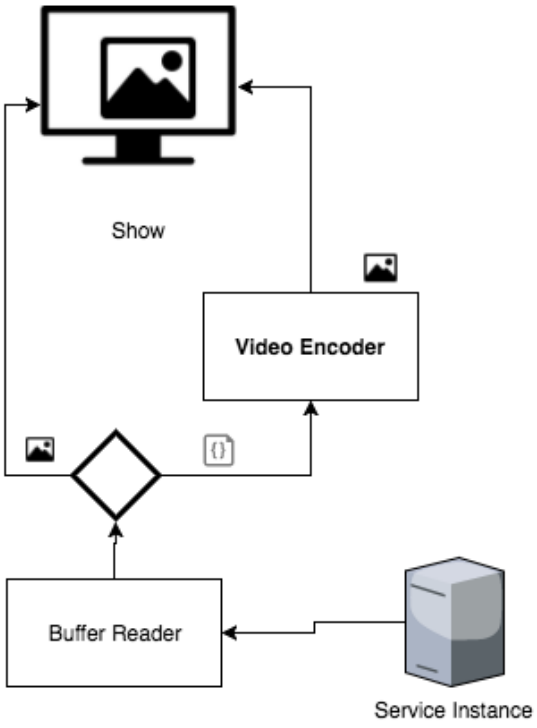


Figure 2.6: Client Architecture [12]

encoded into JPG, than shown on screen.

- **Show**: a jpg image, is directly read and displayed.

An example of how the Client Application works can be seen in the figure 2.6.

In this given scenario, the encoding performed by the client or the Edge, give us not just information related to itself, but the transmission of different data and the cost of the serialization process.

Given this, the three different partitioning scenarios where: (S1) the Edge performs all the activities, (S2) the Edge receives already decoded images from a mobile device, and (S3) the Edge receives the encoded image and streams an encoded image with the face detection.(As shown in Table 2.1)

An example of outcomes is shown in Table 2.2, which is a small sampling of the profiling data for us to better notice how the information is retrieved by the tool. The table shows the overhead for each of the five most expressive modules and five most called functions. The whole profiling covered an arc of five minutes (i.e. 5e+11 nanoseconds), equivalent to 20551547 processor cycles. Considering this, `libopencv_objdetect` took around 85.7% of the cycles in the first scenario (S1).

Table 2.1: Experiment scenarios

	RaspberryPi	**EDGE**	**Client**
S1	Streaming	Read Stream + Decode + Face Detect + Encode + Stream	Read streaming + Show
S2	Decode + Write Buffer	Read Buffer + Face Detect + Write Buffer	Read buffer + Decode + Show
S3	Streaming	Read Stream + Decode + Face Detect + Write Buffer	Read buffer + Encode + Show

Table 2.2: Edge Profiling

Functions Processing Time in Nanoseconds:				
	1st		2nd	
S1	detect_haar	59105000	sre_parse	14119997
S2	detect_haar	59724000	sre_parse	13461991
S3	detect_haar	59632000	sre_parse	14606995
	3rd		4th	
S1	___init___	10180922	sre_compile	6389996
S2	___init___	10563998	sre_compile	5780996
S3	___init___	10180994	sre_compile	6060998
Number of Processor Cycles per Module:				
	1st		2nd	
S1	libopencv_objdetect.so.3.2.0	17610014	libjpeg.so.8.0.2	986295
S2	libopencv_objdetect.so.3.2.0	5369080	libc-2.23.so	1495032
S3	libopencv_objdetect.so.3.2.0	4039863	libc-2.23.so	2078818
	3rd		4th	
S1	kallsyms	895637	libopencv_imgproc.so.3.2.0	378891
S2	python2.7	700766	kallsyms	507515
S3	kallsyms	284041	libjpeg.so.8.0.2	280537

This first example also shows us that, in the Edge simulations, the most used function always concerns the object detection.

The serialization algorithm is a built-in resource of Python called Pickle [91], which is widely used even for performance and hybrid approaches (e.g. [92]). It was responsible for around 1.5% of the added processing time for the client, which in our experiment was representing an average of 0.55 millisecond per frame.

In the Figure 2.7, we can see two graphs summarizing how this performance tests result for the Edge.

The two graphs on the top refers to a first 5 minutes batch and the two in the bottom to a second 5 minutes batch.

Based on the static analysis tool outputs, and adding an average offloading time (the communication cost between devices), we selected the first partition to be the most efficient. The key performance parameter was the number of calls and processing time in the face recognition activity, which is the same in all cases. When comparing S3 with S2, it processes a similar amount of frames with

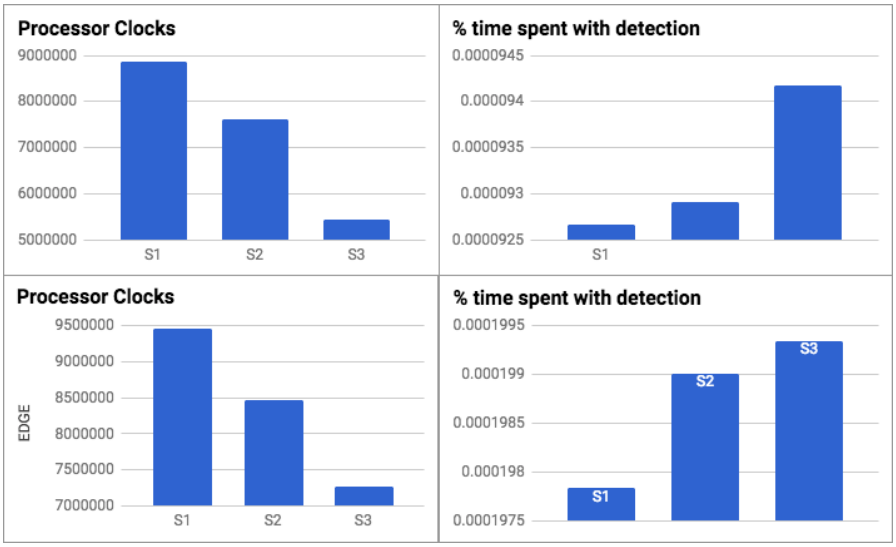


Figure 2.7: Edge profiling overview graphs

significantly better usage of Edge's resources. On the other hand, S1 performed, on average, three times more frames. In fact, on a 5 minutes streaming, for the same amount of total processor clocks, the number spent on performing detection is 85.7% against the 26% of S2 and 19.6% of S3. Offloading time cost was greater: in case of such a small application, the partitioning only adds overheads to the application performance.

2.3 A model for application partitioning and deployment

We decided to work on defining a toolset to assist the creation of edge-native applications based on the learning experiences on partitioning and offloading. The idea is to enable analysis and partition of existing cloud applications so that developers can understand how they could be deployed on the Edge infrastructure. In the following sections, we provide a formal description of the models and methods used to construct our simulation toolset, followed by a description of the real application we used as first input for it.

2.3.1 Models

To map an application based on functions granularity, we construct a function dependency graph. In scheduling and load balancing, this method is used when

the application can be described from the static definition of the dependency graph, and the function sizes are known.

Determining an optimal mapping of the function dependency graph becomes solvable if there are useful heuristics available to estimate the data flow and a structured call graph. We use a static analyzer tool to generate the function call graph from the source code. Then we run the application and collect for each function, using a non-intrusive dynamic profiler, the percentage of runtime spent in it. Besides, for each link between two functions, we collect the number of times the caller calls the callee. We normalize those results and store them in the call graph as a node and edge weights. The normalized edge weights will define the dependency between the two connected functions, thus to estimate how to separate the application to reduce such interactions, it will be enough to use a minimum edge-cut strategy. The node weight is useful information to determine the complexity of the computations handled by the function. This value can be used to balance the partitions or to deploy different optimization strategies. For example, if we want the User Equipment (UE) to save energy, we would concentrate the computational load on the Edge or the Cloud.

It is important to stress how we are deploying a context-insensitive construction of the application call graph. Each node will represent precisely a single contour: an analysis-time representation of a function.

2.3.2 Application Partitions

Having our weighted graph, we partition it using Multilevel [93] version of the Kernighan - Lin [94] algorithm (MLKL). We choose the multilevel strategy to be able to handle potentially large function call graphs. After running the algorithm, the application will be divided into several Modules where the user can select the ceiling for the number of partitions, and the weight of each Module and the interaction frequencies between them will derive from the original graph. The directed graph resulting after the partition step represents the due interactions between the modules. This new level of abstraction means that we lose information like when and for how long two specific functions in two modules will interact at running time. Such information also depends on the user interaction with the application itself and can vary from instance to instance.

We decided to adopt a pessimistic approach in the module deployment phase, taking as the weight of the assumption making that we want to run all the modules instantaneously.

MLKL and METIS

MLKL is a Multilevel Version of the KL algorithm. It means that the algorithm is applied in three repeated phases: Coarsen, Partition, and Uncoarsen.

First, the algorithm coarsens down the graph by merging connected vertices into a smaller one. Then this graph is partitioned and uncoarsened again while optimizing the partition in each uncoarsening step, using KL as refinement function.

The KL algorithm is iterative. It starts with an initial partition, and, in each iteration, it finds two subsets which guarantee a smaller edge-cut. If such subsets exist, then it moves them to the other part, and this becomes the partition for the next iteration. The algorithm continues by repeating the entire process. In the implementation proposed by [95], the KL algorithm computes for each vertex v a quantity called gain, which is the decrease (or increase) in the edge-cut if v is moved to the other part. The algorithm terminates when the edge-cut does not decrease after x number of vertex moves, and those last moves are undone to get the maximum edge cut.

2.3.3 Network Model

We test our partition behavior in our network to see what configuration gets the maximum out of the same network conditions. Thus we deploy all tasks in all nodes and see what are used the most to satisfy network demand considering network capacities.

The network is a fixed set of computational resources and communication links. It is represented by a graph $N = (V, E)$, where V is the set of nodes, and E is the set of edges.

We classify nodes into three categories: UE, Edge Cloud Servers, and Central Cloud Servers. Note that the classes are disjoint, and our proposed method works with other types of disjoint classification of nodes as well.

Nodes and edges have capacities. The capacity of an edge $e \in E$ is denoted by $c(e)$, and the capacity of a node $v \in V$ is denoted by $c(v)$. All capacities are positive integers. $c(e)$ represents the available bandwidth between the two network nodes; $c(v)$ depends on the amount of available computational resources and the cost of accessing them. We suppose several UEs that request services from the application. Each of these services may be different on the *Service type* and the *Location* of the involved nodes. Examples of such services can be a video upstream or augmented downlink video. Each Module is a part of the application that, combined, can solve a specific service request.

2.3.4 Service Request

A service request for user j is specified by a tuple $s_j = (G_j, d_j, b_j, U_j)$, where the components are as follows:

$G_j = (M_j, Y_j)$ is a directed (acyclic) graph called the place-and-route graph (pr-graph). There are a single source and a single sink, that corresponds to the node requesting the service. We denote the source and sink nodes in G_j by $ns_j \in M_j$ and $nt_j \in M_j$, respectively. The other vertices correspond to services or processing stages of a request. The edges of the pr-graph are directed and indicate precedence relations between pr-vertices.

The demand of a request s_j is d_j and its benefit is b_j. Demand is computed from the cost of running a complete module. The benefit is the benefit of serving that precise request of service. It should be calculated from the SLA, but it depends on the network owner as well. By scaling, we may assume that $min_j\{b_j\} = 1$.

We map the User Equipment service request s_j as the realization of a path through the directed partition graph representing the application. In this case, the Module's demand can be calculated over the cost of each function composing the Modules in the specific service request. The routing cost from one Module to the other becomes than the overhead or transmission cost brought by the selected Module interaction scheme. For example, the size of the data to be transferred from one Virtual Machine to the other to keep a consistent state trough all their network instances [96]. Thus, the impact of the service request on the network can vary only based on the Modules' location. To specify the possible realization of a pr-graph in the physical network, we use a function $U_j : M_j \cup Y_j \to 2^V \cup 2^E$ where $U_j(m)$ is a set of "allowed" nodes in N that can perform module m, and $U_j(y)$ is a set of "allowed" edges of N that can implement the precedences and routing requirement that corresponds to y. We now define for each service request s_j the product network $pn(N, s_j)$. The node set of $pn(N, s_j)$, denoted by V_j, is defined as $V_j \triangleq \cup_{y \in Y_j}(U_j(y) \times y)$. We refer to the subset $U_j(y) \times y$ as the y-layer in the product graph. The edge set of $pn(N, sj)$, denoted E_j, consists of two types of edges $E_j = E_{j,1} \cup E_{j,2}$ defined as follows:

1. Routing edges connect vertices in the same layer, they represent the physical links in the network.

   $E_{j,1} = \{((u, y), (v, y)) \mid y \in Y_j, (u, v) \in U_j(y)\}$

2. Processing edges connect two copies of the same network vertex in different layers, representing the move from one Module to the consecutive one in the service chain specified in Y.

   $E_{j,2} = \{((v, y), (v, y')) \mid y \neq y' \in Y_j$ edges with common endpoint m, and $v \in U_j(m)\}$

PCPF problem. The substrate network $N = (V, E)$ and a set of service requests $\{s_i\}_{i \in I}$ described as stated before, are the necessary input for the solution we used for Path Computation and Function Placement Problem (PCFP). The goal is to compute valid realizations $\tilde{P} = \{\tilde{p}_i\}_{i \in I'}$ for a subset of the requests $I' \subseteq I$ so that $\tilde{P}$ satisfies the capacity constraint of N and maximize the total benefit $\sum_{i \in I'} b_i$. For our work, we apply the fractional relaxation of PCFP-problem described in [53]. It is a variation of Raghavan's randomized rounding algorithm for general packing problems [97]. For each user request that we decided to supply, we assign a simple path Pi from its source s_i to its destination t_i. At each node, the random walk proceeds by rolling a dice. The probabilities of the sides of the dice are proportional to the flow amounts.

Algorithm 1. Algorithm for assigning a path P_i to flow f_i.

1: $P_i \leftarrow \{s_i\}$.
2: $u \leftarrow s_i$
3: **while** $u \neq t_i$ **do** ▷ did not reach t_i yet
4: $v \leftarrow$ *choose-next-vertex*(u).
5: Append v to P_i
6: $u \leftarrow v$
7: **end while**
8: **return** (P_i).
9: **procedure** *choose-next-vertex*(u, f_i) ▷ Assume that u is in the support of f_i
10: Define a dice $C(u, f_i)$ with $|\mathsf{out}(u)|$ sides. The side corresponding to an edge (u, v) has probability $f_i(u, v)/(\sum_{(u,v') \in \mathsf{out}(u)} f_i(u, v'))$.
11: Let v denote the outcome of a random roll of the dice $C(u, f_i)$.
12: **return** (v)
13: **end procedure**

Experiment Setup

We created a generic setup for Multi-Access Edge Computing partitioning and distribution. It is composed of four resource-constrained devices connected with an edge server through redundant networks, where different network setups can be tried. The application initially has all the processing activities done in the server, which collects information from the four connecting devices and performs the processing.

The connections used for the experiment explained in this article were carried with wireless 5 GHz and Ethernet connections, where the client devices were equipped with 100 megabits network shields.

The client devices were equipped with cameras using Sony IMX219 sensors, streaming real-time video to the server. The camera was configured to create frames of 640x480 pixels, 25 frames per second, and 4:3 aspect ratio. The connection between the clients and the server was via UDP.

Measurement Tools

The measurements used to configure the tool are grouped in three independent areas, namely: (1) Network performance, (2) Computational performance, and (3) Software processing cost. Each of them composed as described below:

- *Network Performance*: available resources, Jitter between nodes (Latency variation), Locality UE-Edge-Cloud;
- *Computational Performance*: Machine capabilities, Network connection speed, Processor capabilities, Memory availability;
- *Software processing cost*: Dependency between two functions (number of calls), Resource usage from App (Average memory Usage Mb per function), cost of the software execution (processor cycles that are required to execute each function of the software).

The software measurements were taken using instrumented profiling tools, Valgrind [98], and our self-produced tools.

Peer-to-peer communication is measured during execution using a network tool called "Packet Beat", which tracks the packets and connections of the server and send those to our repository.

2.3.5 Modeling the example application

In our experiment, we chose to start at a function level granularity and to partition the application into Modules. A typical AR application has the following chain of services: *capture, preprocessing, detection, recognition, tracking, rendering.* Each of these services calls a sequence of *Modules.* Note that these *Modules* may be different for different applications. Another example can be a partitioning of a Linear Unicast service, which may have the following modules: Streaming, Origination, Manipulation, Encapsulation, Encryption, Encoding, according to to [99].

In our first example (Figure 2.8), we show the result of running the partitioning only on the call graph of the *capture* service (involving camera calibration), where different colors refer to different modules and the number of requested partitions was 5. As a second example (Figure 2.9), we show the Function Call graph generated only by the camera calibration part of the application: on the edges the calls between functions and on the nodes the CPU clocks.

The result of the whole AR application partition is shown in Figure 2.10a. The *Start* node represents the interface with the User Equipment, the $Main$ node is the partition in which the known entry point of the program execution is located.

The arrows are the interaction between $Modules$. For example, we know that $Main$ can receive data and be called by $M1$, while, every call from the $Main$,

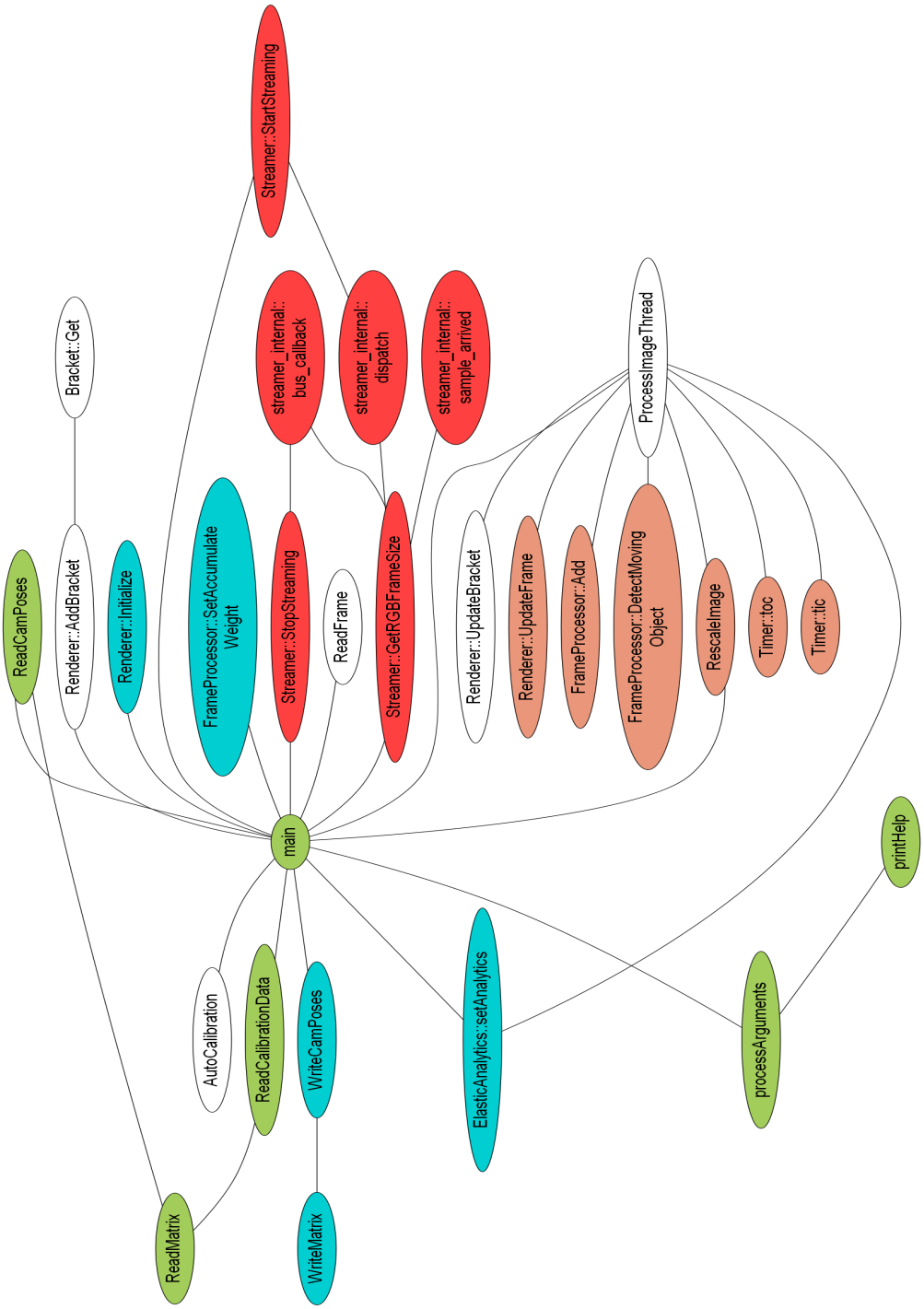


Figure 2.8: Partitioned Call Graph for an Image Capture Service

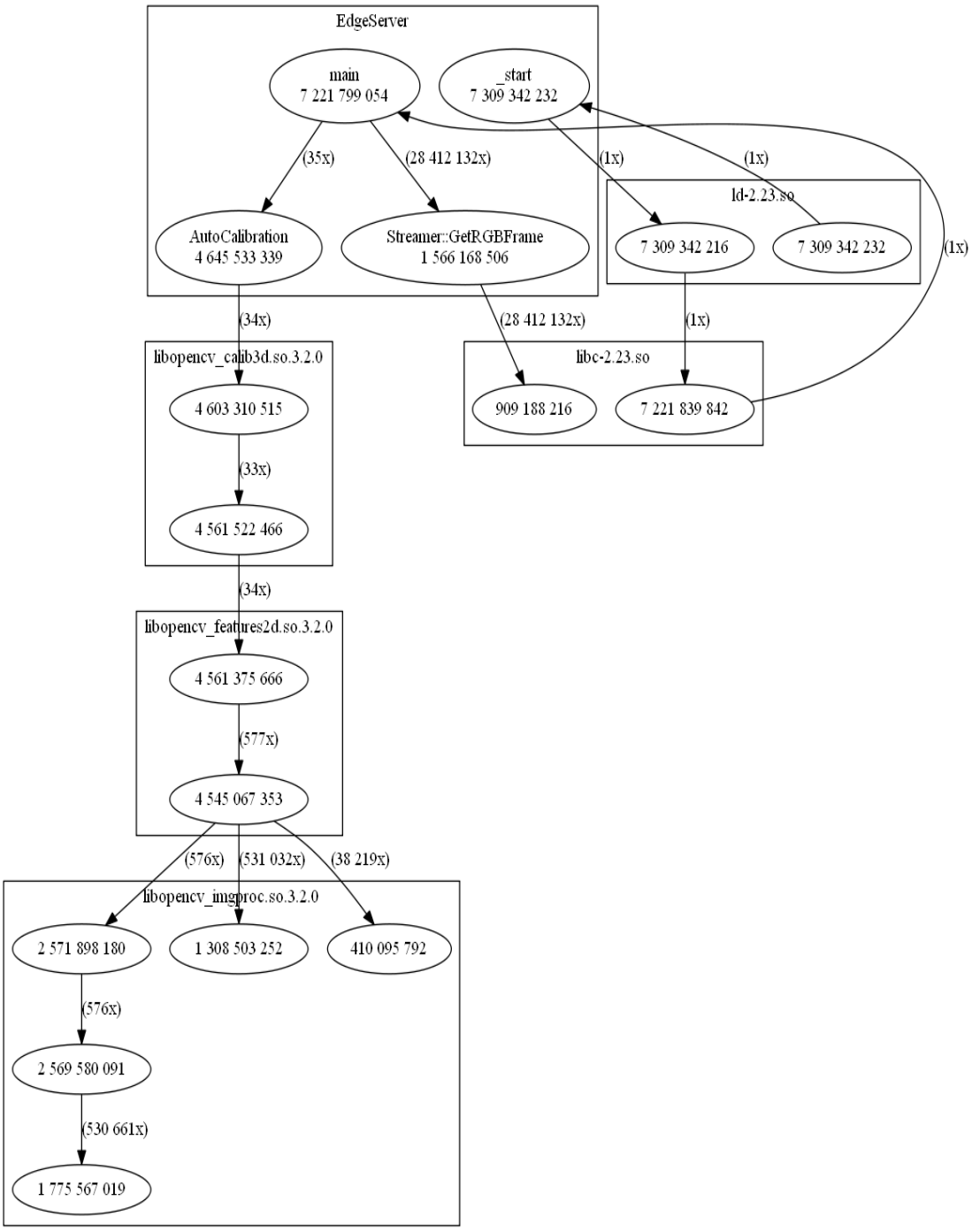
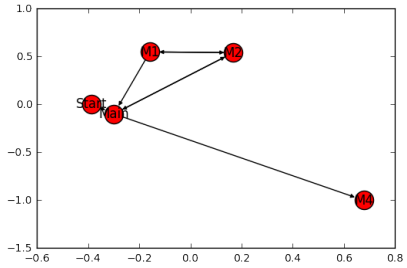
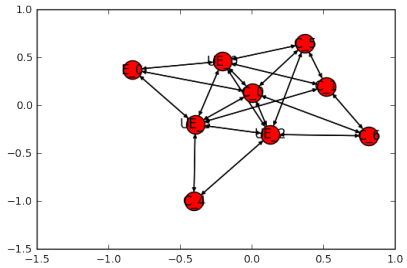


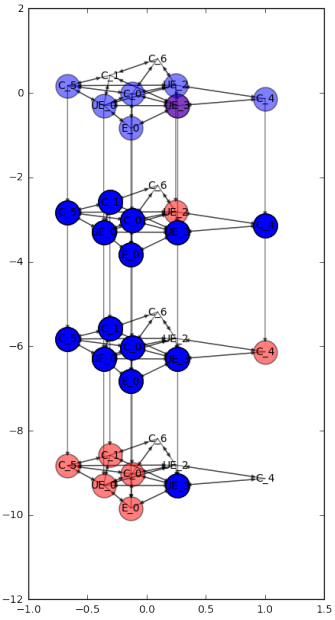
Figure 2.9: Camera calibration call graph



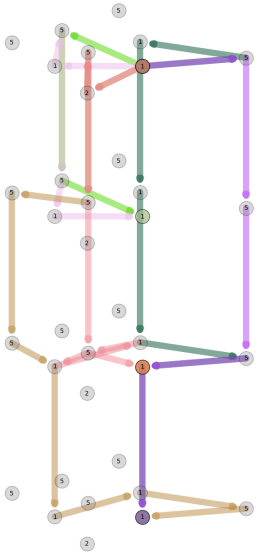
(a) AR Application after partition



(b) Simulation Network



(c) Complete AR application pr-graph



(d) Maximal benefit flows satisfying all user requests

Figure 2.10: Models of a real application deployed on an imaginary network composed of of Cloud, Edge and User Equipment nodes.

goes either to $M2$ or to $M4$. In the construction of service requests we kept the following interaction constraints: if the service needed by UE is in $M1$ the shortest possible request path became $\{(Start, Main)(Main, M2)(M2, M1)(M1, Main)\}$.

The Simulation Network will represent the possible interactions between nodes. In our simulation, we decided to allow both direct UE to Edge and UE to Cloud communications (Figure 2.10b). We consider different average transmission overheads: in the range of few ms (1 or 2) between UE and Edge nodes; 25 ms between Edge and Local Clouds; the sum of the two (26 or 27ms) between UE and Cloud. The resulting pr-graph is shown in Figure 2.10c. We normalize the capacities of the network nodes based on available memory. We experimented on a SLA scenario where we want to reduce the computation time at a minimum overhead.

For the same computation demand, we define the benefit of a chosen deployment path based on the computation cost (we estimated the Edge to be four times more expensive than the Cloud) and the average transmission overhead. Both weights were calculated as the coefficient of variation of the relative measures registered on the Experiment Setup.

In all the generated simulations, a deployment was proposed for which 12 con-

current simulated user requests where served, respecting the capacity constraints of different networks, obtaining maximal benefit flows like the one shown in Figure 2.10d. On average, the benefit was higher than running everything on the device: a complete run on the single device lasted on average 9444 ms, while the average run on our simulations saved from 667 ms up to 3904 ms with maximum average communication overhead per request being 1152 ms (Table 2.3).

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8
Edges	4	2	5	5	5	3	4	4
UEs	3	6	3	3	3	5	4	4
Local Cloud	1	1	1	1	1	1	1	1
Average overhead per request (ms)	37	58	144	29	1152	583	84	148
Average benefit per request (ms)	3941	1348	3743	3348	1820	2841	3340	2395
Average final benefit (ms)	3904	1289	3598	3319	667	2258	3255	2246

Table 2.3: Experiment result: benefits of partitioning and deployment of the same application on different networks topologies

2.3.6 Network Simulations

Later, in collaboration with the author of [100], we verified more extensively the performance of our deployment strategy. The simulation was this time performed within a realistic cloud computing environment called Edge CloudSim. Edge-CloudSim [101] is a cloud computing simulator based on CloudSim [102] that is desigend for edge computing.

Its main modules are:

- Mobility module: manages the location of edge devices and clients. Each UE has x and y coordinates that are updated according to dynamically managed hash tables.
- Task generator module: is responsible for generating network services. By default, it will randomly choose a mobile device as the device that generates and receives the services. Each service is generated according to Poisson distribution. Task inter-arrival time (load generation period), idle and active periods are configurable.
- Network module: simulates WAN and WLAN and handles the transmission delay. WAN is used for cloud communication. By default, it uses M/M/1 queue model for network transmitting. The number of arrivals is based on Poisson process; the service time is an exponential distribution; one server handles the transmissions.

Name	Augmented reality	Health app	Game app	Infotainment app
Usage percentage	30	20	20	30
Poisson interarrival	2	3	20	7
Active period	40	45	60	30
Idle period	20	90	120	45
Data upload	1500	200	2500	250
Data download	250	1250	200	1000
Length	12000	6000	30000	15000
Required core	1	1	1	1
VM utilization on cloud	0.8	0.4	2	1
VM utilization on edge	8	4	20	10
VM utilization on mobile	40	20	50	25

Table 2.4: Applications used to compose our services

- Edge orchestrator module: decides how and where the incoming requests go. This abstract class has been extended to apply our deployment strategy.

We compared its performance to a random resource selection. In Table 2.4 the applications used for the simulation are listed. These applications represent macro-categories of services(e.g. CPU intensive, high download, high upload, high communication rate...), and are simulated by the tool to recreate different usage patterns. For instance, the infotainment application requires less CPU power compared to the augmented reality application. The AR application requires high CPU, small amount of data to download, and bigger amount of data to upload. The tasks generated by the applications have random lengths in terms of the number of instructions and random input/output file sizes to upload/download [103].

EdgeCloudSim extends the functionality of CloudSim giving the possibility to describe: cloud devices, edge devices and mobile devices. In our case:

- Local Cloud: One data center and only one host with 4 VMs.
- Edges: 4 edge servers each composed of one host and three VMs(Xen X86 Linux, 16 cores, 80000 mips, 16000 ram, 400000 storage).
- UE: Each mobile device has only one host which serves one VM.

In Table 2.5 we compare the RANDOM and Product Network deployments for 100 to 500 UE requests.

We can see from Figure 2.11, that the network delay reduces adopting our policy, since it takes in consideration the Edge devices in terms of cost and benefit. The two algorithms can be compared also on the percentage of failed tasks. There are two reasons that may cause a deployment failure: the capacity of the VM is insufficient (too many tasks were allocated at the same time) or there have been network issues during the migration.

In Figure 2.12 we show how VM capacity is well handled in our policy as fewer tasks fail. In Figure 2.13 mobility, WLAN or WAN falures are shown.

Number of mobile devices	RANDOM		PN	
100	Total tasks: 25767	Cloud: 8480	Total tasks: 30871	Cloud: 824
		Edge: 8680		Edge: 9693
		Mobile: 8607		Mobile: 20354
200	Total tasks: 53724	Cloud: 17971	Total tasks: 48994	Cloud: 2140
		Edge: 17770		Edge: 12324
		Mobile: 17983		Mobile: 34530
300	Total tasks: 76786	Cloud: 25422	Total tasks: 82633	Cloud: 8916
		Edge: 25461		Edge: 16707
		Mobile:25903		Mobile: 57010
400	Total tasks: 103898	Cloud: 35156	Total tasks: 111663	Cloud: 11658
		Edge: 34931		Edge: 24116
		Mobile: 33811		Mobile: 75889
500	Total tasks: 134865	Cloud: 45646	Total tasks: 130362	Cloud: 7437
		Edge: 45379		Edge: 35151
		Mobile: 43940		Mobile: 87774

Table 2.5: Simulations runs with different amounts of UE requests

On average our policy has less number of mobility failures; for both strategies WLAN or WAN failures grows significantly when the number of UE is larger than 300. This is obviously more related to a physical upperbound of the network than to the model themselves. The product network deployment based on randomized rounding, seems to satisfy the Pareto-optimal performance that has been proven theoretically in [53]. This experiments batch, together with our previous simulation cannot be sufficient to justify an algorithm for an orchestrator, since the times of the simulations plays an important role in establishing the quality of an online deployment decision. However, they can be exploited offline, to assist in the conversion from Cloud-based to Edge-native applications. An analysis of the existing software, can indeed save time and money to developer, especially considering that most of the Cloud and Edge resources their applications will run on will be pay per use. Simulating different partition of the same service can, finally, help reduce risk of service failure and need for service replication. This concept is further explored in Chapter 4 of this thesis.

2.4 Related Works

2.4.1 Offloading Frameworks and Architectures

Most recent offloading works concentrate on frameworks for adapting and optimizing how to autonomously compose the partitioned application with some basic model for managing context delivery.

CloneCloud [64] concentrates on offload optimization and service availability but uses a simple partitioning scheme for any input. The limit of this approach is

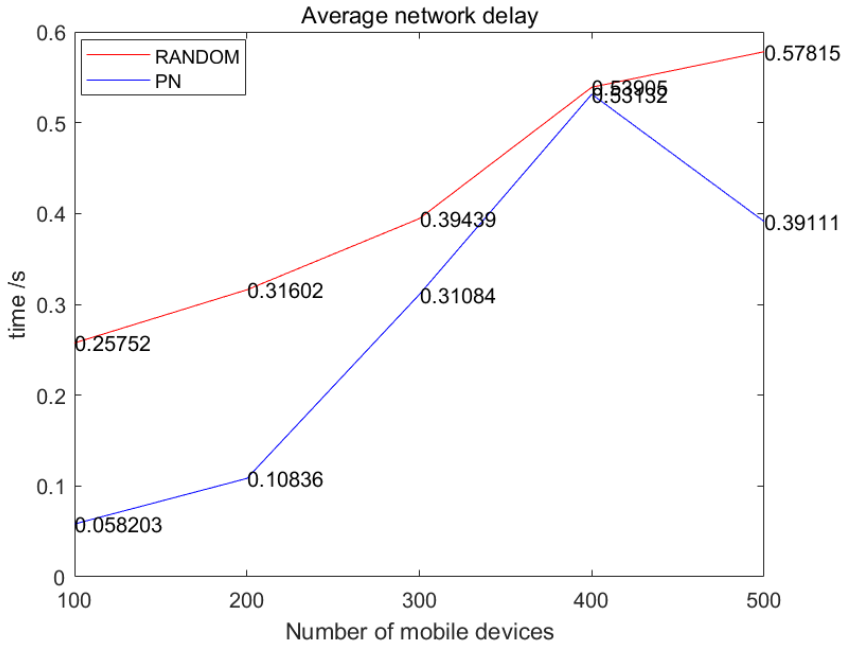


Figure 2.11: Network delay

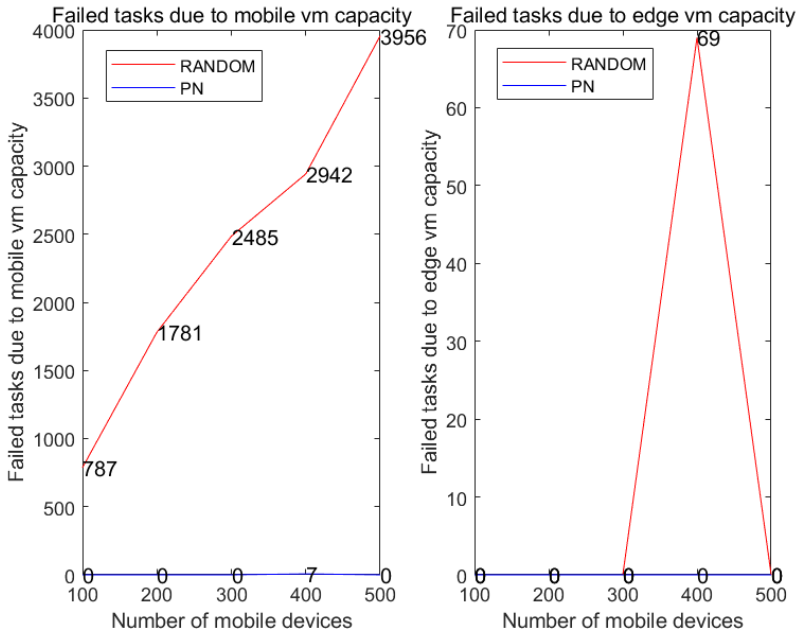


Figure 2.12: Failed tasks- capacity

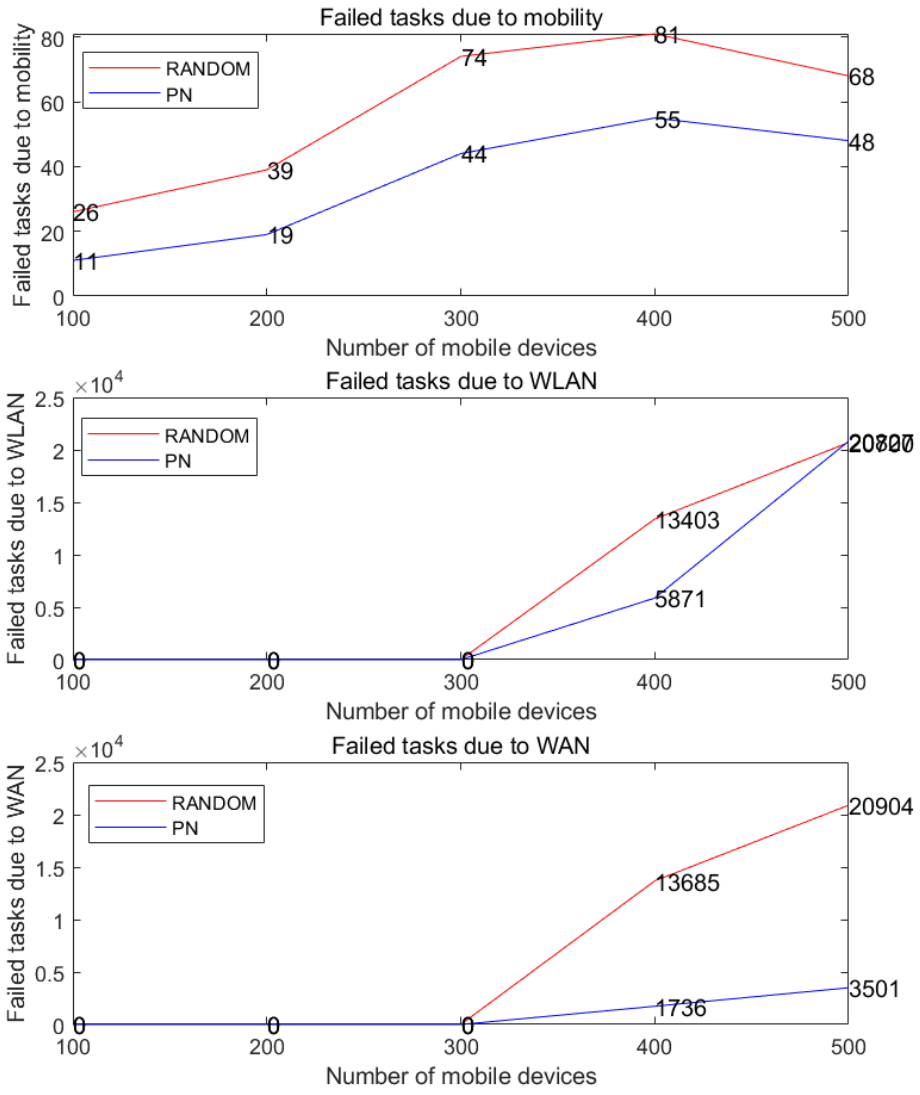


Figure 2.13: Failed tasks- network

that it ignores the fact that different optimal partitioning solutions can be found for different inputs. In our solution, instead we focus on keeping track of the current network and resource status; we tackle a real time scenario, where multiple requests for variation of the same service may arrive at the same time.

Since mobile devices usually use a wireless channel, with the bandwidth frequently changing, static application partitioning methods (those computing communication costs offline and ignoring bandwidth fluctuations) are unsuitable [104].

Authors in [105] suggest to improve predictors for the online partitioning of the application, by adding awareness of the complexity associated with the inputs.

Since online partitioning introduces too high latency, the most promising for AR application is the hybrid approach proposed by [67]. The main objective is to minimize the bandwidth between software components, but the authors promise their algorithm can optimize many alternative objectives as well. In our work we follow the author's suggestions and opt for KL algorithm to implement partitioning.

Based on these works, we can abstract a common architecture [1] consisting of the following components:

The *partitioner*: also referred as a static profiler, whenever a request to offload an application is made, a partitioner decides which components are offloading candidates. This decision is made based on a static analysis of the code. The *context monitor*[2]: provides information about the context of the migration: available edges, device battery status, network connectivity, etc. The *solver*: uses the data of the context monitor and the partitioner candidates to choose whether and on which Edge to execute the offloading candidates. Finally, the *coordinator* manages additional tasks that are necessary to hand over the computation, as in the case of verification and synchronization.

2.4.2 Application Placement

Appropriate resource allocation is an old issue in different disciplines. In this section, we present two resource allocation problems in computer networks: placement of Virtual Machines (VM) in cloud computing and placement of Virtual Networks Function (VNF).

VM Placement. With the term VM placement, we refer to selecting the most appropriate physical machines for VMs. According to [106], objectives of VM placement are maximizing resource utilization, reliability, and availability.

There are several approaches to VM placement in the literature [107, 108, 109]; some variants even consider dynamic placement and multi-clouds placement. For

[1] Similar to [61]

[2] Also known as dynamic profilers/samplers

instance, [107] uses traffic-aware VM placement to improve the network scalability in the data center, defining it as a hard optimization problem solved by a two-tier approximation algorithm to overcome substantial sizes. Authors in [110] present an approach for reorganising two-tiered web applications by allocating resources across data centres in response to workload changes; Each data item that is directly accessed by a client is mapped to the weighted average of the geographic coordinates for the client IPs that access it. The weights are given by the amount of communication between the client nodes and the data item whose initial location is being calculated.

In [111] a cloud management middleware to adjusts the placement of web application components across multiple cloud data centres. Based on observations and predictions of client request rates, it migrates application components between data centres. Their optimization function for the placement is based on RTT reduction. Authors maximize the sum of total request satisfied, attributing them different weights based on SLA priorities. The optimization does not consider costs of migration. Flash crowds or other unexpected workload peaks are not considered in the model.

Also [112] considers resource allocation algorithms for distributed cloud systems. To minimize the maximum distance, or latency, between the selected data centers authors describe an algorithm for minimum diameter subgraph problem that gives the best approximation guarantees. In a similar fashion they select, within each data center, the racks and servers to allocate the requested VMs. The algorithm exploits the tree like structure of a datacenter network. Partition of the requested resources amongst the chosen data centers and racks is decided via heuristics.

With a similar objective to our context aware framework, [113] utilizes CloudStack, an open source cloud management system. A cloud dashboard queries an ALTO server (network protocol by IETF) for the network metrics to feed a placement decision algorithm, which computes a score for each data center and recommends the data center with the best overall score. The ALTO server collects network topology data and provides an abstract view of that topology and costs including dynamic loss and delay information.

Service Chain Placements in NFV. Service Function Chaining (SFC) [114] aims to overcome the limitation of static deployment models applying algorithms that can optimally map SFC to the substrate network. This category of algorithms is referred to as "Virtual Network Functions Placement (VNFP)" algorithms [115]. As explained in [116], in this category of placement problems, we are given a physical network, VNF specifications, and a set of service requests. The algorithm performs the three following steps:

1. Calculate an optimal number of needed VNF types, all the VNFs that should

be instantiated compose a set.

2. Place VNFs to physical nodes such that the demand of VNFs do not exceed the capacity of physical nodes;

3. Assign service requests to VNFs such that the demand per service requests do not exceed the capacity of VNFs.

However, the three steps are not independent, and their order depends on the implementation of the algorithm and the problem statement. For example, [117, 118] give an Integer Linear Programming (ILP) formulation. Many others prefer fast heuristics to allow real-time decisions [119], and propose dynamic programming; [120] provides a Mixed ILP formulation and a heuristic algorithm that solve the problem incrementally, which can solve the problem for incoming flows without impacting existing flows. Among the meta-heuristic solutions, [121] introduces a method based on genetic algorithms, while [122] considers a greedy algorithm and a tabular search-based algorithm.

Although we will not work with VNF specific algorithms, we claim that our methods may be applied to them. It is especially true for network functions such as User Plane Function and special observability, monitoring, tracing, logging, and analytics VNFs.

2.5 Conclusions and Future Works

Migrating from cloud infrastructure to an edge mesh involves a considerable amount of work to refactor applications and service chains. Usually, developers do this refactoring solely based on experience. Simulating tools can be of support, considering that the infrastructure is new and blind deployment may be expensive. Most of the involved resources being pay-per-use, it is crucial to avoid faulty deployments.

In this chapter, we firstly argue the importance of an Edge Computing and 5G infrastructures for the deployment of efficient Continuous Interactive Applications.

Our analysis of application deployment consisted of two phases. In the first work we identify the requirements for an AR use case, select a partitioning granularity and other possible ones to evaluate. We then propose an architecture inspired by previous works, with a focus on a hybrid adaptive solution. We selected the toolchain for context and application analysis to integrate into the framework. Finally, we demonstrated our first experiments on the offloading using a simple face recognition use case.

In the second work, we described the methods and the algorithms we used to develop the prototype of our tool to partition and deploy an application in a

5G distributed network. This is the minimum analysis to be performed to enable a dynamic reallocation of the applications based on the variation of the context conditions. (This context-aware scheduling is later tackled in Chapter 4 of this dissertation) The tool executes three main steps. First selecting the application granularity and construct a graph model. Then reduce it into Modules by solving the NP-hard graph partitioning problem it represents; finally, implement and apply a fractional relaxation of the Path Computation and Function Placement Problem as described by [53]. Simulations were run with various requests of service simultaneously. For our specific setup and our AR application, we can implement a distributed scenario with reasonably low overhead.

The next step would be to implement the new application partition suggested by the framework and locate them in the physical network to verify how close our simulations are to reality. By running the new deployment we could perfect the parameters we used to describe the network capacities and the benefits of the distributed execution. Interesting measures to validate the outcome on different AR applications could be quality and efficiency-related ones: for example, Video Quality as Average Bit rate expressed in Kbps.

2.6 Contributions

In this Chapter we collected our work concerning Thesis 1. I presented an extensive survey of the literature. I established requirements for Edge native applications, concentrating on AR use cases. I created a framework for application offloading in Edge computing and selected a profiling toolset to use. I verified this framework via profiling of three different partitioning scenarios for a video streaming application. I created a model for application partitioning and deployment on a 5G based Edge network. My model exploits a combination of online and offline profiling tools to map an application and decide it's ideal partitioning based on a min edge cut of the function call graph. I created a tool to estimate the best allocation of those partitioned service replica based on network capacity, benefit/SLA function and user requests and service chains. I tested the tool on an Edge Cloud simulator and compared it to a random allocation strategy.

Figure 2.14 summarize the main concepts and contributions.

Thesis 1 (Facilitate Deployment at the Edge)**.** *Adapting code from Cloud to Edge involves a great amount of work to refactor applications and services. Usually developers do this solely based on experience, simulating tools can be of support since the infrastructure is new and blind deployment may be expensive. Furthermore the extensive mobility of the applications represents a challenge for the choice of a correct separation of applications in micro-services.* ***I created a framework for***

partitioning and estimation of best deployment of a monolithic application. I implemented it in the form of a tool to assist into selecting the proper way to refactor an application, via graph partitioning schemes. I demonstrated this concept partitioning an AR application at a function level granularity. *Via non-intrusive online profiling of the application I created a call graph representing functions runtimes and number of calls between functions. The partition process applies min edge cut using a refinement algorithm, MLKL, that Coarsen, Partition and Uncorsen a graph multiple times. Any weighted graph may be used to represent the application in this step in our tool.*

This method aims at suggesting partitions that minimize their interaction, the idea is that the partitions should be easily moved around the network without compromising performances. Indeed more complex approaches may involve other parameters, such as the amount of allocated memory or the dependency to other code/services.

Finally the tool can suggest pareto-optimal places for deployment, based on Network resources capacities and existing links. I simulate the placement and routing issues into a single algorithm taking care to handle multiple user request at the same time. I show how it is possible to compute a deployment that satisfies SLA keeping in consideration cost of resources and benefit of requests. I proved that the approach does performs better than a random deployment, reducing deployment and networks failures.

thesis	relevant publications							
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
(1) **Facilitate Deployment at the Edge**	•	•	•					
(2) Migration in Edge Computing			◦	◦	◦			
(3) Distributed Mobile Edge Scheduling			◦			◦	◦	◦

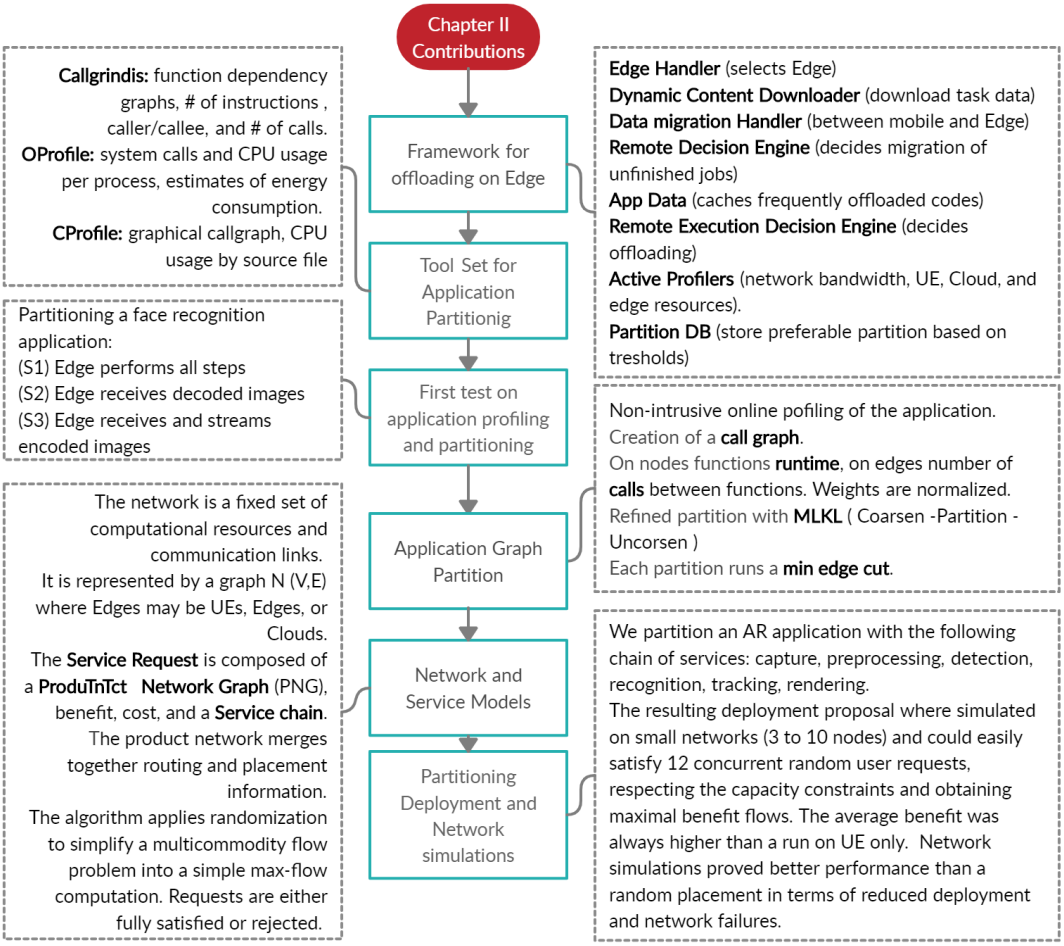


Figure 2.14: Summary of Chapter Contributions

Chapter 3

Migration in Edge Computing

The current trend for Cloud and Edge computing is toward self-contained stateless services, where a whole application can be divided into simple parts with minimal interaction among them and virtually no need for persistent data. Orchestration frameworks and technologies favour this trend, like in the case of Kubernetes [123] where, in the default configuration, for Pods restart or redeployment all session data and temporary data would be lost.

To develop a containerized application the best practice would be using stateless containers. Data generated in one request to the application in the container will not be recorded for use in other requests. The application will be easier to restart and more lightweight. However, real-world applications do require stateful behaviour: data generated in one request should be recorded somewhere in the container to be available for use later on.

It is not always a trivial task to decouple the application components into containers trying to make most containers stateless; there are scenarios in which statefulness cannot be bypassed.

In applications that also require low latency, the challenge is to preserve the state of the container hosting it, while following the user physically moving away from the hosting part of the network.

In 5G fog networks, this is the case for some virtual network function and a number of consumer application, such as real-time multi-player gaming, remote health inspections (e.g. ultrasounds) or autonomous vehicles.

A concrete example for this use case is that of a smart vehicle providing on board infotainment, while crossing countries borders: HD maps or video games would be hosted on an Edge in country A and should be handed over quickly to country B while the car moves towards it at high speed.

Migration of an application from an Edge to the Cloud may also be needed for recovery reasons.

However there is no well-established method to implement live migration of

containerized applications in Edge environments.

In this Chapter, we propose a mid-range design to integrate live migration in an ad hoc peer to peer Edge network exploiting DockerSwarm and in a setup made with state of the art orchestrator (Kubernetes).

The Chapter is structured as follows. After a deeper introduction to the objective of Live Migration, we compare container and VMs solutions. In Section 3.1.3 we present a classification of migration models, libraries and existing solutions. In Section 3.1.4 we compare existing container orchestrators and justify our choices. Section 3.2 we present our first setup of a Fog peer to peer network implementing LM. Section 3.3 presents our effort into integrating LM in Kubernetes, preserving the controls of the platform. In Section 3.4 we compare our work both with offloading and placement frameworks. Section 3.5 summarizes our findings while 3.6 .

3.1 Background

The objective of this chapter is to specify basic concepts behind Fog, application encapsulation and migration.

3.1.1 Fog computing in 5G

One of the key enablers for 5G networks is Edge computing: the concept of exploiting smaller resources collocated at the edge of the network, for example at the antenna site, to reduce delays. The delay reduction is twofold. On one side we reduce the number of network and routing elements that our data has to travel trough, thus cutting processing and transmissions delays; on the other, we also diminish the physical distance impacting propagation delay. In most cases, latency is a byproduct of distance. Although fast connections may make networks seem to work instantaneously, data is still constrained by the laws of physics and can't move faster than the speed of light.

A complex case in Edge computing solution is represented by Fog computing, that involves very diverse devices, also User Equipment, to collaborate at the edge of the network to allow computations offloading and task sharing. As stated by [23] Fog networks are characterized by high node and user mobility, high diversity of the nodes and incomplete mapping of the Fog network itself. We can consider it as a Device to Device ad-hoc network, coexisting with the more structured mobile network, trough edge nodes.

Authors in [124] also use the term Fog to indicate any three-layer IoT-cloud architecture with an intermediate layer of geographically distributed gateway nodes,

typically well positioned at the edges of network localities, that are densely populated by IoT sensors and actuators.

Overall we can state that a Fog network is a distributed system close to a large-scale sensor network, with prevalent wireless access and high need for interoperability. Physically close nodes should be able to collaborate among themselves but also to interact with more reliable and far resources.

As specified by the National Institute of Standards and Technology of the US chamber of commerce in [125], fog nodes are context aware and support a common data management and communication system. They can be organized in clusters, both vertically (to support isolation) and horizontally (to support federation).

The main challenge in orchestrating this type of networks will be finding the beneficial trade-off between uncontrolled and centralized orchestration. In our proposal we do not concentrate specifically on the federation part of the problem, but it is worth to mention that there are multiple strategies to implement cluster federation, including Kubernetes itself.

Among customized orchestration for Fog, [124] work is also worth mentioning. The authors propose IoT gateways working as full fog nodes, by extending the existing base of open-source Kura IoT gateway. They employ containerization techniques, in particular, Docker, Docker Swarm and live migration to deploy measuring/sensors data collection services on Raspberry-Pis. Authors mention a Swarm based orchestration solution, which make us suppose a solution based on API calls to Docker Swarm functionalities from the Kura gateway. In our work, we specify a more generic solution that is immediately deployable on any existing cluster.

3.1.2 Comparison of Virtual Machine and Container Migration

While VMs are often chosen to abstract, split, duplicate, and emulate entire servers, OSs, databases, and networks, containers are used to package single functions that perform specific tasks.

In Fog computing, the situation is the same. For example, the ETSI standard for Virtual Network Infrastructure [126] relies on Virtual Network Functions being Container Infrastructure Service Instances. The Container Infrastructure Service can then be deployed on the hardware resource (bare metal) or deployed on a VM, to ensure isolation.

The first reason why we preferred containers in this scenarios is one of flexibility and scalability: containers smaller size favours migration, since any incremental file synchronization or migration mechanism will need to remotely compare a larger amount of data in case of VMs.

In [127] Linux Containers (LXC) are proven to have a clear advantage over Linux Kernel-based Virtual Machine (KVM) in terms of total migration time, service downtime, and amount of transferred data. This is mainly because containers are more compact and the filesystem and in-memory contents include mostly what is relevant for the application, whereas VMs content can be related to many other background processes.

The second factor influencing the choice was the "cloud-native" aspect. The term describes applications designed specifically to run on a cloud-based infrastructure. Edge computing must follow this principle since, apart from the locality and size, it will act like a cloud infrastructure.

Cloud-native applications are usually loosely coupled micro-services running in containers managed by an orchestration platform. To anticipate failure, run and scale reliably, applications must adhere to a set of constraint that facilitate automation (Single-concern, self-containment and runtime-confinement to mention a few [128]).

Because containers best practice requires small images, arbitrary user IDs, marking ports and using volumes for persistent data, most VM native applications cannot be mapped one to one with a container.

Decomposition from VM to Containers becomes an essential part of the migration process. It helps to split large monolithic application topology into small, logical pieces and work with them independently.

Taking into account these differences and the fact that Fog and Edge nodes are diverse, often with limited bandwidth, unstable network connectivity, costly or limited storage and processing capability, running a container-based solution will be much more beneficial.

As mentioned in [129], with containers, the complexity can be reduced through container abstraction since they avoid reliance on low-level infrastructure services. Automation can be supported to maximize portability. Security and governance can be achieved by placing services outside the containers. Higher computing capability can be provisioned with service composition, achievable even if the containers run on different physical machines.

Given the above mentioned advantages, more and more mainstream operating systems begin to adopt container technology to provide isolation and resource control, which has demonstrated great potential for service migration [130].

3.1.3 Container Live Migration

Live migration attempts to provide a seamless transfer of services between physical machines without impacting client processes or applications.

It can be used for those services that require too long start-up time and that must ensure low downtime.

Strategies and techniques to achieve migration of containers build on top of those of VMs.

In fact, migrating containers involves migrating running processes and warrant that all accessed files will be available at the destination host.

In the literature there has been successfull attempts at migrating containers [131] [132]. In this work we will exploit the CRIU [133] project.

The Checkpoint Restore In Userspace is a project to implement checkpoint/restore functionality for Linux. It can be used to perform Lazy migration (postcopy) and precopy migration, live and iterative strategies:

- In lazy, or postcopy memory migration, when the task accesses missing memory pages, CRIU processes the page fault, transfers the required page from the source node and injects it into the running task address space. The dump action should be invoked with *–lazy-pages* option. In addition, *–address* and *–port* options may be used to select IP address and port that will be serving the page requests.
- precopy migration can be performed by running pre-dump commands specifying the images directory where to save them.
- In live and iterative migration CRIU utility can be used both with a shared file-system such as NFS or by using rsync. Also, a Page server to reduce read and writes operations is available.

3.1.4 A Comparison of Opensource Container Orchestration

Container orchestration frameworks provide support for management of complex distributed applications. Different frameworks have emerged only recently. The three most prominent orchestration frameworks are [134]: Docker Swarm, Kubernetes (K8s), and Mesos (DC/OS).

Kubernetes is an open-source technology for automating deployment, operations, and scaling of containerized applications. It groups the containers making up an application into logical units for easy management and discovery, for example, based on their resource requirements and other constraints. Kubernetes also provides horizontal scaling of applications, which can be performed manually or automatically based on CPU load. Finally, it provides automated rollouts and rollbacks and self-healing features. An overall architecture of K8S is described in Figure 3.1.

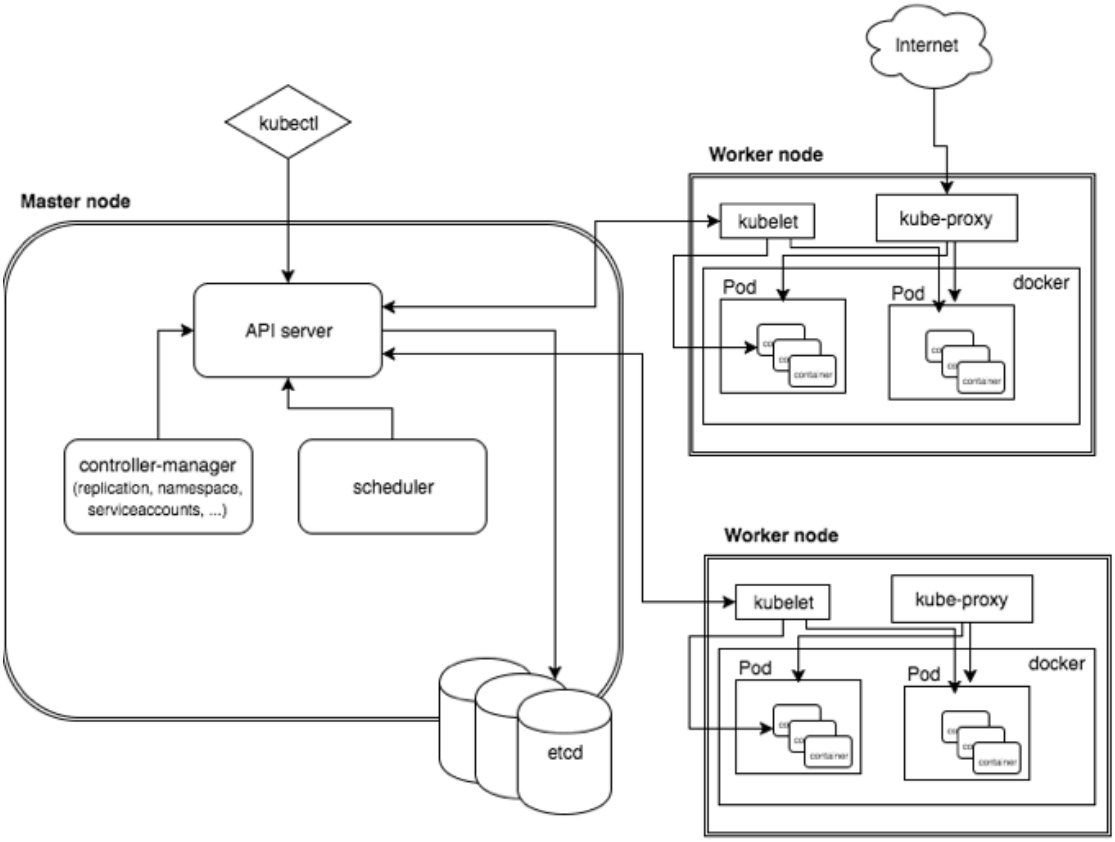


Figure 3.1: Kubernetes Architecture

Kubernetes deploys and schedules containers in groups called Pods. Pods are scheduling units (and can contain one or more containers). Containers in a Pod run on the same node and share resources such as filesystems, kernel namespaces, and an IP address. Deployments can be used to create and manage a group of Pods. A service tier can be added for scaling horizontally or ensuring availability. Services are endpoints that can be addressed by name and can be connected to Pods using label selectors. The service will automatically round-robin requests between Pods. This allows two Pods to communicate over the Kubernetes overlay network. Kubernetes will set up a DNS server for the cluster that watches for new services and allows them to be addressed by name. Each host can be a node part of a cluster. Each cluster will have a master node that places container workloads in user Pods on worker nodes or on itself. The same machine could also host multiple virtual-nodes using VMs or Containers. There are then, a number of components associated with a Kubernetes cluster [123]. An *API Server* will be the management hub the Kubernetes master node. Its role will be to facilitate

communication between the various components, maintaining cluster health. The *etcd* stores all configuration data, including those that influence the API server. *Controller Manager* will ensure to scale workloads up and down according to placement decision performed by a *Scheduler*. Finally, the *Kubelet* component will receive Pod specifications, for example in a yaml file format, and manage Pods running in a host.

Docker Swarm provides native clustering for Docker containers. It turns a pool of Docker hosts into a single virtual Docker host. Because Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts.

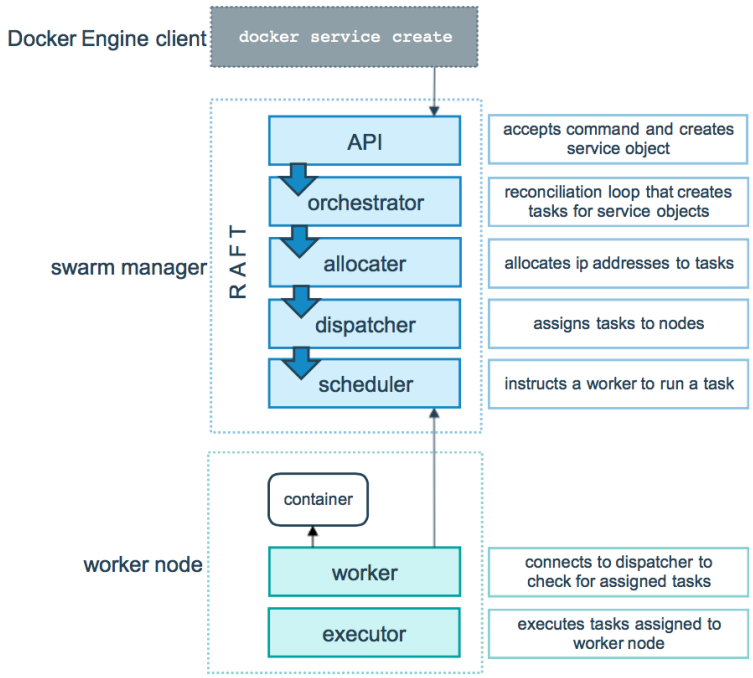


Figure 3.2: Docker Swarm architecture

Docker Swarm architecture consists of managers and workers. The user can specify the desired state of various services to run in the Swarm cluster using YAML files. A node is an instance of a Swarm. Nodes can be distributed, they can be virtual or physical, and located on proprietary machines or in public clouds. A cluster of nodes (or Docker Engines) is defined as a swarm. In Swarm mode, one orchestrate services, instead of running container commands. Manager Nodes will receive service definitions from the user, and dispatch work to worker nodes. Worker Nodes will collect and run tasks from the managers. A Service in Swarm

is nothing but a configuration specifying the container image and the number of replicas in the swarm.

At service deployment, the swarm manager accepts the service definition as the desired state for the service. The underlying logic of Docker swarm mode is a general purpose scheduler and orchestrator. The service and task abstractions themselves are unaware of the containers they implement.

Figure 3.2 shows how swarm mode accepts service create requests and schedules tasks to worker nodes.

DC/OS (the Distributed Cloud Operating System) is an extended Mesos distribution that allows fine-grained sharing of cluster resources across multiple scheduler frameworks. Its decentralized scheduling architecture can support large-scale clusters till 50,000 nodes. It also offers support for load balancing non-container orchestrated workloads such as databases or high-performance computing applications. We believe that for Fog applications such scale will not be needed. However, in the next section, we proceed with a description and comparison of all three technologies.

DC/OS is itself a distributed system, a cluster manager, a container platform, and an operating system. It includes a group of agent nodes that are coordinated by master nodes. Components running on the master nodes perform leader election with their peers. Cluster management is provided by Mesos.

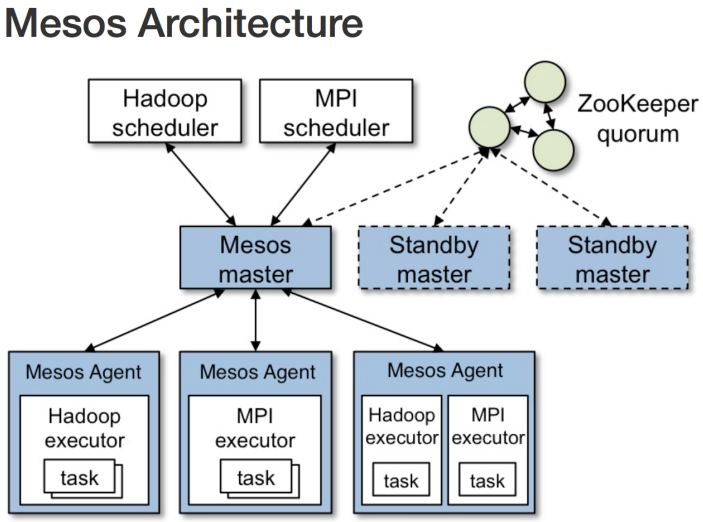


Figure 3.3: Marathon architecture overview

All tasks on DC/OS are containerized. Container orchestration is one example of a workload that can run on Mesos architecture, and it is done using a specialized

orchestration "framework" built on top of Mesos called Marathon (See Figure 3.3).

Containers can be started from images downloaded from a container repository or they can be native executables containerized at runtime. The platform supports two container runtimes: Docker and Mesos. Docker is currently required on every node but it may become optional in the future, as components and packages migrate to using Mesos Universal Container Runtime.

The logic of the management is the following [135]: a Mesos Master aggregates resource offers from all agent nodes and provides them to registered frameworks (containerized applications running on Mesos togheter with its executors and a corresponding scheduler). Examples of Mesos frameworks include Marathon, Chronos and Hadoop Offer: a list of an agent's available CPU and memory resources. An Agent daemon can run on the same component than the master daemon or on a different one. Agent nodes provide resources exposing them as resource offers and making them available to registered schedulers. The schedulers then accept these offers and allocate their resources to specific tasks, indirectly placing tasks on specific agent nodes. The agent spawn executors to manage each task type and the executors run and manage the tasks assigned to them.

3.2 A framework for application migration in Fog Computing

The proposed system is a peer-to-peer collaborative computation network. The nodes in the network are light-weight uniform piece of software, whose main responsibilities are: to maintain the graph of the available nodes, to move the code of the pieces of the deployed application and to organize communication among the deployed modules (as represented in Figure 3.4).

Applications that are intended to be run over the network, must be partitioned to minimize the imposed overhead. As we exposed in Section 2 we can separate monolithic applications along the minimum cuts of their function call graph (or data flow graph or in some other balanced middle-way), to minimize data flow in between parts [2].

In this work we regard applications as a composition of micro-services, assuming that with more or fewer efforts any application can be transformed into such an architecture.

In such a framework we had to address in the design process: (A) the mapping of the available resources, (B) the creation of the deployment plan, (C) the plans and approaches to move codes to their assigned location, and (D) the strategy to maintain the health of the service and the shared state of the application, that is how to transfer data between the functions, and how to keep globals up-to-date.

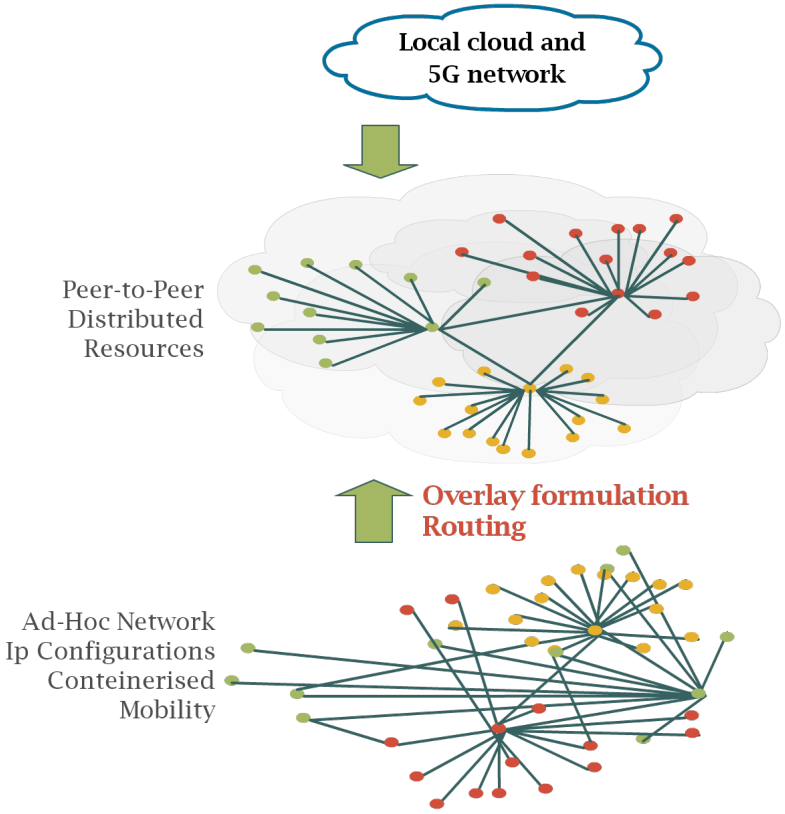


Figure 3.4: Our fog network implementation is perceived from the rest of the mobile network as multiple overlays deploying different services

3.2.1 Network mapping

When in need to deploy or execute a service, a network user can make a decision on the deployment according to the mapping strategies best fitting the characteristics of the service. To make it possible, the system should provide the initiator node information about the possible Service Level Agreements (SLAs): the description of the available resources along with the price of the usage of a particular resource, also expressed in terms of latency or volatility of the perceived QoS (Quality of Service). We will further discuss possible parameters to be considered in the next paragraph.

3.2.2 Task assignment

With the mapping of the available network and the call-graph/interaction graph of the service, the next task is to map the modules to nodes. There are several methods to do task assignments, like the placement of Virtual Machines (VM) in cloud computing or placement of Virtual Networks Function (VNF). The paper [136] shows a global classification of VM placement solutions into online and offline approaches, categorizing them based on their target objective. Some of the solutions are single objectives, others may have several. The point of view of the analysis may be either the final user or the network providers. Among the objectives we can mention:

- Energy consumption: based on minimizing power consumption and number of active nodes;
- Cost: as Return On Investment (ROI), resource exploitation or allocation cost;
- QoS: can be expressed in terms of response time, overhead time;
- Resource usage: RAM, CPU, storage;
- Reliability;
- Load balancing: avoidance of congestion, data overload.

3.2.3 Deployment infrastructure: an Orchestrators Comparison

Given our premises on Fog, an important choice to improving services deployment is the selection of orchestration environment. One can use either Kubernetes, Docker Swarm or DC/OS for Docker engines.

[134] identifies Mesos as the most fitting platform for prototyping novel techniques for container networking and persistent volumes because of the adherence to all relevant standardization. This classification made a stride away from Mesos.

Docker and Kubernetes are instead suggested for prototyping for container runtimes.

One weakness of Kubernetes, compared to Mesos, is that automated installation of a cluster with multiple master nodes is not supported (in the open source version). It is also less scalable: no more than 5000 nodes in a single cluster. In our scenario, however, we considered the number of nodes sufficient.

Kubernetes is the most generic orchestration framework. However, the differences with Docker EE (docker Enterprise solution) in supported features is small.

Kubernetes is also identified as the most mature container orchestration framework. It has unique support for integrating with public cloud platform's load-balancing tiers and offers a wide range of external service discovery. In fact, a large number of public cloud providers have offer Kubernetes as a Service.

Kubernetes usually is not a complete solution and requires custom plug-ins to set up; on the other hand, Docker Swarm has the advantage of tightly integrated into the Docker ecosystem, and uses its own API. This removes many compatibility issues and other differences and favours smooth integration.

Unfortunately, the community and support around Docker are not as expansive and convenient to address. Most cloud providers today offer Kubernetes as a service. Kubernetes currently holds the largest market share and is almost the standard platform [137].

Because of these reasons, in our industrial work we selected Kubernetes, as the best orchestrator for a Fog network infrastructure that integrates within the existing mobile ones. Instead in our first work on a Container Orchestration framework for an ad-hoc Fog network we chose the API of Docker Swarm that is the least restrictive for common actions such as creating, updating and stopping a container.

Service Migration Infrastructures At deployment, thus, the challenge is how to move the code and resources between the nodes. As a base scenario, we assume that the modules of the service to be deployed resides at the initiator. The initiator can contact the joined nodes assigned to the tasks and push the codes and data that he needs to run. Multiple applications, each made of several components should simultaneously use the Fog infrastructure. Memory isolation is necessary for security and integrity reasons but also for bug prevention and performance tuning. The real-time constraint is an added requirement to the migration.

This can be achieved by using either a shared file-system such as NFS [138] or by using rsync, a unix-like OS utility, able to copy files across networked hosts. The latter requires to know the destination host address or a predefined path. The

former solutions, based on undefined path, require more downtime, but overall have shorter migration time. This is because only memory has to be transferred since they involve a shared file system.

Migration using predefined path gives the smallest downtime but increases the total migration time. In the case of predefined paths one can have either disk-based or diskless solutions. In the first case, a temporary file system (tmpfs) can be used to copy files from the original host to target mounted volume. This approach requires two reads and two writes operations: it moves data on the original host to a tmpfs, copies them in the target host tmpfs and finally loads them in target memory.

Instead, a diskless approach involves having a fix known address and port to directly dump the memory on the target host. This reduces the reads and write operations and can be performed by a separate actor (eg. a page server).

To shorten the migration time precopy and postcopy strategies can be applied as well.

As the term precopy suggest, one can copy all tasks memory before moving a container image and permanently stopping it. Performing multiple precopy steps would generate other sets of images which will contain memory changed after the previous step. Doing several precopy iterations may reduce the amount of data dumped on dump stage (the final memory copy before stopping the container) and thus lead to shorter freeze time.

However, this implies starting the migration quite ahead of time, since the size of the first transfer may be considerable, and thus create the need for predictive or preventive mechanisms.

In case of lazy, or postcopy memory migration, one can minimize application downtime. Unlike precopy memory migration, lazy migration does not copy the task's memory, but rather keeps the memory pages at the source node. Only the minimal task state required to start the application is copied to the destination node and resumed. When the task accesses missing memory pages, those pages will be retrieved and injected in the running task address space.

Iterative migration mostly uses pre-copies. Even though it can also implement hybrid approaches with post-copy, we followed the classical sequence:

1. Pre-dump: we take memory page to migrate and pre-dump them into some place (this can be either tmpfs or page server or final host). Tasks will remain running after pre-dump, unlike regular dump. Multiple iterations of these steps can be done.

2. Dump: this dump only takes the memory that has changed since the last pre-dump;

3. Copy: Copy images to the destination node;

4. Restore: On the destination node restore the apps from images;

5. Kill: right after the last dump is performed, the stopped tasks on the source node gets killed.

After selecting an iterative approach for Live Migration we proceeded with a comparative study of existing solutions.

Analysis and comparison of LM Existing Solutions Migration of containers becomes possible since CRIU checkpoint restore functionality for Linux. CRIU checkpoint and restore of containers is integrated in Linux Containers [139](LXC), runC [140] compatible containers (eg. Docker [141]), Virtuozzo (based on OpenVZ [142]) and Jlastic. While the latter two present out of shelf solutions for live migration, the others only allow calls to the CRIU library:

- runC, from the Open Containers Initiative (OCI), is a lightweight universal container engine; it is a command-line tool for spawning and running containers according to the OCI specification. RunC is available in Docker Experimental mode together with CRIU library and allows to implement live migration.

- Linux Container (LXC) is a lightweight virtualization technology integrated into Linux kernel enabling multiple containers on top of the same host. LXC is a userspace interface for the Linux kernel containment features. Through a powerful API and simple tools, it lets Linux users easily create and manage system or application containers [143].

- Docker [141] extends LXC with a kernel- and application-level API that together runs processes in isolation: CPU, memory, I/O, network, and so on. Docker also uses namespaces to completely isolate the application view of the underlying operating environment, including process trees, network, user IDs, and file systems [144].

- Virtuozzo [145] team created the live migration technology for containers (OpenVZ); today they offer a production-ready containers engine with live migration. It is claimed that migration could be done within 5 seconds.

- Jelastic [146] is a proprietary solution, offering a container orchestration platform that provides live migration of production applications across hardware regions, data centres and cloud vendors. In a recent demo, Jlastic shows Live Migration of a Minecraft container from AWS to Azure without downtime.

Live migration of LXC containers is done using LXD (Linux system container management project) and CRIU [147]. The state of the container is saved, copied to the destination node through remote images and then the container is restored to the target. Successful completion of migration kills the container data on the source server. Live migration of OpenVZ containers involves private keys to check authorizations. Once the connection is established, the container can be migrated by first synchronizing the file system, then updating the memory pages in the container. Finally, the last dump file is transferred to the destination. The container is resumed while its data in the source host are removed.

LXC can be considered an OpenVZ redesigned to be able to be merged into the mainline kernel. However, OpenVZ uses a distributed storage system, where all files are shared across a high bandwidth network. This means that during container migration no file transfer needs to be performed. This is the reason why checkpoint/restart migration on LXC has higher CPU utilization, thus lower performance when compared with OpenVZ. However, as shown in [147] if the CPU load is lower, LXC proves less migration time and downtime.

Anyhow, due to the limited bandwidth for edge servers and the volatile wireless connection of Fog nodes, it can be challenging to deploy distributed storage [148]. Therefore, OpenVZ containers are not suitable for our scenario.

Authors in [149] advocate that LXC gives more flexibility for running different applications, services and protocols. Dockers are standalone applications running in an isolated environment and do not offer any system level functionality as LXC does. Moreover, the checkpoint and restore phases take a considerable amount of time for a lightweight Docker container.

In our work, we decided to opt for Docker to reduce complexity and friction of adoption for the final user. Docker containers are easily scalable and allow component reuse (Plug-in like features) and collaboration. Finally the support available for Docker is higher.

It is true that Linux containers, being integrated into the kernel OS systems utilities are more lightweight and give direct access to the kernel. However, it is in question whether such access should be allowed in a Fog infrastructure. LXC upstream's position is that Privileged LXC containers are not and cannot be root-safe since there are a number of exploits which will let a container get full root privileges on the host [150]. Some risk is not avoidable as they would require blocking so many core features that the average container would become completely unusable. Unprivileged containers could also present security issues: actions a non-root access can perform and generic kernel security bugs.

In the first iteration, Docker containers were essentially LXC containers, and they came with the same security features. Docker initially relied on LXC as its container interface, but because LXC provides each container with a full Linux

system in an isolated namespace, Docker developed its own runtime, containerd, as a replacement. Containerd is used for single application containers, while LXC is able to run multiple applications inside system containers.

Contrary to LXC, Docker will start containers with a very restricted set of capabilities [151]. In most cases, containers will not need "real" root privileges at all. This means that "root" within a container has fewer privileges than the real "root"; making it harder to do serious damage or to escalate to the host.

Based on our analysis we define a migration scenario as following (Figure 3.5):

Any UE or Edge node may trigger a migration event. The node currently serving the UE will react by selecting the next best node to migrate the service. Service migration is done in an iterative step during which the UE can still access the service on the first node. At the final step of the migration the service is down just the amount of time needed for the second node to restart the container. The change of the node hosting the service should be transparent to the UE that always calls for it at the same IP.

3.2.4 First Experimental Settings

In this first experiment we model a Fog computing environment fulfilling the requirements described at Section 3.2, using existing and more or less mature technologies as *Docker*, *Docker Swarm*, *CRIU* and *runc*. The orchestrator enables dynamic adjustments on the underlying infrastructure where the managed service is running on.

As mentioned before, Swarm could potentially use docker experimental feature integrating CRIU basic live migration. However, this has to be completely configured and handled by the user, so we wanted to verify whether such a setup was feasible.

Management Orchestration layer The Management and Orchestration layer is responsible for building up and maintaining an overlay network, that is capable of running the service at predefined QoS (as shown in Fig. 3.4). The orchestrator consists of a set of peer applications that run on each physical machine. When a node joins a network, it contacts a known area-node whose address could be obtained through DNS lookup. The node then shares some of its own connections, to make the network more connected. When a node needs to submit a task it has to map the available network. The notion "availability" includes the feasibility of the QoS requirements of the given task, in the meaning of taking into consideration the bandwidth requirements and acceptable total network latency.

Technically these nodes should be implemented in a cross-platform language. Thus, we have chosen Java for this purpose. The control communication, that is

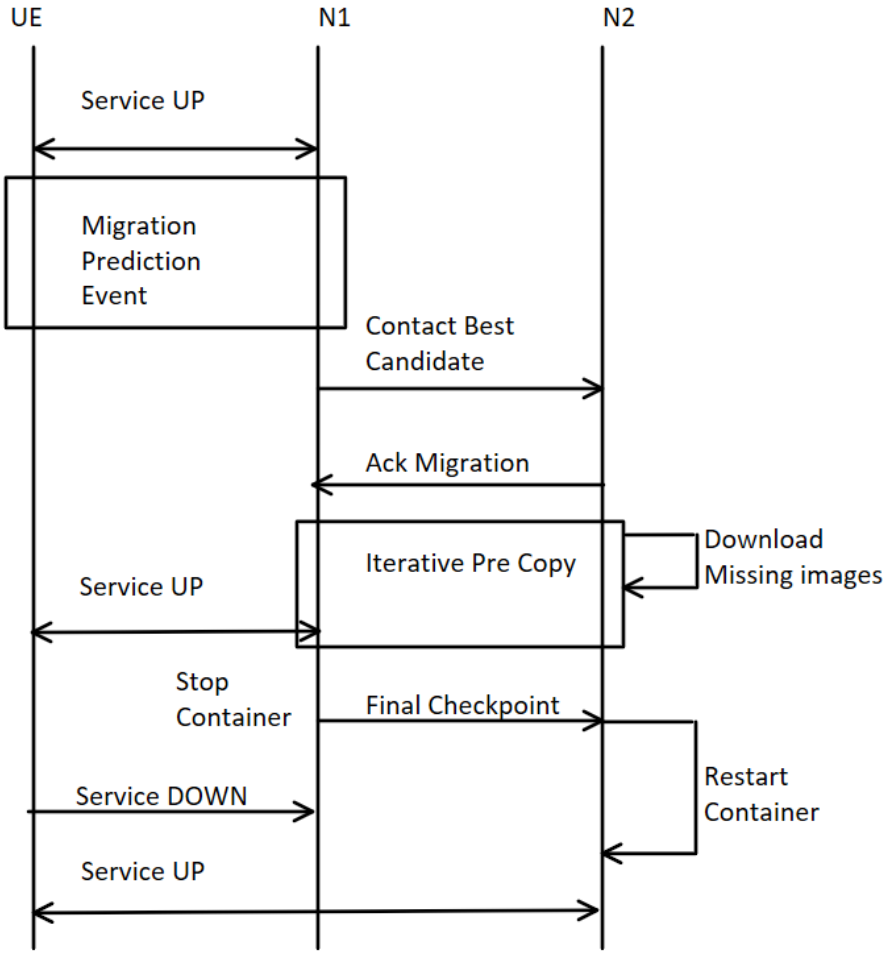


Figure 3.5: Migration for Edge Computing

sending connection or mapping requests, or making agreements on secondary channels uses REST protocols over a standard TCP connection. The nodes maintain the addresses of the nearby nodes along with network segment reachable within a given time constraint.

Network mapping

Since fog computing systems are expected to scale massively, new compute nodes are likely to join frequently. Therefore, node discovery is an essential part of fog computing systems.

In this iteration, nodes are simulated as multiple java programs running on two different linux machines. Before the deployment phase, a peer sends a request to the neighbouring nodes with a unique identifier. As a node receives this request it returns an acknowledgement, through which the actual latency can be learned. A receiver then forwards the request to its neighbours, who are statistically predicted to be within the remaining latency constraint, measures the actual latency and so on. If a peer has already received a mapping request with the same id, it only sends back the acknowledgements since the sub-network that is reachable through it is already submitted in another mapping path. This helps to avoid network overflowing with useless traffic. The results will be sent back recursively to the requiring node, eventually reaching the originator, where a local picture of the neighbouring nodes will be assembled.

The first phase is randomized and about collecting the shape of the network along with some important statistics of the available resources of the nodes and the quality of the links between them. Based on this the deployment plan can incorporate the learn knowledge on volatility and availability of the components.

Task assignment

Task assignment phase covers the creation of a deployment strategy for the tasks in the service, that is a mapping between the physical network and our task call graph, which is the result of partitioning the software to be deployed. The topic of this task assignment is often referred to as Path Computation and Function Placement and is described in [53]. For this scope, we selected an eager randomized approach of randomized rounding introduced in [53].

According to this method, the service chain graph, representing the application tasks and their interaction, is traversed in Breadth-First Search fashion, assigning the root task to the initiator node. At edges in the graph (calls between the modules/tasks), the location of the subsequent task will be chosen pseudo-randomly from the neighbouring nodes of the source task node. This happens so that hosts with higher capabilities and/or better communication links from the node of the

parent task will be chosen with a higher chance. Different policies can be provided. A more detailed description of the process is given in Chapter 2.

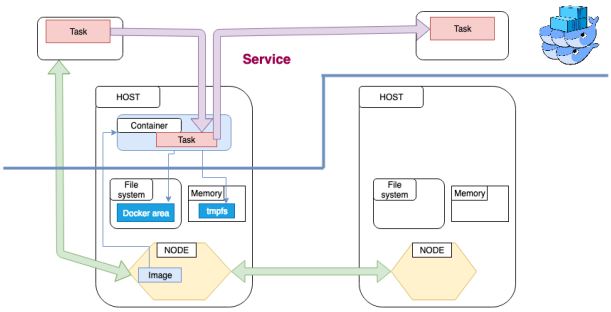


Figure 3.6: Before migration

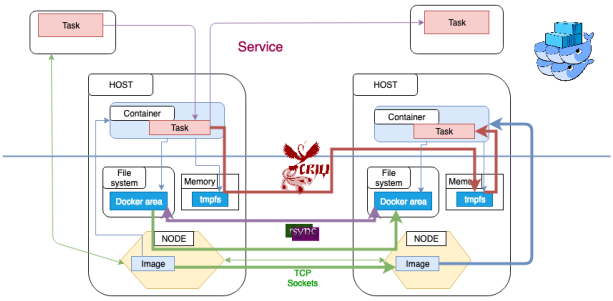


Figure 3.7: Migration

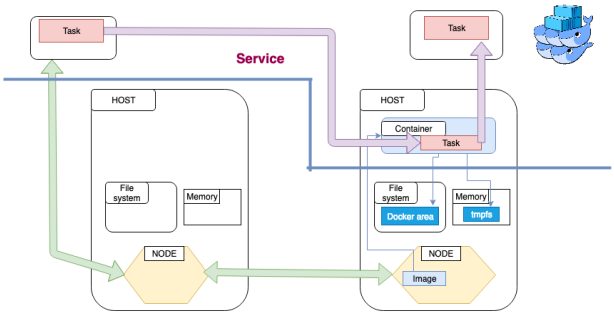


Figure 3.8: After migration

Deployment

We choose Docker containers to encapsulate tasks. To reduce the complexity of routing the communication between tasks located at different hosts, the containers carrying the code of the tasks will be deployed to a Docker swarm, assigning a unique port to each task. At the deployment, the initiator node of the service creates a swarm and sends a request to the nodes that are the assigned location of the individual tasks to join the swarm.

At the first deployment of the service, when no images exist for the tasks, the efficient way to move and execute codes at deployment is to move the all the service-specific volumes and build the images and containers at the destination node.

In this case, the missing images are passed with the data needed at the execution through TCP sockets, to avoid download from online repository (green line in Figure 3.7).

Migration Support The problem of relieving the workload of struggling hosts, or reducing the increased communication latency can be addressed through tasks relocation. In our simulation we select candidates for migration in a naive fashion: for each node a local_cost is calculated as a score summing the percentage of available CPU and MEM resources that the container would occupy if moved on the node; than the migration can be triggered as follows:

Algorithm 3.1: Migration trigger

```
Trigger Migration (node, container):
    neighbours_costs = get_cost(node.neighbours)
    if local_cost <= min(neighbours_costs) then
        keep running container
    else
        select random node where cost < local_cost
        migrate container
```

To implement iterative migration, we rely on *runC* containers, that are natively integrated with *CRIU*. The main steps we will follow to achieve container LM are the following:

a) select a container engine compliant to runC

b) setup a mechanism to share memory or data among containers on different machines

c) install the CRIU library

d) transmit base image and the task-specific static volumes (ex. files, that are mounted into the container) to the new host

e) run checkpoint commands

f) starts the container at the destination host and mounts the volume and tmpfs, where the dumps from the old node will be saved. After the transmission of the final dump, the state of the old task can be restored at the new host.

g) iteratively update the state of the container: one or more pre-dump needs to be taken, which keeps the original process alive, while starts tracking memory changes.

h) utilize a diskless approach with a known address and port to directly dump the memory on the target host (e.g. CRIU page server or a K8s Service endpoint, tmpfs mounts)

i) when migration is done (final dump) kill the other instance

In our simulator Live migration of the container takes the following steps (Figures 3.6, 3.7 and 3.8): (1) through the management nodes we transmit the docker image and the task-specific static volumes [152] (files, that are mounted into the container), that is, the docker image containing the kernel and libraries the task needs. (2) If there are files in the volumes, that are written by the task, they might be synchronized using tools like rsync [153], that provides a very fast method for remote file synchronization. (3) When the above steps are done, it is possible to create the container at the target destination.

What left is to transmit the internal state of the migrated process. For achieving the least process downtime possible, one can execute incremental checkpointing [154] of the process to be relocated. At first, (4) one or more pre-dump needs to be taken, which keeps the original process alive, while starts tracking memory changes. After a number of these pre-dump operations, when everything is ready on the receiver side, the last dump can be taken (5), that kills the process at the original location.

The shipping of the dumps of the old container should take place using the CRIU page server [155], that is a component that has been developed to move user memory to a destination system. This enables disk-less transmission of the state, loading the taken dumps into the so-called *tmpfs* [156] mount at the destination container. The *tmpfs* mounts make possible to write temporary files in the system memory. For that (6) the system starts the container at the destination host and

mounts the volume and the *tmpfs*, where the dumps from the old node will be saved.

After the transmission of the final dump, the state of the old task can be restored at the new one. When the migrated process is up, the neighbouring tasks in the call graph will communicate with the new tasks. Since the described system uses Docker swarm, there is no need to rewire the connections, because it grants us relocation transparency.

Meanwhile, on the original node, it is possible to detach the node from the swarm, since the dump operation stopped the process of it.

The best strategy we have found to approach the idea of *live migration* is that of incremental checkpoint (we use iterative pre-copy for the incremental steps of the migration).

The main reason to introduce migration in a Fog environment is to increase service availability, thus we want a migrations that introduces the least downtime. In iterative migration the overall migration time will be dependent on the Pre-Migration steps (1 and 2), the Container Creation (3), the Iterative Pre-dump (4) the Last Dump (5) and the Re-Start times:

$$T_{tot} \triangleq T_{PM} + T_{CC} + T_{IP} + T_{LD} + T_{RS}, \tag{1}$$

However, the downtime will be a fraction of those, since the original container will stay active while being copied. The downtime will regard only the last copy and the restart phases:

$$T_{DT} \triangleq T_{LD} + T_{RS}, \tag{2}$$

While this first setup proved to us that we can achieve an iterative migration on a simulated Fog network, it also presented various limitations:

Docker Swarm utilizes virtual machines to run on a non-Linux platform. An application which is designed to run in docker container on Windows can't run on Linux. This limits the usage of Swarm for our Fog scenario.

Docker Swarm does not provide an embedded way to connect containers to storage, requiring manual configuration. Furthermore in a Fog network there is no guarantee that the same host will run the same container each time it is run. In this case, every time the application restarts or it is migrated it will require separate scripts handling volumes.

Finally Docker Swarm provides too basic information on container status and does not collects enough data on the deployments.

Mainly for this reasons, we decided to improve our setup by switching orchestration towards Kubernetes, as presented in the next chapter.

3.3 Integrating Live Migration to Kubernetes

Migration strategy in Kubernetes is by default cold: it shut down, move and restart, losing any intermediate state.

Currently, relocating a Pod in Kubernetes is only feasible by removing the source Pod, then recreating a new Pod of the same type.

New pods have no obvious relationship to the original pods they replace. They have different names, different user ids, different IP addresses, different hostnames (since we set the pod hostname to pod name), and newly initialized volumes.

Also, update options are limited. For example, one cannot add or remove containers from a Pod. Instead of explicitly replacing, one can update a Pod using a patch.

Kubectl patch modifies on the fly the Pod configuration. It is important to realize that the container is anyway destroyed and re-created with the new version, so the service still has an outage. Application developers need to design around this fact.

Live migration can be a mean to relocate a container or to change its state, without Pod restart.

To integrate live migration in Kubernetes we started by experimenting CRIU migration features using runC library for Docker (docker_runc). The docker image is flattened to a runC one so that calls to basic library checkpoint and migration configurations can be used.

As a second step, we had to understand how to access a docker library for a Docker container deployed on Kubernetes.

There was a possibility to access remotely to nodes in the cluster and run bash commands on them. We could have installed a docker environment on the node and create containers to migrate at the node site. That strategy would have been against the principles of SELF and RUN TIME CONTAINMENT for cloud-native orchestration. That is why we resolved the problem nesting a Docker Daemon instance inside the container. For running containers under Kubernetes it is necessary to implement a Container Runtime Interface (CRI) integrated with a container runtime environment compliant with the Open Container Initiative (OCI). As mentioned in Section 3.2.3, OCI includes a set of specifications that container runtime engines must implement and a seed container runtime engine called runC. This is ideal in our case since CRIU leverages on runC. In our setup we used docker, thus runC is integrated. Cri-o [157], would have been a Kubernetes-specific, lightweight alternative, to docker. It directly runs over runC containers, thus it may save extra memory and CPU since it removes one extra layer in the container image. However, due to limits in its extensions and complexity of the installation, we preferred to create a live migration solution on the default CRI installation.

In the following sections, we present possible use cases for Live migration in Kubernetes and further detail our solution.

3.3.1 UCs for Live Migration in Kubernetes

Live migration would let one reduce downtime for services that do not support fail-over and have a long start-up time.

Theoretically, a statefull software could run under a single-instance since in Kubernetes there are StatefulSet: a single-instance stateful application using a Persistent Volume and a Deployment. The stateful deployment configuration will keep application states even if Pods have to restart. However all restarts would be very slow due to the rolling cluster reboot: when in need to rescale or balance the deployment, Kubernetes incrementally deletes all the Pods controlled by a Replication Controller (RC) and allows the RC to recreate them elsewhere.

To avoid time-consuming reboots, one could live migrate the state of a running instance, stop it, and restart on the target replica.

The main reason to use migration in a Edge network is however related to the user mobility: the application follows the user when he moves from one Edge site to the other. We synthesize other use cases for live migration in Fog network in the following three scenarios:

- Load Re-balancing: to replace an unreliable or under-resourced node, a cold swap may cause service disruptions. Instead, one could mark the node as unused and gradually phase it out. An internal scheduler could automate the migration of components that are too costly to simply restart. Also dynamically switching node locations for cost-effectiveness would become feasible.

- Hardware Maintenance Without Downtime: in case of need, some Edge server could be substituted by a neighbour once until maintenance is complete without compromising the service.

- Affordable Redundancy: replicas are usually needed to guarantee service coverage even in case of major disruptions to the infrastructure. For a network where the demand is high and the available resources are low(small bandwidth, limited CPU) replicas may be expensive to maintain. Similarly, a shared file system would be too heavy on the network traffic. Instead, a reduced amount of replicas, able to perform live migration, would only require a smaller amount of network resources for a far limited time.

3.3.2 Experimental Setup

In this section, we integrate live migration in a Kubernetes cluster moving a docker state from one Pod to another.

Our environment is composed by an Edge server with 5G non-stand-alone connectivity, where the K8s master resides (an Ubuntu 16.4 VM instance vCPU: 4 Disk: 80GB RAM: 8GB).

Kubernetes minion nodes are two raspberry-pies(models 3B and 3B+) and an ubuntu dell (Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz) connected via 5G modems to the server.

We have an OpenVPN Certificate Authority and Server on the Edge, so that we can join the nodes with their virtual address to overcome NAT.

We deploy a DinD container in our cluster for each node as a Stateful set. These nodes will be running the docker container we would like to migrate. For instance a busybox running a counter script or a minecraft server (the server container would be migrated, while its persistent volume and the related services are still under K8s controls).

DinD works by running a Docker daemon inside a Docker container. Containers created with the DinD daemon are not visible to our original host Docker daemon.

All tasks are deployed to the Fog nodes using our customized scheduler, as we describe in [7].

For the sake of simplicity, we will show examples for migrating a single natural number generator container, that counts from zero to infinity, however in the test section a table summarize the different containers we experimented with. The application is wrapped in two nested layers: a Kubernetes Pod and, within the Pod, a docker container *DinD* running a local daemon.

The Kubernetes Pod groups together all the sub-tasks necessary for the migration process in form of containers: an *Initiator*, a *Checkpointer*, a *Migrator*, and the *DinD* itself (See Figure 3.10). The solution is nested because the *NNG* container will be created and run by the Docker daemon inside *DinD* only after Pod A has started.

The experiment is executed on multiple machines, each a "Fog" node of our Kubernetes cluster.

With DinD we created a simple cluster with a master node, from where Kubernetes performs orchestration tasks; and two workers where the Pods, among which we want to move our application, will be deployed.

After that, we implemented a sidecar container solution in a Kubernetes Pod that contains a DinD container. Applications and services often require related functionality, such as monitoring, logging, configuration, and networking services. These peripheral tasks are implemented in a sidecar pattern: separate components or service inside their own process or container, they provide an homogeneous

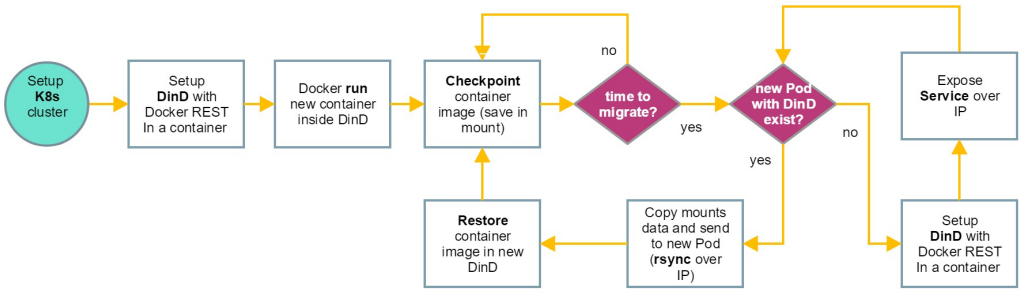


Figure 3.9: Live Migration - a simple flow

interface for others to access to them. In our case the DinD container provides a docker daemon service so that other containers can instantiate migratable ones in it.

The logic of the migration process is synthesized in Figure 3.9.

The Docker daemon can listen for Docker Engine API requests via three different types of Socket: unix, tcp, and fd. To access the Docker daemon remotely is enough to enable a tcp Socket.

At configuration, the DinD container is setup so that the Docker daemon starts on the path of a Kubernetes mount. Through bind mounting, we give all the containers in Pod A full access to mounted resources of the DinD daemon.

So in the container description, DinD must be specified as a privileged container and the mount should be declared both in the container and in the Pod.

```
        ...
        securityContext:
            privileged: true
        volumeMounts:
          - name: dockermount
            mountPath: /var/lib/docker
        ...
volumes:
    - name: dockermount
      emptyDir: {}
```

This allows for other containers in the same Pod A, to be able to create and destroy containers in DinD using REST requests over the localhost (docker run calls with p 80:80) , and if they share that mount, to be able to create, destroy containers inside DinD and migrate them to other Pods.

In Pod A, the sidecar container, DinD, starts a Docker REST service on a fixed port (as shown in Fig.3.10). This container will serve as a server, locally exposing all daemon services. It is conventional to use port 2375 for un-encrypted communi-

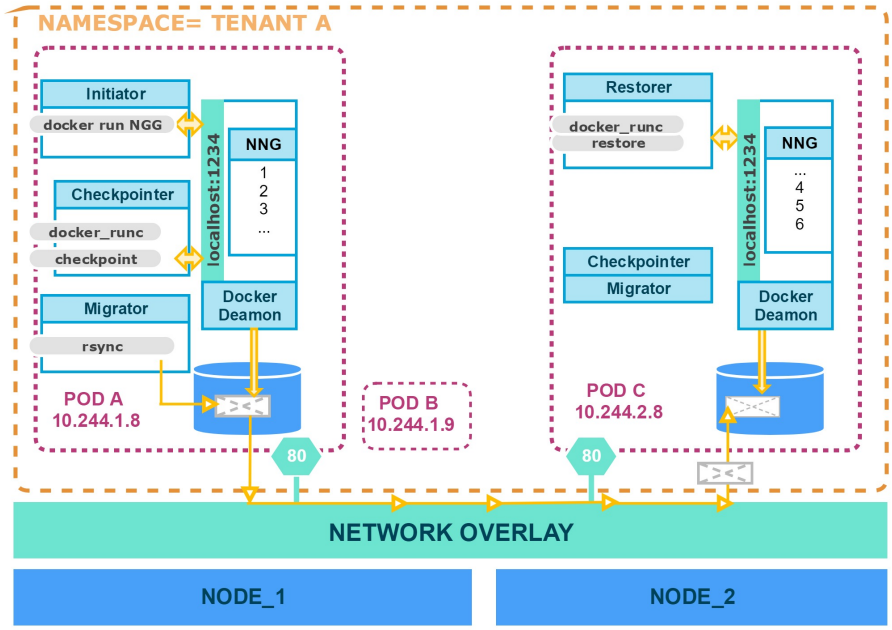


Figure 3.10: Live Migration in Kubernetes

cation with docker daemon. The Docker clients will honor the *DOCKER_HOST* environment variable and forward request to the specified address. This means that the containers in Pod A will have an environment variable *DOCKER_HOST* set to *tcp://localhost:2375*. With this configuration, any container in Pod A will have Docker binaries that point to the Docker daemon on the DinD container. In other words, we enable calls to the daemon through localhost.

The initiator container in Pod A can then issues Docker commands to start a container.

The container, in our example a Natural Number Generator, will run as child processes of the DinD daemon and inherits its CPU and memory constraints.

The NNG in our case is just a simple while loop in a busybox image that now runs inside DinD.

The yaml file configuring Pod A will have the initiator container described like follows:

```
...
- name: initiator
  image: docker:1.12.6
  command: ["/bin/sh"]

  command: ['docker', 'run', '-p',
  '80:80','-d','busybox:latest',
  '/bin/sh', "-c"]
```

```
args: [i=0;while true; do echo $i;
i=$(expr $i + 1);sleep 1; done]
resources:
    requests:
        cpu: 10m
        memory: 256Mi
env:
  - name: DOCKER_HOST
    value: tcp://localhost:2375
```

It is important to notice that, even tough NNG is created after Pod startup, since it shares resources with DinD it will still die with it in case of relocation of Pod A. In case of removal of the Pod also NNG resources or eventual storage will return under Kubernetes.

The checkpointer container regularly, in our case every 20 seconds makes a runc call to the DinD to make a checkpoint of the running container, and to write it into the shared mount.

To setup a shared mount is enough to declare a volume and in all involved containers declare a mount, the mounts path can differ between each other as long as the name is the same.

```
volumes:
  - name: shared-data
    emptyDir: {}
  containers:
    ...
  - name: checkpointer
    ....
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/
```

The migrator container is in charge of sending data trough Kubernetes IPs to a replica of our Pod in node_2. Moving the checkpointed state, the container and the shared file resources between physically separated pods(that is residing on different physical machines) can be carried out in many different ways, as through simple http posts, rsync or even shared filesystems. Our migrator will be sending data through http using Kubernetes IPs to a replica of our Pod in node_2, where the restorer container is listening.

In our image (3.10) the external IP address of Pod C will be 10.244.2.8.

To share data among nodes without a shared filesystem, we have created a service (e.g. my-service): a Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them. Similarly, for the local host set up, we created an environment variable mapping what port the restart service listen to.

To enable Pods communication also an 'endpoint' must be created. The endpoint name is the same as that of the service name. Kubernetes will handle the service discovery process and only the port reference will be needed. Pods A and C

use the endpoints and the service: if migration and restore of the image in Pod C is needed, it will be POSTed by Pod A to the endpoints object called my-service.

The restorer will have to expose a service to get the migration data from the migrator. This service (as any services) has two IP addresses - an internal IP (also called cluster IP, for communication with containers in the pod) and an external IP that can be exposed outside. Port forward today only works for TCP protocol.

In the service definition the target portshould be the same as that of app-port. This means that the service exposes Pods who have the port declared as app-port. Thus, if Pod B or Pod C have ports as app-port, the service will expose these Pods.

So internally, it is exposed as *<cluster-IP>:80*.

The restorer as any previous container has access to a mount shared within all containers on Pod C, where the task should be migrated. It listens for calls from the migrator, takes the data and saves them in the local mount, calling to the local DinD to restart the container in its own resource space.

To reduce the number of containers interacting with DinD, initiator, checkpointer and migrator in our example could be all part of a single internal scheduler in charge of the whole life cycle of inner containers and of relocating the "live" part of the Pod.

We performed several test with different deployments, trying to variate in image and checkpoint size. To select images we took inspiration from possible fog scenarios, were resources have a reason to join and leave the network:

- **Mysql**: basic shared database scenario
- **Minecraft**: gaming server between mobile peers(eg. friends hosting a turnament, one of them hops on a train)
- **Stanford corenlp**: distributed AI learning (eg. a smart car fleet)
- **Jenkins** and **Jira**: collaborative tool (eg. laptops coming and going from a workplace)

In Table 3.2 we report average times of deployment and migration. Since the scheduling time and pod creation would be identical in both scenarios, we did not consider them. Deployment only represents average container creation time. Equally, for migration, we used two existing pods as described before, and only calculated the time of checkpoint, scp (migration) and restore of the container. Each migration run over a container with 20 seconds lifetime. In most cases time spent to restart the container is often shorter compared to the start time, with the added value that the container will start exactly at the state were it left. The overall migration time is influenced by the checkpoint size and the checkpoint migration in itself.

Table 3.1: Applications deployed in our tests

Application	**Image size MB**	**Checkpoint size MB**
minecraft server	68.53	0.328
mysql	151.88	9.29
jenkins	285.8	84
jira	482.5	20
stanford-corenlp	1500	4.96

Table 3.2: Comparison between k8s container deployment in pod and migration from one existing pod to the other using CRIU and Dind strategy

	Average start and run time (s)			**Average Migration time (s)**			
Application	**start**	**run**	**tot**	**checkpoint**	**migrate**	**restart**	**tot**
minecraft server	23	20	43	4	1	13	18
mysql	16	20	36	5	1	14	20
jenkins	10	20	30	8	7	10	25
jira	31	20	51	7	1	10	18
stanford-corenlp	17	20	37	4	1	14	19

In our tests migration saved between 5s and 33s, introducing a communication overhead between 0.33 and 84 MB. On average, service disruption time is halved, if we compare complete migration with restart and run. As a worst scenario, if we compare only the start of a fresh container with the complete migration, the average time stays in favours of the migration, with exceptions of Jenkins deployment where restarting a checkpoint or a new image takes exactly the same time.

In situations where the throughput between nodes is higher in magnitude than the checkpoints size, the iterated pre-copy phases does not have a prolonged duration and can be an effective means to reduce downtime. In case of checkpoint size greater than what we had in the Jenkins example, our setup would have been better configured with an iterative migration based on post-copy [158].

3.4 Related Works

In the last years, the interest over service placement techniques has piqued, because of the many network paradigm that has arisen especially in connection with 5G

and Edge Computing.

Authors in [159] have modeled service replicas and migration in MEC. The overall conclusion is that service migration and replication should coexist: in case of short-duration applications, duplication increase the system performance by avoiding service migrations. For services of long duration, user mobility strongly require service migration.

Among the works on Service Migration for Edge Computing the paper [129] discusses two concepts similar to service migration: live migration for data centers and handover in cellular networks. Authors distinguish live migration in datacenter from what is needed for service migration in MEC. They argue that the former focuses only on reducing downtime, while the latter should attempt to reduce the overall end-to-end latency. The authors present an extensive taxonomy on the topic. They favor Agent-based solutions over VM and containers but stress how these technologies are still at a too early stage.

Also [136] shows a global classification of VM and VFN placement solutions categorizing them based on their target objective. The point of view of the analysis may be either the final user or the network providers. Recent works have been done also in the direction of applying containers for service migration. In [148], after a systematic study of Docker container layer management and image stacking, the authors propose a migration method that reduces file system synchronization overhead, without dependence on the distributed file system. The first evaluation result in a reduction of the total service handoff time by 80% with network bandwidth 5Mbps.

In [149] three different mechanisms are proposed and evaluated to improve the end user experience by using container-based live migration paradigm. The author privilege LXC over Docker because of their lighter structure. The described methods are the classical approaches for VM and distributed systems: temporary file system (tmpfs) and disk-less based lightweight container migration, and the shared file system.

Basic notions on the topic of containers checkpointing and live migration can be found in [130]. Also, [137] summarizes the differences between LXC and Docker and introduce the success of Kubernetes.

Related to our use case, in [160] a comprehensive description has been given on the advantages of the federated learning approach, with a range of applicable algorithms. In [161] the authors present a practical method, named Federated average, and through a series of experiments, they show that the idea of federated learning can be effectively and efficiently used in similar scenarios to ours, as defined in Section 3.3.1.

Authors in [110] worked on geographical distribution of VMs in cloud. Their software, Volley, performs automatic service placement based on iterative opti-

mization algorithms, and performs service migrations when detecting changes in data center capacity or user location. A spring model to simultaneously reduce latency and reduce inter-datacenter traffic is adopted: an update rule iteratively moves nodes with more communication closer together. Volley does not attempt to migrate the data, but leaves the migrations specifics to the application.

To the best of our knowledge, there is no effort in the literature to integrate live migration in existing orchestrator platforms such as Kubernetes or Mesos.

Work on improvement of resource scaling using live migration have been done in [162]. ELASTICDOCKER scales CPU and memory assigned to each container according to application workload. As vertical elasticity is limited to the host machine capacity, ELASTICDOCKER performs container live migration using CRIU in case of insufficient resources on the hosting machine. Since ELASTICDOCKER improves performance when compared to Kubernetes autoscaling, it could be interesting in future work to experiment if similar techniques, using our migration scheme in Kubernetes, could as well improve autoscaling.

Multiple efforts towards stand-alone solutions, both opensource and proprietary, where made.

Authors in [131] present a live migration solution for Docker by means of logging and replaying iteratively the container. The paper is far from presenting a full orchestration and migration stack. This is one of the earliest efforts we could find on checkpointing containers. The results compared to a VM implementation where interesting, with downtime reduced up to 65% under different scenarios, while the total migration time diminished between 27% and 47%.

Also, [132] presented a generic checkpoint/restore mechanism evaluated with Docker. Their experiment and performance evaluation shows that the checkpoint and restore time scale linearly with the micro-service application image size. Again no complete orchestration solution is presented.

Container live migration specifically for Edge Computing and 5G is also a recent interest in the literature.

In 2016 [163] proposed Foglets, a programming infrastructure for fog nodes. It provides APIs for a spatio-temporal data abstraction for storing and retrieving application data on the local nodes; as well as primitives for communication among network resources. Algorithms are presented for launching application components and handling their migration between nodes, based on the mobility pattern of the sensors and the computational needs of the application. The implementation uses Docker containers and RocksDB [164]. Compared to our pattern proposal for Kubernetes, in Foglets all instantiated containers will have to use a Foglet image as a base layer.

In 2017 [148] propose a framework that enhances service handoff across edge servers by leveraging the layered storage system of Docker containers to improve

migration performance. Iterative migration is used to eliminate unnecessary transfers of a redundant and significant portion of the application file system. Base memory images are transferred ahead of the handoff, and only the incremental memory difference when migration starts.

Partially under the TAKE5 Finnish Ministry project, [165] worked on a framework which leverages MEC to support diverse applications in smart city scenarios. To always ensure high QoE, the Follow Me Edge concept is introduced. The authors prove better efficiency of container migration based on OpenVZ. However migrating a blank container has considerable latency, even using NFS and shared-async mode the average time is 10s. The Follow Me Edge stays as a framework proposal and is yet not available for orchestration. However, the authors present interesting findings: in a later publication from Take5 group [149] three migration approaches for LXC are investigated as an enabler of the Follow Me Edge concept. NFS approach delivered the shortest migration time, it also imposed the highest downtime. Meanwhile, the larger migration time was caused by the file system copy, which was done while the container was running. Iterative migration achieved a mean downtime of 1,042 s. Performance issue in the current implementation of CRIU's page server slowing down diskless migration was finally identified. We used this information in our work when deciding for Docker containers and to choose against NFS.

Authors in [127] presented a layered framework for service migration in Multi-access Edge Computing nodes (MECs). The framework supports both container and VM technologies, and the author claims it can be easily implemented using existing functionality of container and VM implementations. As a proof of concept, nested KVM and LXC containing running various applications were run inside the two host VMs mimicking MECs. Their results are based on the concept of sharing the same configuration base layer for all MEC nodes (OS, kernel etc.) to reduce migration size. Contrary to a Kubernetes implementation, the framework does not include orchestrating mechanisms and does not keep in consideration node heterogeneity and networks fluctuations.

Among customized orchestration for Fog [124] propose IoT gateways working as full fog nodes, by extending the existing base of open-source Kura IoT gateway. They employ containerization techniques, in particular, Docker, Docker Swarm and live migration to deploy measuring/sensors data collection services on Raspberry-Pis. Authors mention a Swarm based orchestration solution, which make us suppose a solution based on API calls to Docker Swarm functionalities from the Kura gateway. On our work, we specify a more generic solution that is immediately deployable on any cluster.

For completeness, we include live container migration solutions in shelf products platforms. As mentioned in Section 3.2.3, those are meanly Virtuozzo and

Jelastic.

Virtuozzo [145] leverages OpenVZ containers and can thus apply live migration using NFS. It is optimized for hosters and offers hypervisor (VMs in addition to containers), distributed cloud storage, dedicated support, management tools, and easy installation.

Jelastic [146] is a cloud services provider that combines PaaS (Platform as a Service) and CaaS (Container as a Service). Jelastic provides custom Docker containers. Live container migration is integrated into the platform services as well.

3.5 Conclusions and Future Works

Current orchestrators for mobile networks cannot handle the complexity added by new paradigms such as Fog computing. Having to integrate and manage such peer-to-peer and ad-hoc solutions requires the introduction of: (A) mapping of available and mutable resources, (B) creation of a dynamic deployment plan, (C) infrastructures to guarantee codes and application location assignment, and (D) the strategy to maintain the health of the service; especially the shared state of the application.

In this chapter we described some of the current technology, their benefit and how we combine them, to build a framework tackling all four of these needs.

We advocated that implementing live migration can help to achieve zero downtime even in situations when redundancy is not an option, like in the case of hardware maintenance or for re-balancing low latency critical services.

We presented two application scenario leveraging on the current container orchestration technologies: one on Docker Swarm and another on Kubernetes.

Our checkpoint and restore of containers relies on CRIU. The library facilitates the migration process but has multiple restrictions [166]. As of today, the migration process seems more error prone than VM live migration: since checkpoint and restoring are two separate processes they need to be implemented in scripts and if resuming fails the last migration should be redone. However, the smaller size of the average migration and reduced time still work in favour of container solutions.

IP addresses used by the applications should be available on the destination host: when restoring TCP sockets, CRIU will try to bind and connect them using their original credentials; if IPs are not available the system call will fail. In our case, however, the app lives in a net namespace (a container). In this case, CRIU will call action scripts to lock the network and the Docker daemon will handle it through the libnetwork library.

In CRIU usually restored processes have the same PID as the originals. This prevents to migrate processes already existing at the destination host. This is one

strong point to involve Kubernetes in the migration. Kubernetes can handle IPs and namespaces, so we could work around this limitation.

Another reason to rely on an orchestrator was that CRIU does only plain snapshot/restore of the process. No cleanup is performed: no rundown of the application, no disconnection of network connections and no files is closed. In a Fog network garbage collection and redistribution of resources would be needed.

In the Docker Swarm case we described a Federated Learning case scenario. At present the orchestrator functionalities Network mapping, Task assignment and Deployment are fully implemented. In the next iteration, we would like to optimize the current migration strategy, based on experimental Docker CRIU calls, with what described in Migration Support (Section 3.2.4).

In the Kubernetes work, we presented a possible integration of Live Container Migration patterns into Kubernetes. The purpose of such an integration is to allow Kubernetes to exploit highly volatile resources at deployment of latency critical applications in Fog networks.

Migrating docker containers outside Kubernetes would have nullified all the advantages of having an orchestrator. For this reason, our solution uses docker in docker and sidecar container patterns built upon Kubernetes overlay network.

Other advantage of our proposal is that containers created inside the DinD container are still exploiting Kubernetes capabilities. They are reachable through localhost since containers running in Kubernetes Pods share the same network namespace; the Pod IP allows reachability from all the nodes in the cluster (fixed path), thus the migration process is simplified.

Kubernetes garbage collects Pods after they are terminated, preventing nodes from running out of space. Ideally having a one to one mapping from "side" DinD and "car" container, will keep this benefit.

CPU and memory resources will be inherited by containers inside the DinD. Finally, there will be storage cleanup if the Pods is removed or terminated.

The cons of our approach mostly derive by using DinD. As Jeŕomê Petazzoni wrote [167]: it is possible to do something reliable and fast involving multiple processes and state-of-the-art concurrency management with DinD; but it is simpler and easier to maintain, to use the single actor model of Docker. The risk we could incur in is that of nesting side effects and of a build cache shared across multiple invocations (possible data corruption).

DinD problems are solved in case of DooD (Docker outside of Docker). DooD uses its underlying host's Docker installation by bind-mounting the Docker socket. Even if DooD is the preferred solution because of simplicity and lesser risk, we advocate that it infringes the principle of run-time confinement and self-containment. In fact, DooD involves creating containers in the host node, bypassing the Kubernetes infrastructure. In this sense also image immutability through the cluster

cannot be guaranteed, since Kubernetes will have no awareness whatsoever of the container existence. Deploying migrating Docker containers outside the domain of Kubernetes also requires a completely separate orchestration and reintroduce all the complexity previously discussed.

The need for live migration integration for some cloud-native services still justifies our attempt at a safe setup, involving a single instance of DinD per Pod.

A live migration feature in Kubernetes will allow us in the future to rethink scheduling and resource deployment on a finer grain, without compromising user quality of experience. Services could be redeployed more often according to the state of the network without the downside of introducing continuity disruptions.

Live migration should be included in a Fog platform to compensate the inherent high mobility of the nodes, for all those scenarios that cannot be solved with a stateless service deployment.

Our solution relies on Checkpoint and Restore and is thus limited by it. The library presents multiple restrictions [166] that are continuously being addressed by the community. In our scenario the most significant one being that CRIU calls for containers in privileged mode. We also experienced that different machines/environments and images can incur in diverse errors at checkpoint and restore time. To reduce errors we plan to shift the implementation to newer container daemonless solutions, such as podman [168], so that root mode will not be a necessity. We argue that with a relatively low impact on the network, our strategy of migration can still be convenient in therms of time saving and reduction of service disruption.

Framing migration inside a platform such as Kubernetes allows us to compensate for migration failures, since we can fall back on preexisting high-availability mechanisms such as an N-Way Active based redundancy [169].

A live migration feature in Kubernetes will allow us in the future to rethink scheduling and resource deployment on a finer grain, without compromising user quality of experience.

For future work we are investigating rescheduling of services, were we exploit live migration and trigger pods re-deployment according to the state of the network and the health of the cluster nodes.

3.6 Contributions

In this Chapter we collected our work concerning Thesis 2. I made a study and comparison between VM and container technologies. I tested and compared different orchestrators. I ideated a framework to enhance Edge computing with container migration. I tested it on a real 5G Edge setup. I created a simulation of peer to peer Fog nodes that can join an Edge cluster and send each other tasks. After migration is triggered it exploits Docker swarm and CRIU.

I defined possible reasons and application for container migration in Edge computing. I created a live migration flow for Kubernetes. I tested the migration flow in an implementation on top of a 5G network.

I verified that the container migration reduces downtime for most of the services tested.

Figure 3.11 summarize the main concepts and contributions.

Thesis 2 (Migration in Edge Computing)**.** *In Edge computing it will be necessary to move an edge-native application from one edge to the other, following the user, or, for recovery reasons, even from an Edge to the Cloud. The best approach to avoid this would be to have Stateless Apps, but there are cases in which this is not a possibility.*

I implemented a Java simulation of a Fog network, integrating live migration trough means of CRIU library calls and coordination via Docker Swarm. After testing the limits of this configuration I set up a better version based on Kubernetes. I constructed a docker in docker solution that is deployed as a daemon set on each node. This component is in charge of actuating iterative pre-copy migration of its hosted containers, if triggered by any other software running in the cluster. I have tested the migration against stop and restart of multiple container deployments and shown how, given a stable network connection (like a 5G coverage) we can use migration to achieve a lower Service Downtime. .

thesis	relevant publications							
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
(1) Facilitate Deployment at the Edge	◦	◦	◦					
(2) **Migration in Edge Computing**			•	•	•			
(3) Distributed Mobile Edge Scheduling			◦			◦	◦	◦

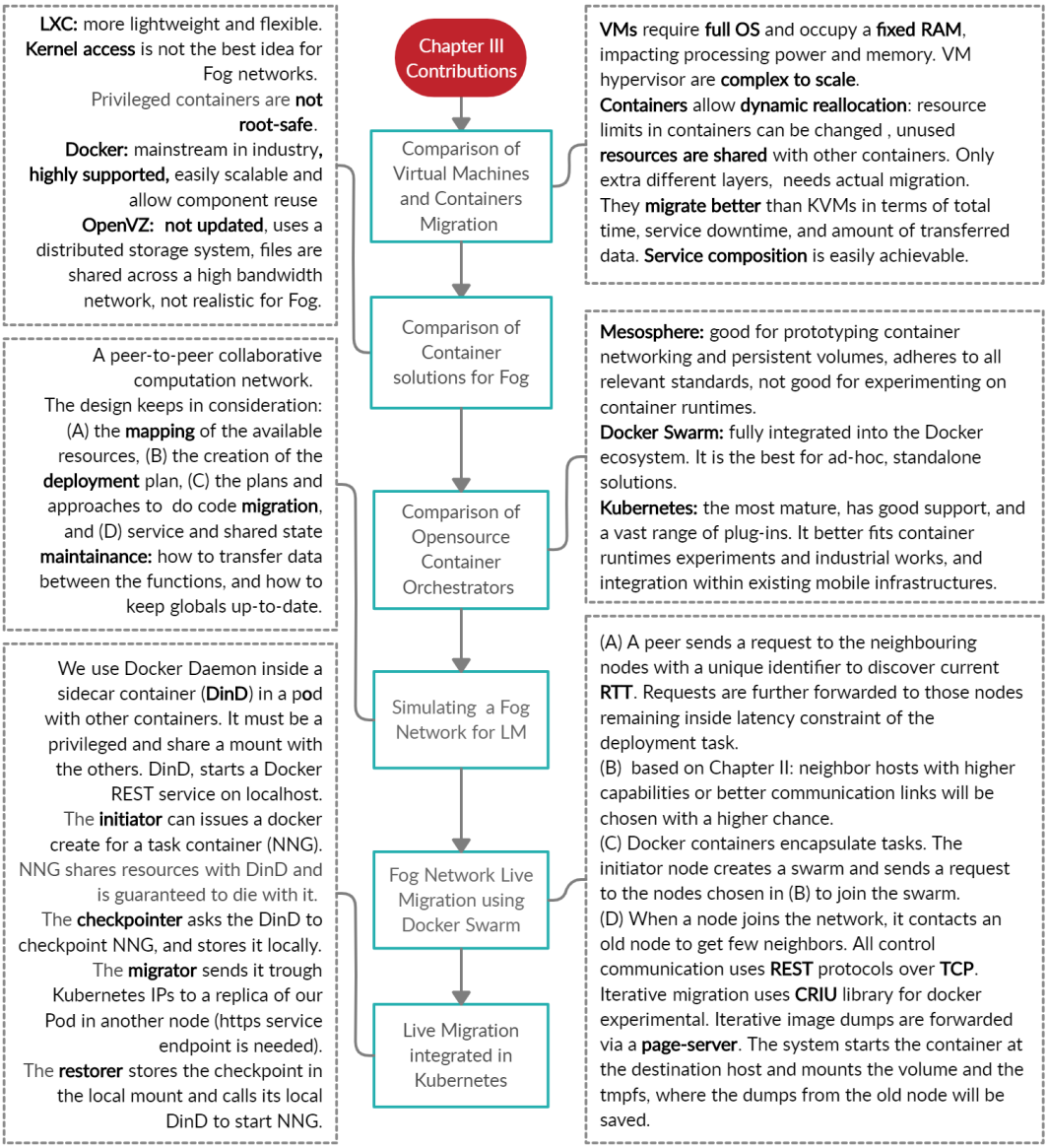


Figure 3.11: Summary of Chapter Contributions

Chapter 4

Scheduling for Distributed Mobile Edge

In many Edge scenarios, especially those related to distributed Edge and IoT, it is for the best to reduce the need for application migration from the start.

The orchestrators built for cloud resources usually oversimplify the allocation task, since they do not contemplate high application mobility or great variability of cloud resources.

Reducing rescheduling and frequent reallocation of resources is instead crucial in case of smaller and heterogeneous realities, such as an industrial Distributed Edge scenario.

We can reduce need for migration trough the improvement of the deployment strategies. Improving the scheduler component of the edge orchestrator via dynamic memory allocation can boost the health of the Edge cluster and reduce ping-pong effects between neighbour Edge nodes.

4.1 Motivation

Distributed Mobile Edge Clouds are Edge resources across different physical machines, located in a relatively close distance. Example scenarios for this kind of on premises setups come from the Industrial IoT (IIoT): collaborative robotics, remote robot controlling, flexible reconfiguration of manufacturing pipeline, etc.

A Distributed Edge introduces new challenges for the involved resources management functions.

It is likely in these scenarios that an Edge instance on the premises will host not only the software in charge of mobile connectivity, but also the orchestration logic.

The general role of an Orchestrator is to handle the applications life-cycles (ac-

tions like service binding, querying, copying, updating and deleting). Cloud-native Orchestrators badly adapt to the job. Their scheduling configurations are usually simplistic since they seldomly need to handle distribution of the computational resources: quite often Virtual Machines with very similar baseline characteristics and hosted in the same datacenter. The devices that will compose the orchestrated cluster in our IIoT case scenario are diverse in capabilities, resources, availability, life-time, supported temperatures and so on.

Cloud-native orchestrator solutions need to be adapted and customized towards this context. In the next section we will briefly explain how IIoT applications benefit from 5G and Edge computing infrastructures and what characteristics of current orchestration solution made us verge towards Kubernetes as a starting point for edge-native applications orchestration in IIoT.

The Chapter is structured as follows: after a deeper introduction to the objectives (Section 4.1.1), we describe the Kubernetes Scheduler (Section 4.2). In Section 4.3 we present our implementation and finally our tests (Section 4.4.3). The Chapter ends with the Related works and Conclusions (Sections 4.5 and 4.6).

4.1.1 Problem Statement

Scheduling resources and jobs on a distributed Mobile Edge requires awareness of the Edge Context (Network and Nodes resources). In this Chapter we try to adapt an existing Orchestration platform to enable optimal Edge scheduling.

More in details, the objectives of the chapter are to:

a) define how to ensure deployment and orchestration of edge-native applications when the cluster is spread among communicating devices at the edge, keeping in mind the peculiarities and challenges of such a new infrastructure.

b) describe our verification setup in details. How real 4G and 5G connectivity work together, the 5G New Radio and the concept of non-standalone (NSA) 5G core network.

c) present a lightweight CoAP-based (Constrained Application Protocol) tool, to collect cluster metrics, able to traverse Network Address Translation when UEs act as modems for devices in the network (i.e. tethering);

d) introduce a Kubernetes compatible scheduler, based on real-time application context information, with an aggressive scoring mechanism to prioritize the balance of resources usages all over the cluster.

4.2 Kubernetes schedulers

As stated previously in this thesis, Kubernetes (K8s) can be considered as an orchestrator since it is able to automate deployment, scaling, and management of containerized applications. It is important to state that from March 2019, a stripped down version of Kubernetes for IoT and Edge application has been released: K3s [170]. Scheduling, network and cluster logic are kept the same and only the kubelet size is significantly reduced through a reorganized plugin structure and a less resource-intensive database (sqlite3). When faced with the choice between K3s and K8s, the latter was still preferred, considered the following:

1. The K3s project was fairly too recent and not production-ready, not enough information was available online;
2. The scheduler component was identical in both Kubernetes;
3. K3s does not allow multiple masters, so if the master goes down the whole cluster is lost;
4. a K3s cluster is not compatible with K8s, adding unnecessary complexity in the case of multilayered orchestration, to enable aggregated control of clusters located in distributed facilities.

An other scheduler option could have been Poseidon/Firmament scheduler [171, 172] that incorporates flow network graph based scheduling and applies min-cost flow optimizations. Due to the inherent rescheduling capabilities, the scheduler enables a globally optimal scheduling (Single-step scheduling) and high scalability. Focusing on a global optimum for an allocation process complicates and slows down the scheduling algorithm, but it reduces the necessity of rescheduling and maintenance phases. According to [172], Poseidon supports high-volume workloads placement and complex rule constraints. It achieves a 7X or greater end-to-end throughput than the Kubernetes default scheduler, as long as resource requirements (CPU/Memory) for incoming Pods are uniform across jobs. Due to its characteristics this scheduler is more adapt to Big Data or AI jobs were with a large number of tasks the throughput benefits would be high. Currently, Poseidon-Firmament scheduler is in alpha release, does not provide support for high availability, and is not used in any production deployment [173]. This scheduler would be interesting as a stepping stone for implementing the algorithm proposed in Chapter 2; yet this was not a good option for our IIoT use case, so our final choice remained K8s default scheduler.

For our specific scenario, we found that the most lacking orchestration feature in K8s was related to the scheduling techniques themselves. As we will show later in this Chapter, and as we stated in [6], in a distributed IoT cluster scenario, the

amount of time spent scheduling and rescheduling a pod may grow significantly, with service disruptions up to a minute in case of node death. In the default K8s scheduler nodes are evaluated at scheduling time; when the user creates a new pod and assigns it to the Kubernetes cluster, the pod gets into the event stream (Object Store) with a "Pending" status. The scheduler watches constantly for this type of event and decides the most appropriate binding to a node.

At first, a subset of *feasible schedules*, containing only those nodes that are satisfying the given constraints for the deployment are collected; Then the scheduler computes a subset called *viable schedules*, which ranks the selected nodes based on scoring functions:

1. **PodFitsResources**: If the free amount of CPU and memory on a given node is enough
2. **NoDiskConflict**: if a pod can fit due to the volumes it requests
3. **NoVolumeZoneConflict**: checks possible zone restrictions.
4. **PodFitsHostPorts**: check if the needed port is free
5. **CheckNodeMemoryPressure** and **CheckNodeDiskPressure**: if a pod can be allocated on a node reporting memory pressure condition or disk pressure condition.
6. **MatchNodeSelector (Affinity/Anti-Affinity)**: By using node selectors (labels), it is possible to define that a given pod can only run on a particular set of nodes or that it cannot be allocated on a node that has already certain pods deployed (pod-anti-affinity).

The scheduler also uses "general-purpose" cloud computing scheduling criteria, called priorities: for example **ImageLocalityPriority** ranks according to the location of the requested pod container images.

A more clear schema of the interactions among K8s components during the default scheduling, is shown in Figure 4.1.

The default scheduler that we compare to acts mostly as a rule based entity:

1. first it determines all the nodes that exist and are healthy
2. then runs the predicate tests to filter out nodes that are not suitable
3. finally runs priority tests according to a spreading function, so that candidates are ordered by a score influenced by few parameters such as if the pod image is already present or if there is not already a duplicate of the service in that location.

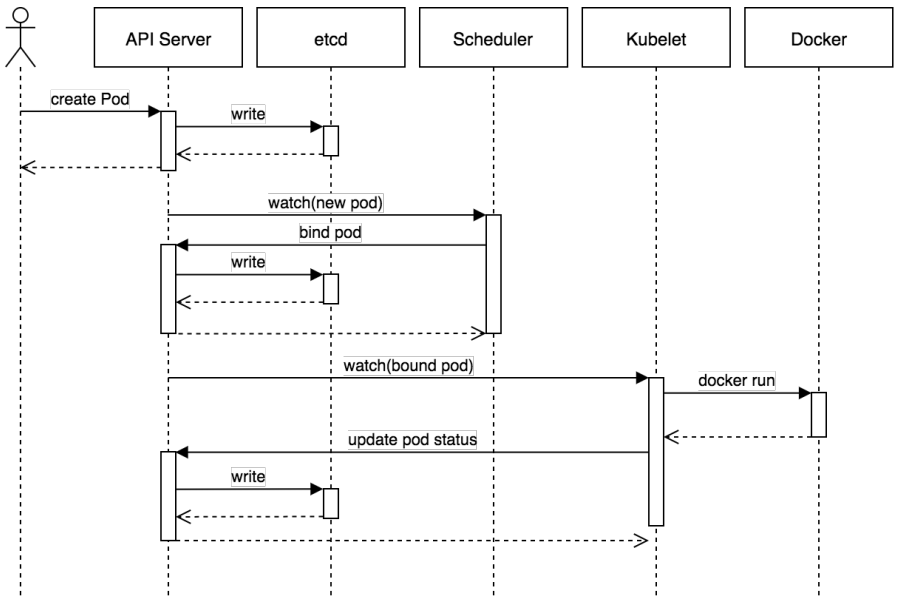


Figure 4.1: Default K8s scheduling process [13]

4. If nodes have the same score they are selected in a round-robin fashion

Our scheduler implementation modifies the last two steps of the scheduling phase. Furthermore it is a hybrid between multi-step [174] and single-step scheduling. Multi-step scheduling is the approach of K8s default scheduler: each pod is considered independently to provide a local optimum for each allocation process. This solution mitigates the complexity of having multiple parameters involved in the decision process. Overall it requires more maintenance processes, especially in case of rearrangements based on pod priorities, pod preemption and reassignment.

In our case we do not perform exactly Single-step scheduling since the optimum is calculated every batch of pods.

4.3 Edge Cloud Scheduler Implementation

There was a possibility to instantiate the custom scheduler as a pod on the master node to profit from the virtual network built up by K8s. Our version does run in the master node of the cluster, however, it is outside of K8s control, to avoid delays and interference from the default scheduler. Hence we preferred to handle the communication separately and make use of an external asynchronous module, so that the scheduler itself can operate without an excessive computational overhead.

The status data fed by the nodes of the cluster are input to the scheduler for node score computation. The requirements of a Pod also influence the choice.

The scheduling mechanism is able to dynamically adapt to the changes, and its fair since it distributes workloads based on the actual node status.

The motivation behind our approach also relies on the observation that the existing scheduler does not fully satisfy the need to preserve the health of the cluster and its services. Sharing physical resources among containers might lead to a degradation in the performance of the applications running inside them [175]. Kubernetes resource reservation mechanism is only available for CPU and RAM. However other resources are shared such as bandwidth and network access [176].

In the following paragraphs, we will first describe our tool to collect information about network and cluster context (Section 4.3.1), finally the scheduler algorithm described (Section 4.3.2) : first the task prioritization and candidates selection are summarized, then the new score computation rationale is presented.

4.3.1 Monitoring agents

The limited resources on our minion nodes required a stripped-down solution to monitor their resources. Kubernetes monitoring solutions mostly employ greater resources since they rely on multiple containers and more complex networking. In our solution monitoring agents were added to the minion nodes for reporting runtime data of the edge devices to the edge-cloud network management system. The agent uses CoAP [177] for the communication between client and servers. When gathering data, the master node acts as the client and edge devices act as the servers. Low message overhead, low latency and high efficiency in comparison [178] with other IoT communication protocols such as MQTT and DDS, made us verge toward this solution. CoAP follows a Client-Server model, our solution takes advantage of an asynchronous function facility in python, *asyncio*, which facilitates execution of concurrent operations. The client sends multiple requests to all the connected servers and collects the responses, then exposes the collected data on localhost. This simplified service is always available for the custom scheduler to gather data simultaneously from all nodes.

Such a solution is completely independent of K8s, so it can be replaced and maintained without having to update the cluster, which is often an expensive and error prone task.

The client can choose what parameters to request, via command-line. The frequency of the collection of all parameters can be adjusted dynamically. On the edge devices, a server-like application handles GET request, and is able to measure the temperature of the CPU and its usage; total amount of free and used physical memory, and total amount of swap memory. Network parameters are: latency, jitter and packet loss. In case of Network Address Translation(NAT), a *login* mode can be activated to initiate an ACK sequence from the device to the master. This will ensure that any farther communication can transverse NAT in

both directions.

4.3.2 Customized scheduler

An allocation process begins when the user issues the creation of a Kubernetes entity (Pod, Deployment, Service). The entity gets into the Object Store and the scheduler can manage it through the Kubernetes API (every action performed in the cluster passes through the API, there is no direct interaction with the entities of the system). At this point, if the pod is set to use the custom scheduler in its specification, it will be assigned to it. This step does not consist of a real assignment, as it is actually the scheduler that has the role of continuously watching the Object Store for events triggered by Pods that by specification should be related to it. To integrate to the default scheduler our customized version will fetch nodes and pods through Python Kubernetes APIs, then select pods with "phase=Pending" and matching in the *schedulerName* property.

The'hybrid' single-step scheduling optimizes the pods globally, as a single-step scheduler would do, but it categorizes the pods and arranges them locally.

The specification of resource requirements in the Pod influence the allocation. Memory and CPU and any other custom label can be easily specified for this phase, just like in the default scheduler. However the network owner will be able to select also connectivity and other network related criteria to influence the placement.

The scheduler will however take also inputs data from the metric agents and its finals scores may be influenced by the infrastructure owner as he sees fit. Memory can be specified as the minimum that has to be guaranteed or the maximum that is allowed at run-time. CPU will be specified as the number of cores needed by the software to run properly. In the simpler case of Pods with no requirements, the best nodes are taken in order.

If there are requirement set, the total resource requirement of all the containers for each pod is computed. The list of pods to deploy will be ordered from the most resource-intensive pod to the least one. Pods with no requirements remain in an untouched order on a separate list.

Ordering by resource consumption is a powerful way to simplify the knapsack problem [179] towards a greedy solution: remembering that the available nodes are lowest first, the optimization is fulfilled by binding the'biggest pod' (the first in the requirement ordering) to the best node in the list that can accommodate its requirements. After that the record about the free available memory in the node is updated.

Nodes are first classified based on their *Usability* for that specific scheduling time: Liveness (is it responding to ALIVE probes), CPU and Memory (are enough and available for the job).

If, for example, there are pods requesting only CPU and others both specifying CPU and memory, two lists will be generated:

CPU only – The CPU usage values are compared for all the nodes. The lowest value get the highest priority;

CPU and MEM – The values of the CPU and memory usage are simply summed to build the score. The scores all the nodes are then compared. To build the *priority list* nodes receive a score based on their run-time state.

In the investigation from [180] multiple automated placement decision algorithms are compared and ranked. Among those, our improved score computation solution is close to the Coefficient of Variation (CV) weighting method, combined with Multiplicative Exponent Weighting (MEW). This upgrade was done based on the observation that this combination was the one resulting with a greater influence towards highly volatile parameters.

This score is computed as a linear composition of their utilization state properties (e.g. CPU usage, memory usage, core temperature, etc.), where the minimum value correspond to the best candidate(s). The properties that are considered are formulated so that their minimizations correspond to an increased health for the system. Properties like reachability are also taken into consideration, expressed through parameters such as network latency and failure rate.

The first iteration of experiments used the score as:

$$score = \left\lceil \sum_{i=1}^{|P|} p_i w_i \right\rceil, \ i \in \mathbb{N} \tag{1}$$

This formulations was not normalized and had a tendency to grow too rapidly. The final score formulation become:

$$score = \prod_{i=1}^{|P|} \left(\frac{p_{min}}{p_i} \right)^{-w_i} \tag{2}$$

For each parameter related to the context of the scheduling, $P := set\ of\ parameters$, the stream of values received from the devices is a time series data. The weighted scores will be $W := set\ of\ weights\ of\ the\ parameters$ with $|W| = |P|$; so that each p_i parameter data from a node device, is normalized by the minimum value it historically reached for that node; and with w_i the weight assigned to it.

The approach is performed by implicitly trying to minimize the utilization of each node, at each scheduling step (a minimization function where the minimum is dynamically adjusted based on the current workload).

For this multi-objective problem the weight aggregation strategy is:

$$w_i = D_i = \frac{\sigma_i^2}{\mu_{i,n}} \tag{3}$$

The weight is the index of dispersion of the values taken by the parameters of a node at run-time. The scheduler will use the weights to compute how compromised is the operational state of a certain node.Since the computation is based on time series, the longer the scheduler runs, the stronger the influence of the diverging parameters will be, ensuring the balance between the different objective functions. The scheduler can learn, during execution, which parameters should be considered with more importance, compared to others, and will optimize the allocation of computational tasks while also updating the values of the weights with the results of each scheduling process.

Conceptually, the more the values of each parameter fluctuate during run-time, the higher the value of the index gets, which means that the corresponding parameter will have a stronger contribution on the computation of the score.

Since the history of the parameters is linearly growing we simplify the complexityvia an *online update*, or recursive estimation of the mean, as shown below:

$$\mu_{i,n} = \mu_{i,n-1} + \frac{(x_n - \mu_{i,n-1})}{n} \tag{4}$$

$$\sigma_{i,n}^2 = \frac{\sigma_{i,n-1}^2 (n-1) + (x_n - \mu_{i,n-1})(x_n - \mu_{i,n})}{n} \tag{5}$$

The benefit of this is that the length of the computation will remain constant throughout the run-time of the system, so its impact on the scheduling time performance will always be known and stable. This method will always converge, since it does not use cumulative sums.

4.4 Setup and Evaluation Criteria

The scheduler and the metrics component were implemented in two versions each. Several scheduling patterns have been explored, from simple methods such as round-robin allocation of each pod to the ranked nodes, to more sophisticated methods based on bin-packing algorithms for pods that have specific system requirements. Two different Setups where tested. An Old Setup [6] within WiFi and a New Setup employing 2 kinds of cellular networks: 4G and non-standalone 5G.

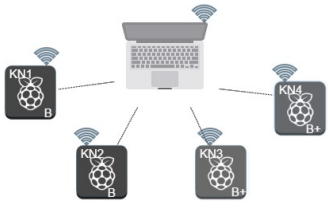


Figure 4.2: Preliminary setup

4.4.1 Old Setup: Baseline Experiments and preliminary results

The objective of the first testbed was to simulate an industrial setting where the devices composing the cluster would be heterogeneous and reflecting the ideas of Industry 4.0 [181].

The master node was an HP Elitebook 8560w, while the worker nodes Raspberry Pi minicomputers (models 3B and 3B+) as shown in Figure 4.2. Limitations of this first trial were that: the responsiveness of the cluster was reduced by the WiFi connection. The network setup was over-simplistic since devices were in the same subnet and had static IPs.

To simulate an intensive work, multiple batches of pods were deployed at the same time: first 2 replicas, then 10, 20, 30 and 40, The application was a small Python Flask web server. For higher numbers of simultaneous pods there was a risk that the cluster would start to resent both schedulers and nodes would shut down just because the requested resources were too high.

First experiments [6] also involved an application deployed on 2 nodes, in the form of 2 replicas of the same pod. Two scenario were tested:

Only one of the two nodes dies – The service doesn't respond when it is called and Kubernetes directs the call to the fallen node. After 20-30 seconds, Kubernetes recognizes that the fallen node is dead and a new replica of the pod is deployed to one of the remaining available nodes. Meanwhile Kubernetes directs the calls only to the working node. In this fashion, disruption of service should be prevented after 30 seconds, unless excessive request are sent before the replication is completed.

Both nodes die – It takes around 3 minutes for Kubernetes to decide to redeploy new replicas of the pod to the remaining nodes. During that period, Kubernetes acts as if the service was still available (even though service calls do not yield any response, since no application is available to take them). After that time new replicas of the pod are created and deployed, one after the other, with a 20-30 seconds interval between each other.

To compare the two schedulers we analyzed the pod distribution, the time to schedule and the delta of the temperature of the CPUs pre and post scheduling.

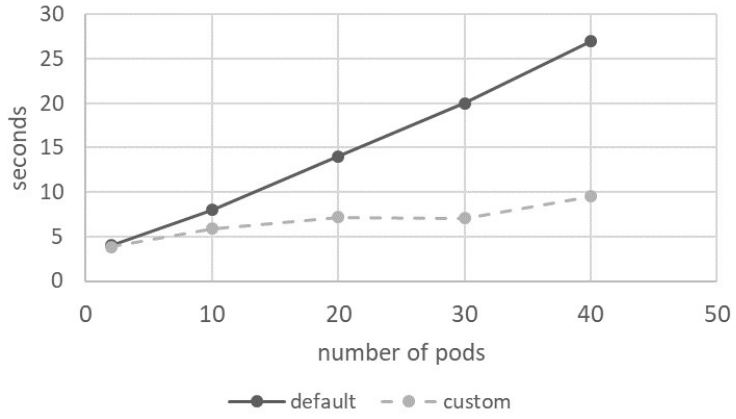


Figure 4.3: Scheduling time comparison

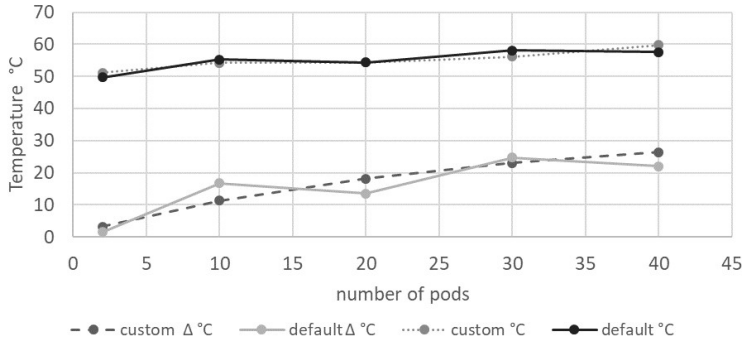


Figure 4.4: Node temperature comparison

This last metric is an observational indicator for cluster health, since it is a parameter indicating the usage of the nodes not considered in the allocation by our scheduler. Data were collected by three means: execution time measurement code, either embedded in the scheduling components (when possible) or coming from test packages; Cluster information displayed by the Kubernetes dashboard; Monitoring data retrieved from the devices (the same data that includes the parameters used by the optimization component).

The results from the custom scheduler were satisfyingly balanced. And this is obtained not only by focusing on the number of pods per node, but also taking into account the strain that the node is sustaining, in a way that is, most of the times 'stricter' than the approach taken by the default scheduler. Nodes overheating happened a few times with the default scheduler. Time efficiency of the default scheduler was proven in Figure 4.3, with

The average temperature and delta of both schedulers resulted similar, (Figure 4.4) showing that our scheduler does not compromise the life expectancy of the cluster. With custom scheduler being 64% faster in case of 40 pods deployments.

4.4.2 New Improved Setup

To further explore the IIoT use-case, the second cluster was built on top of a real 4G/5G cellular network. This end-to-end verification system is built up by using 5G NSA core, and an Edge network with Kubernetes cluster that operates across the wireless connectivity. The connectivity between core and edge is established via a high-performance 100G router. The core network provides 4G and 5G connectivity with the following main components, installed in a virtualized environment: Evolved packet gateway (EPG), 3GPP compliant Policy and Charging Rules Function (PCRF) [182] and Policy Control Function (PCF), Mobility Management Entity (MME), Home Subscriber Server (HSS), User Database compliant to 3GPP User Data Convergence standard [183]. The setup operates in NSA mode, which means the control plane uses 4G control functions, while the user plane is provided by 5G. The Edge network is distributed, the master node is instantiated in a VM close the core network, while the 4G, and 5G worker nodes are connected to the Kubernetes cluster via 4G and 5G connectivity, respectively. The worker nodes types are Raspberry Pi model 3B and 3B+, two of them can perform 4G attach to the test APN of the core via 4G USB modems (Huawei e3372) and the other two via 5G modem through USB tethering.

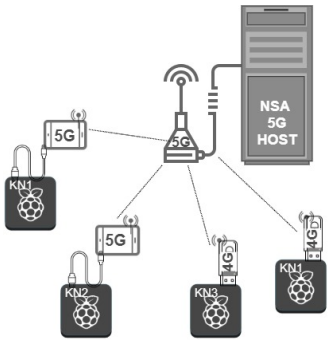


Figure 4.5: Distributed Edge setup

This configuration allows the nodes to have access to the internet (dashed line in Figure 4.6), for example to access a datacenter or to download container images needed for deployments.

Mobile connectivity, however, introduces in the cluster the issue of Network address translation (NAT) . The mobile nodes are not physically in the same subnet and cannot be pinged by the master on the server nor can they communicate with

each other using their internal addresses. This is against basic requirements of a Kubernetes network [123]:

1. any two containers should be able to communicate without NAT
2. any node should communicate with all containers without NAT
3. the IP that a container sees itself as must be the same IP others can use to reach it

As shown in Figure 4.6 the K8s cluster will have to rely on a Container Network Interface plugin. Pods are connected to the node network namespace with a virtual Ethernet pair: two namespaces with an interface on each end (veth0 in the root node namespace, and eth0 within the pod). Pods in the same node are connected to each other and to the node's eth0 interface via a bridge: docker0. The mapping of virtual IPs to pod IPs within the cluster is coordinated by the kube-proxy process on each node. This process sets up iptables. In our cluster setup we used Flannel [184] as the plugin to configure the layer 3 IPv4 network fabric for Kubernetes.

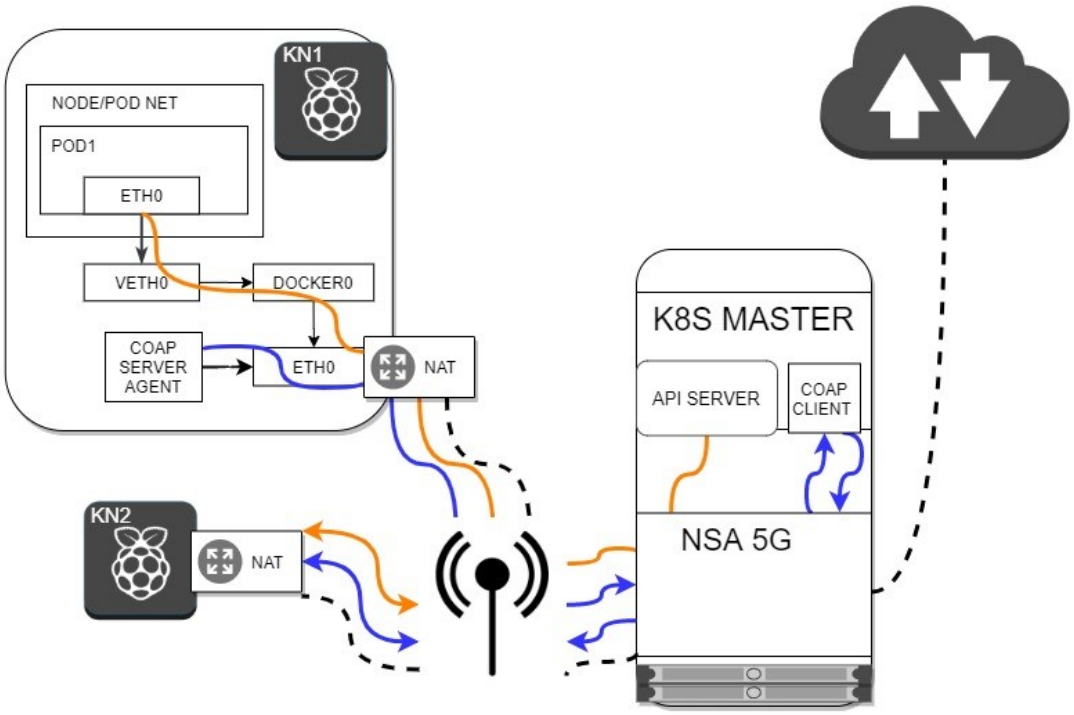


Figure 4.6: Network connections in our setup

Any deployment or service inside the cluster will not be influenced by NAT since it will follow this virtual network (orange line), however, our monitoring tool being outside K8s namespace, had to be adapted to overcome the NAT (blue line). Our approach consisted in having the Server side on the *Raspberry pis* to send an ACK as soon as the tool starts. Since only the minion nodes are behind the NAT, they can see and have access to the Master node. After the client side on the K8s node receives the ACK, his address will be resolved in the NAT tables of the modems attached to the Raspberry pi. From now on the VM hosting the master node will be able to act as a client as described in Section 4.3.1. An other approach would be to handle a virtual private network (VPN) to enable direct node to node communication at the Edge. There are two solutions in this direction:

1. To handle the VPN inside the cluster: a Pod in K8S will host an instance of a openVPN server and allow other nodes to become clients, by exposing itself as a cluster IP service. The tunneling is handled inside Kubernetes so, to be able to communicate with Edge devices directly, a UE should become part of the Edge cluster. In our scenario this is not preferable, because we may want for a device to be able to join our network without becoming part of the distributed Edge, but just as a client.

2. To handle the VPN outside of the cluster: an openVPN server instance may be running on the Edge master site, all devices that want to join must receive the client configurations from the server, but they do not need to join the K8s cluster. Nodes of the cluster will have to join using the virtual address of the VPN or they will have to communicate their alternative VPN address to the master. Let us imagine a mobile device moving near our distributed Edge, now it will be able to obtain a VPN address via an https get request to the server. The Edge master exposes the available services addresses to all VPN clients. From now on, the device will be able to use cluster services associated to nodes virtual IPs. This configuration reduces the traffic handled trough the Edge master and promotes segregation of duties, separating clients-only devices from cluster devices.

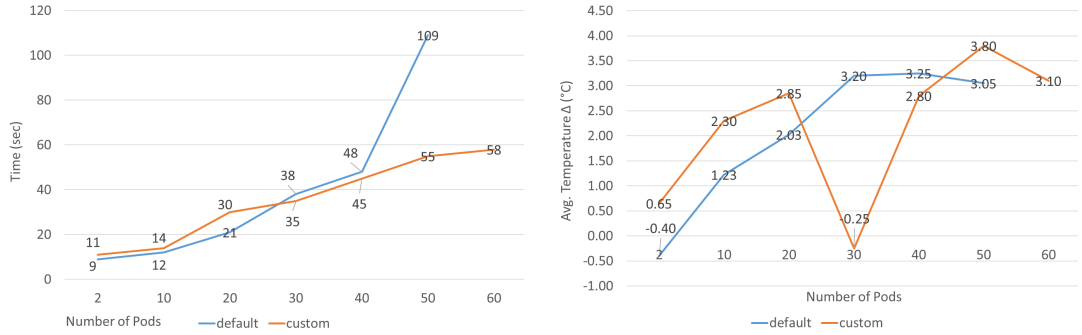
4.4.3 Final Test Results

Compared to the previous implementation [6], the current custom scheduler applies an algorithm that makes it more sensitive to environment and application changes. This enabled us to extend the limits of nodes capabilities, which is especially important in the case of Edge Computing.

In the new test setup the number of pods allocated per node is less balanced, as expected from the algorithm of our scheduler (Table 4.1). This results in a variation

Table 4.1: Pod allocation per node

	Allocated pods per node							
	Default scheduler				Custom scheduler			
Total # pods	kn1	kn2	kn3	kn4	kn1	kn2	kn3	kn4
2	1	0	1	0	0	1	1	0
10	2	3	2	3	2	3	3	2
20	5	5	5	5	3	8	6	3
30	7	8	8	7	5	8	10	7
40	10	10	10	10	14	10	8	8
50	13	12	13	12	11	16	14	9
60	-	-	-	-	13	20	15	12



(a) Time of scheduling deployments (b) Average cluster temperature deltas

Figure 4.7: Cluster health comparison among schedulers

of the overall cluster temperature that is also less linear: nodes get the chance to cool down and "rest". When actuating the allocation of a significant number of pods, the custom scheduler becomes faster, not only at selecting nodes but also for what matters: the overall completion time of the deployments. In our test cases this improvement in performance showed itself starting from the deployment of 30 pods, as visible in Figure 4.7a. This confirms that when the devices are under stress, our scheduler performs better by choosing less overwhelmed nodes. The result becomes even more interesting when considering the allocation of 50 pods onward. In this instance, the time spent by the default scheduler resulted being 109 seconds, against the 55 seconds taken by ours.

The main reason for this is the fact that, while the default scheduler prioritizes strongly the balance in the allocation of the pods (mostly by number), ours tries to preserve a safe operational state of the devices. The default scheduler then had issues optimizing the allocation when the Kubernetes core agent running in the worker nodes started reporting that the devices were in "risky states", but since

each of them would sporadically report something similar (because a very similar number of pods was being allocated), it found itself having to adjust its decision several times per pod, before actually binding them to a node. All this added to the fact that some nodes probably were not capable of reporting themselves correctly to the master node, again due to pressure, as defined in point 5 of Section 4.2.

As further proof for that, the custom scheduler managed to successfully allocate 60 pods on the Raspberry Pis, while the default one made the entire cluster crash (the devices overheated and stopped working), as visible in Table 4.1.

4.5 Related works

Only the sum of requested resources in each node of a cluster is taken into consideration by the default scheduler in Kubernetes. This is not effective enough when resource optimization should also account for potential sudden and drastic performance degradation. Despite that, there are not too many works focusing mainly on the improvement of the scheduling process in Kubernetes.

Specifying only limits for resource utilization is not enough to avert the risk of resource contention, as shown in [185]. The authors explored the problem and proposed as a result the software architecture for a scheduler which tries to avoid the issue by characterizing the "incoming" applications. Briefly, the scheduler makes an effort to put containers that are characterized by high resource usage in different host machines. The scheduling time also happened to get an improvement in speed that was around 20%, compared to the default scheduler, in a few test scenarios. Similarly to our approach, the authors tried to actualize a balanced distribution of tasks, but the overall speed improvement is not significantly strong and there was no dynamic input taken into consideration.

A close approach to ours in introducing better multi-objective scheduling in Kubernetes has been published in 2019 [186]. The authors formulated energy efficiency as a multi-objective optimization problem between maximal use of green energy, optimal performance with minimal interference, and overall energy minimization. The ILP problem is solved via Mosek Solver.

The orchestration concept based on Kubernetes has been extended to fog computing in [187]. Authors designed a set of labels for the default scheduler to addresses the application deployment challenges in fog set-ups: distribution, connectivity, availability, heterogeneity, and real-time.

Also [175] extends the basic K8s scheduler with labels: applications are classified depending on which resource they use more intensively – CPU, I/O disk, network bandwidth, or memory bandwidth. The score of a node at scheduling time is penalized if similar labels are already present on that node. This ensures that the resulting Kubernetes scheduler can balance the number of applications in

each node while still minimizing the degradation caused by resource competition. The overall execution time of the experiment is about eight minutes. This value is only 20% better than the mean time of the Kubernetes scheduler. The total time is similar to the best case of the default scheduler and overall the scheduling process has a lower time variance.

Similar efforts to reduce resource degradation have been expressed in [188]. Authors define a Reference Net (a Petri net) to model performance and management of resources for Kubernetes, identifying different operational states associated with "pods", containers and their shared resources. Such a model may be potentially used to calculate interference generated from certain deployments.

A network-aware scheduling approach for container-based applications in Smart City deployments is proposed in [189]. The Kubernetes scheduler is configured to make use of nodes RTT labels to decide where they are suitable to deploy a specific service with the target location specified on the pod configuration file. After completion of the scheduling request, the available bandwidth is updated on the corresponding node label. The objective of this scheduler is not to reduce strains on nodes nor to reduce the scheduling time, but to enhance the deployment performance by choosing nodes closer to the target of the service. In fact, the overall time of scheduling is slightly increased compared to the default scheduler.

The monitoring mechanism in [190] takes both system resource utilization and application QoS metrics into account. However it is not lightweight since it is composed of a container-based cluster-monitoring tool for Kubernetes, a database designed to store time series data and a visualization application. The authors also provide a dynamically configurable resource provisioning algorithm for K8s. As mentioned in [186] these tools occupy a great number of resources in the cluster.

As mentioned before in this Chapter, a notable example of a custom scheduling mechanism that can be installed as a plug-in into Kubernetes is the Poseidon-Firmament Scheduler [171], which tries to solve the scheduling problem by modeling a graph of the network flow, on which it runs its the optimization. It is therefore meant to solve the issue of optimal allocation only in regards to the network performance, which anyway proved to be good enough as an approach to make it 50%-80% faster than the Kubernetes default scheduler in the process of binding a pod to a node.

VM placement [191], such as cloud resource management [192], can be considered to be the "traditional ingredients" in the field of provisioning for resilient online services, which had a strong influence in the research related to scheduling and placement of containers. A very informative listing of several solutions for VM placement can be found in [136]. These solutions are categorized by the objective they want to fulfill, in terms of resource usage optimization, and the categorization underlines the fact that these solutions are specific for a single or very restrictive

set of parameters. Our study, on the other hand, was conducted with a different mindset, aiming to design a solution that would allow the infrastructure owner to directly control how many parameters are taken into consideration for the system optimization. In [180] a thorough analysis of weight computation formulas, related to multi-objective optimizations, is presented, especially for tasks like dynamic and automated strategies for decision making.

The paper by Parest [193] shows several other methodologies to accomplish similar estimations, nonetheless, both this and the aforementioned work do not present approaches that make use of online scoring and dynamic weight attribution.

4.6 Conclusions and Future Works

In this Chapter we presented work based on the scheduler published in [6] and [7] .

We realized a scheduler for distributed Edge Computing usecases, such as IIoT. The scheduler component is designed for a Kubernetes cluster and is dynamically capable of adjusting the way it allocates pods to nodes, based on a measure of the strain that is affecting the devices at run-time.

The current Kubernetes custom scheduler applies an algorithm that makes it more sensitive to environmental and application changes. This is not ideal in an Edge Computing scenario. We extend the limits of nodes capabilities, especially scheduling edge-native applications.

A fully functioning 5G and Edge computing network was built, complete with 5G radio and connectivity located in the proximity of the IoT devices. All the LTE virtualized functions are hosted at the same server rack as the Kubernetes cluster, as per the definition of Edge Computing.

Our solution is capable of adjusting jobs allocations over the nodes, balancing not only memory and CPU usage, but also multiple specific network and infrastructure parameters.

Points of improvement regard the speed and efficiency of the scheduling process in case of stressful deployments: starting from 30 pods onward, the application is ready to run earlier for the custom scheduler. At 50 pods the time of scheduling is cut in a half compared to the default scheduler. While the custom scheduler also manages to allocate up to 60 pods, the default one causes all the devices to overheat and shut down even before completing container creation. The proposed solution has trade-offs with operational and capital expenditures, those can be tackled by reducing or increasing the sample rate of the system. Further tests should be applied for this kind of fine tuning. In the next works we plan to reduce the sampling of the nodes and network status. Each parameter should have a separate sampling rate, adapted according to the application dynamic behavior

and previous footprint in the score calculation. The idea is to further shrink down the impact of the metric agents on the nodes and the network, intending to shorten the reaction time of the scheduler.

4.7 Contributions

In this Chapter we collected our work concerning Thesis 3.

I analyzed existing schedulers and found their limits for Edge computing application. I ideated a new scoring algorithm for a K8s scheduler to prevent choice of unstable resources. I created a lightweight tool to profile nodes and to feed the scheduler. I tested the scheduler and monitoring tool in a real 5G Edge setup. I have proven that it can schedule more pods of the same deployment while keeping the service agreement and without killing the nodes.

Figure 4.8 summarize the main concepts and contributions.

Thesis 3 (Distributed Mobile Edge Scheduling)**.** *In many Edge scenarios, especially those related to distributed Edge and IoT, it is for the best to reduce the need for application migration from the start.* ***I proved that we can reduce need for migration trough the improvement of the deployment strategies. I made a lightweight monitoring tool to feed real time telemetries to a scheduler component of the edge orchestrator. Thanks to awareness of cluster status resources are used more efficiently: 10 to 20 more ngnix containers can be scheduled on our cluster of raspberrypies compared to the default K8s scheduler. The devices do not crash due to overheat contrary to the default scheduling setup.***

I selected a scoring system that no only increased life of the raspberrypies but that makes the scheduler up to 64% faster than the default. The score is based on a composition (Multiplicative Exponent Weighting) of nodes telemetries, where the minimum value correspond to the best candidate. Weights are using Coefficient of Variation: the more a parameter is variable the higher its influence on the deployment.

thesis	relevant publications							
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
(1) Facilitate Deployment at the Edge	∘	∘	∘					
(2) Migration in Edge Computing			∘	∘	∘			
(3) **Distributed Mobile Edge Scheduling**			•			•	•	•

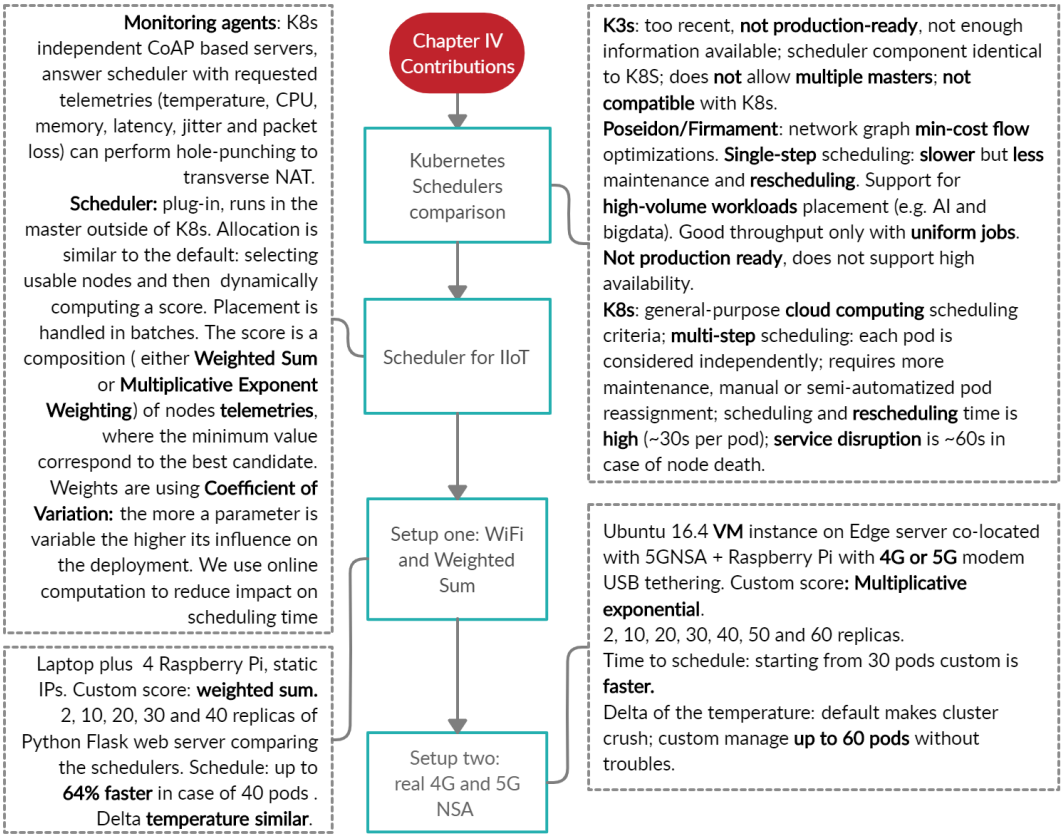


Figure 4.8: Summary of Chapter Contributions

Chapter 5

Dissertation Extended Summary

This dissertation presented novel research results in three fundamental areas of Edge Computing: applications deployment, applications migration, Network and Cloud integration. We identify characteristics and unique needs for Edge-native applications. We presented a tool-set to suggest partitioning of a cloud applications in components closer to our definition of edge native. Than we simulated and optimized first deployment based on historical data of network and user requests. We suggested a new approach to integrate live migration of containers in the Edge infrastructure. We demonstrated a new context aware scheduling for Distributed Edge computing.

We finally provided a collection of services to integrate network connectivity to the Edge and a proof of concept implementation for new techniques of integration of Cloud services on the Edge.

The three main thesis and results of this works are summarized as follows:

5.1 Results

Thesis 1 (Facilitate Deployment at the Edge)**.** *Adapting code from Cloud to Edge involves a great amount of work to refactor applications and services. Usually developers do this solely based on experience, simulating tools can be of support since the infrastructure is new and blind deployment may be expensive. Furthermore the extensive mobility of the applications represents a challenge for the choice of a correct separation of applications in micro-services.* ***I created a framework for partitioning and estimation of best deployment of a monolithic application. I implemented it in the form of a tool to assist into selecting the proper way to refactor an application, via graph partitioning schemes. I demonstrated this concept partitioning an AR application at a function level granularity.*** *Via non-intrusive online profiling of the application I created a call graph representing functions runtimes and number of calls between functions. The partition process applies min edge cut using a refinement algorithm, MLKL, that Coarsen, Partition and Uncorsen a graph multiple times. Any weighted graph may be used to represent the application in this step in our tool.*

This method aims at suggesting partitions that minimize their interaction, the idea is that the partitions should be easily moved around the network without compromising performances. Indeed more complex approaches may involve other parameters, such as the amount of allocated memory or the dependency to other code/services.

Finally the tool can suggest pareto-optimal places for deployment, based on Network resources capacities and existing links. I simulate the placement and routing issues into a single algorithm taking care to handle multiple user request at the same time. I show how it is possible to compute a deployment that satisfies SLA keeping in consideration cost of resources and benefit of requests. I proved that the approach does performs better than a random deployment, reducing deployment and networks failures.

Thesis 2 (Migration in Edge Computing)**.** *In Edge computing it will be necessary to move an edge-native application from one edge to the other, following the user, or, for recovery reasons, even from an Edge to the Cloud. The best approach to avoid this would be to have Stateless Apps, but there are cases in which this is not a possibility.*

I implemented a Java simulation of a Fog network, integrating live migration trough means of CRIU library calls and coordination via Docker Swarm. After testing the limits of this configuration I set up a better version based on Kubernetes. I constructed a docker in docker solution that is deployed as a daemon set on each node. This

component is in charge of actuating iterative pre-copy migration of its hosted containers, if triggered by any other software running in the cluster. I have tested the migration against stop and restart of multiple container deployments and shown how, given a stable network connection (like a 5G coverage) we can use migration to achieve a lower Service Downtime. .

Thesis 3 (Distributed Mobile Edge Scheduling)**.** *In many Edge scenarios, especially those related to distributed Edge and IoT, it is for the best to reduce the need for application migration from the start.* ***I proved that we can reduce need for migration trough the improvement of the deployment strategies. I made a lightweight monitoring tool to feed real time telemetries to a scheduler component of the edge orchestrator. Thanks to awareness of cluster status resources are used more efficiently: 10 to 20 more ngnix containers can be scheduled on our cluster of raspberrypies compared to the default K8s scheduler. The devices do not crash due to overheat contrary to the default scheduling setup.***

I selected a scoring system that no only increased life of the raspberrypies but that makes the scheduler up to 64% faster than the default. The score is based on a composition (Multiplicative Exponent Weighting) of nodes telemetries, where the minimum value correspond to the best candidate. Weights are using Coefficient of Variation: the more a parameter is variable the higher its influence on the deployment.

5.1.1 Facilitate Deployment at the Edge

Our analysis of application deployment consisted of two phases. In the first work we identify the requirements for an AR use case, select a partitioning granularity and other possible ones to evaluate. We then propose an architecture inspired by previous works, with a focus on a hybrid adaptive solution. We selected the tool chain for context and application analysis to integrate into the framework. Finally, we demonstrated our first experiments on the offloading using a simple face recognition use case.

In the second work, we described the methods and the algorithms we used to develop the prototype of our tool to partition and deploy an application in a 5G distributed network.

The tool executes three main steps. First selecting the application granularity at function level and construct a graph model. Then reduce it into Modules by solving the NP-hard graph partitioning problem it represents;in our case we used MLKL min edge cut algorithm. Finally, implement and apply a fractional relaxation of the Path Computation and Function Placement Problem.

We classify nodes into three categories: UE, Edge Cloud Servers, and Central Cloud Servers.

The capacity of an edge $c(e)$ represents the available bandwidth between the two network nodes; that on the nodes,$c(v)$ depends on the amount of available computational resources and the cost of accessing them. We suppose several UEs that request services from the application. Each of these services may be different on the *Service type* and the *Location* of the involved nodes.

Each Module is a part of the application that, combined, can solve a specific service request. A service request for user j is specified by a tuple $s_j = (G_j, d_j, b_j, U_j)$.

$G_j = (M_j, Y_j)$ is a directed (acyclic) graph called the place-and-route graph (pr-graph). There are a single source and a single sink, that corresponds to the node requesting the service. We denote the source and sink nodes in G_j by $ns_j \in M_j$ and $nt_j \in M_j$, respectively. The other vertices correspond to services or processing stages of a request. The edges of the pr-graph are directed and indicate precedence relations between pr-vertices.

The demand of a request s_j, d_j is computed from the cost of running a complete module and the benefit of satisfying s_j, b_j, comes from the SLA.

We map the User Equipment service request s_j as the realization of a path through the directed partition graph representing the application. In this case, the Module's demand can be calculated over the cost of each function composing the Modules in the specific service request. The routing cost from one Module to the other becomes than the overhead or transmission cost brought by the selected Module interaction scheme. Thus, the impact of the service request on the network can vary only based on the Modules' location. To specify the possible realization of a pr-graph in the physical network, we use a function $U_j : M_j \cup Y_j \to 2^V \cup 2^E$ where $U_j(m)$ is a set of "allowed" nodes in N that can perform module m, and $U_j(y)$ is a set of "allowed" edges of N that can implement the precedences and routing requirement that corresponds to y. We now define for each service request s_j the product network $pn(N, s_j)$. The node set of $pn(N, s_j)$, denoted by V_j, is defined as $V_j \triangleq \cup_{y \in Y_j}(U_j(y) \times y)$. We refer to the subset $U_j(y) \times y$ as the y-layer in the product graph. The edge set of $pn(N, sj)$, denoted E_j, consists of two types of edges $E_j = E_{j,1} \cup E_{j,2}$ defined as follows:

1. Routing edges connect vertices in the same layer, they represent the physical links in the network.

   $E_{j,1} = \{((u, y), (v, y)) \mid y \in Y_j, (u, v) \in U_j(y)\}$

2. Processing edges connect two copies of the same network vertex in different layers, representing the move from one Module to the consecutive one in the service chain specified in Y.

$E_{j,2} = \{((v, y), (v, y')) \mid y \neq y' \in Y_j$ edges with common endpoint m, and $v \in U_j(m)\}$

The final goal is to compute valid realizations $\tilde{P} = \{\tilde{p}_i\}_{i \in I'}$ for a subset of the requests $I' \subseteq I$ so that $\tilde{P}$ satisfies the capacity constraint of N and maximize the total benefit $\sum_{i \in I'} b_i$. We apply the fractional relaxation of PCFP-problem, a variation of Raghavan's randomized rounding algorithm for general packing problems. Simulations were run with various requests of service simultaneously.

Related publications of this thesis [1, 2, 3].

5.1.2 Migration in Edge Computing

The current trend for cloud and Edge computing is toward self-contained stateless services. To develop a containerized application the best practice would be using stateless containers. However, real-world applications do require stateful behaviour. It is not always a trivial task to decouple the application components into containers trying to make most containers stateless; there are scenarios in which statefulness cannot be bypassed. JIRA and Jenkins could be mentioned as examples of container based solutions used daily in production that could benefit from migration. In applications that also require low latency, the challenge is to preserve the state of the container hosting it, while following the user physically moving away from the hosting part of the network. In Edge computing it will be necessary to move an edge-native application from one edge to the other, following the user, or, for recovery reasons, even from an Edge to the Cloud.

We worked on two scenarios integrating container live migration and leveraging on the current container orchestration technologies.

In the Docker Swarm case we described a Federated Learning case scenario: a peer-to-peer collaborative computation network represents Iot Fog computing resources. The nodes in the network are light-weight uniform piece of software, whose main responsibilities are:

1. The mapping of available and mutable resources. Before the deployment phase, a peer sends a request to the neighbouring nodes. As a node receives this request it returns an acknowledgement to use to calculate latency.

2. The creation of a dynamic deployment plan. This step is based on Thesis 1 work. In a naif SLA implementation, hosts with higher capabilities and better communication links will be chosen with a higher chance.

3. The migration of tasks. At the deployment, the initiator node of the service creates a swarm and sends a request to the nodes that are the assigned

location of the individual tasks to join the swarm. At the first deployment of the service (no images exist) we move all the service-specific volumes and build the images and containers at the destination node. The source code of a task along with the data needed at the execution is passed by the nodes through TCP sockets. In all other occasion container checkpoint and restore is used.

In the second work, we presented a possible integration of Live Container Migration patterns into Kubernetes.

We created a simple cluster with a master node, from where Kubernetes performs orchestration tasks; and two workers. We implemented a sidecar container solution in a Kubernetes Pod that contains a Docker in Docker (DinD) container. For the sake of simplicity, we migrate a single natural number generator container, that counts from zero to infinity. Migration is performed using CRIU between the two DinD daemons.

The Kubernetes Pod groups together all the sub-tasks necessary for the migration process in form of containers: an *Initiator*, a *Checkpointer*, a *Migrator*, and the *DinD* itself. The solution is nested because the *NNG* container will be created and run by the Docker daemon inside *DinD* only after Pod A has started.

Advantage of our proposal is that containers created inside the DinD container are still exploiting Kubernetes capabilities. They are reachable through localhost from all the nodes in the cluster, thus the migration process is simplified. Kubernetes garbage collects Pods after they are terminated. CPU and memory resources will be inherited by containers inside the DinD.

A live migration feature allow us to rethink scheduling and resource deployment on a finer grain, without compromising user quality of experience.

Related publications of this thesis [4, 5].

5.1.3 Distributed Mobile Edge - Scheduling and Exposing Services

We realized a scheduler for distributed Edge Computing usecases, such as IIoT.

The scheduler ranks the nodes to select the best candidate. The score is computed as a composition of the nodes utilization state properties (e.g. CPU usage, memory usage, core temperature, etc.), where the minimum value correspond to the best candidate(s). Score computation is based on the Coefficient of Variation (CV) weighting method, combined with Multiplicative Exponent Weighting (MEW).

$$score = \prod_{i=1}^{|P|} \left(\frac{p_{min}}{p_i} \right)^{-w_i}$$

For each parameter related to the context of the scheduling, $P := set\ of\ parameters$, the stream of values received from the devices is a time series data. The weighted scores will be $W := set\ of\ weights\ of\ the\ parameters$ with $|W| = |P|$; so that each p_i parameter data from a node device, is normalized by the minimum value it historically reached for that node; and with w_i the weight assigned to it.

The parameters weight aggregation strategy is:

$$w_i = D_i = \frac{\sigma_i^2}{\mu_{i,n}}$$

A fully functioning 5G and Edge computing network was built, complete with 5G radio and connectivity located in the proximity of the IoT devices. All the LTE virtualized functions are hosted at the same server rack as the Kubernetes cluster, as per the definition of Edge Computing.

Our solution is capable of adjusting jobs allocations over the nodes, balancing not only memory and CPU usage, but also multiple specific network and infrastructure parameters.

At 50 pods the time of scheduling is cut in a half compared to the default scheduler. While the custom scheduler also manages to allocate up to 60 pods, the default one causes all the devices to overheat and shut down even before completing container creation.

The scheduler is part of our framework for integration of network and cloud services to edge computing. We tested the idea trough a proof of concept: an edge-native application on a real 3GPP mobile network, showing the power of Container as a Service combined with Network exposure APIs serving as a Mobile Edge Computing platform.

Cloud orchestrators are designed to handle a cluster in a single Zone, meaning that all nodes should be co-located. Works to collaborate among different Zones or Cloud operators involves strategies such as cluster federation. These solutions are still in development stage and are limited since the deployment is never distributed: a service replica can either have all pods fully assigned in a cluster or none of them are.

To leverage at the same time on Edge and Cloud, we plan to enable a more flexible distribution. We propose a solution that extends the base capabilities of a Kubernetes cluster. The CSP allows Enterprises to deploy a distributed K8s cluster, where nodes can be allocated at the Edge and on multiple partner Cloud providers.

The CSP provides an API to create an instance of K8S orchestrator on its domain, it could be on the Edge or on the Cloud. The API allows generations of nodes on different cloud provider VMs. The Kubernetes container network interface (CNI) has integrated a script to contact the CSP Virtual Private Network Certificate authority, in charge of generating and spreading certificates to connect the distributed nodes. Each VM in the cloud sites will have a tunnel towards the other nodes from the same cluster. The CSP can leverage on Network exposure Function APIs based on ETSI MANO and make them accessible as services to the K8s clusters. This configuration allows the part of the application that resides at the Edge to access network based services, without the need to involve a central cloud. The CSP is in charge of managing a pool of accounts from different Cloud resources, so that the enterprise has a single point of contact to instantiate its services, without having to access the cloud provider sites separately. From the Kubernetes API, the Enterprise can access to a Catalogue collecting partner cloud services through OSBA brokers and operators. In this way the user can create resources for persistent storage.

Related publications of this thesis [6, 7, 8].

Chapter 6

Summary

This dissertation presented novel research results in three fundamental areas of Edge Computing: applications deployment, applications migration, Network and Cloud integration.

Thesis 1 (Facilitate Deployment at the Edge)**.** *I created a framework for partitioning and estimation of best deployment of a monolithic application. I implemented it in the form of a tool that proposes how to refactor an application, via graph partitioning schemes. I demonstrated this concept partitioning an AR application call graph. I suggest pareto-optimal places for deployment, simulate the placement and routing of multiple user requests into a single algorithm. I proved that the approach does performs better than a random deployment, reducing deployment and networks failures.*

Thesis 2 (Migration in Edge Computing)**.** *I designed a framework for migration of containers in Fog. I implemented a Java simulation of a Fog network, integrating live migration via CRIU calls and Docker Swarm. I identified its limits. I constructed a Kubernetes solution based on a docker in docker container deployed as a daemon set on each node. This component is in charge of actuating iterative pre-copy migration of its hosted containers, if triggered by any other software running in the cluster. I have tested the migration against stop and restart of multiple container deployments and shown that we can achieve a lower Service Downtime.*

Thesis 3 (Distributed Mobile Edge Scheduling)**.** *I create a deployment strategy that reduces rescheduling and needs to migrate services from dying nodes. I selected a scheduling scoring system that makes the scheduler up to 64% faster than default. I made a lightweight monitoring tool to feed real time telemetries to a scheduler component of the edge orchestrator. 10 to 20 more ngnix containers can be scheduled on our cluster of raspberrypies compared to the default scheduler. The devices do not crash due to overheat.*

References

[1] Reale, Anna, M. Tóth, and Z. Horváth, "Towards context aware computations offloading in 5G," in *11th European Conference on Software Architecture, ECSA 2017, Companion Proceedings, Canterbury, United Kingdom, September 11-15, 2017*, 2017, pp. 89–92.

[2] Reale, Anna, P. Kiss, C. Ferrari, K. Benedek, S. László, and M. Tóth, "Application functions placement optimization in a mobile distributed cloud environment," *Studia Universitatis Babes-Bolyai Series Informatica*, vol. 63, pp. 37–52, 2018.

[3] P. Kiss, Reale, Anna, C. Ferrari, and Z. Istenes, "Deployment of IoT applications on 5G edge," in *2018 IEEE International Conference on Future IoT Technologies, Future IoT*, 2018, pp. 1–9.

[4] Reale, Anna, P. Kiss, M. Tóth, and Z. Horváth, "Designing a decentralized container based fog computing framework for task distribution and management," *INTERNATIONAL JOURNAL OF COMPUTERS AND COMMUNICATIONS*, vol. 13, pp. 1–7, 2019.

[5] Reale, Anna, P. Kiss, M. Tóth, and Z. Horváth, "Live migration and orchestration of containerized applications on fog computing and 5G," *Acta Cybernetica*, submitted.

[6] M. Chima Ogbuachi, C. Gore, Reale, Anna, P. Suskovics, and B. Kovács, "Context-aware k8s scheduler for real time distributed 5G edge computing applications," in *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, Sep. 2019, pp. 1–6.

[7] M. C. Ogbuachi, Reale, Anna, P. Suskovics, and B. Kovács, "Context-aware kubernetes scheduler for edge-native applications on 5G," *JOURNAL OF COMMUNICATIONS SOFTWARE AND SYSTEMS*, vol. 16, pp. 85–94, 2020.

[8] Reale, Anna, B. Kovacs, M. Tóth, and Z. Horváth, "Services for an edge-native application," in *13th Joint Conference on Mathematics and Informatics*, 2020.

[9] vividcomm, "Iot: Cloudlets," https://vividcomm.com/2018/12/28/IoT-cloudlets/, 2020, Online; accessed 07-July-2020.

[10] Q.-V. Pham, F. Fang, V. N. Ha, M. J. Piran, M. Le, L. B. Le, W.-J. Hwang, and Z. Ding, "A survey of multi-access edge computing in 5G and beyond: Fundamentals, technology integration, and state-of-the-art," *IEEE Access*, 2020.

[11] S. Kekki, W. Featherstone, Y. Fang, P. Kuure, A. Li, A. Ranjan, D. Purkayastha, F. Jiangping, D. Frydman, G. Verin *et al.*, "Mec in 5G networks," *ETSI white paper*, vol. 28, pp. 1–28, 2018.

[12] C. J. Ferrari, "5G Edge Computing Proof of Concept and Use Case Exploring," Master's thesis, Eötvös Loránd University, Hungary,Budapest, 2017.

[13] Xiao Yuan, "A Brief Analysis on the Implementation of the Kubernetes Scheduler," https://www.alibabacloud.com/blog/a-brief-analysis-on-the-implementation-of-the-kubernetes-scheduler_595083, 2019.

[14] Q.-V. Pham and W.-J. Hwang, "Fairness-aware spectral and energy efficiency in spectrum-sharing wireless networks," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 11, pp. 10 207–10 219, 2017.

[15] Q.-V. Pham, T. Leanh, N. H. Tran, B. J. Park, and C. S. Hong, "Decentralized computation offloading and resource allocation for mobile-edge computing: A matching game approach," *IEEE Access*, vol. 6, pp. 75 868–75 885, 2018.

[16] Y. Dong, Z. Chen, P. Fan, and K. B. Letaief, "Mobility-aware uplink interference model for 5G heterogeneous networks," *IEEE Transactions on Wireless Communications*, vol. 15, no. 3, pp. 2231–2244, 2015.

[17] L. Zhang, K. Wang, D. Xuan, and K. Yang, "Optimal task allocation in near-far computing enhanced c-ran for wireless big data processing," *IEEE Wireless Communications*, vol. 25, no. 1, pp. 50–55, 2018.

[18] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing – a key technology towards 5G," *ETSI White Paper*, vol. 11, 2015.

[19] K. Dolui and S. K. Datta, "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing," in *2017 Global Internet of Things Summit (GIOTS)*. IEEE, 2017, pp. 1–6.

[20] M. Satyanarayanan, "Cloudlets: At the leading edge of cloud-mobile convergence," in *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, 2013, pp. 1–2.

[21] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, no. 4, pp. 14–23, 2009.

[22] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Cloudlets: Bringing the cloud to the mobile user," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*, 2012, pp. 29–36.

[23] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[24] E. T. S. I. 2, *Network Functions Virtualisation (NFV) Release 3; Architecture;*, ETSI ETSI ETSI GS MEC-IEG 004 V1.1.1 (2015-11), 2015.

[25] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

[26] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis, "The locator/id separation protocol (lisp)," CISCO, Tech. Rep., 2013.

[27] P. Raad, S. Secci, D. C. Phung, A. Cianfrani, P. Gallard, and G. Pujolle, "Achieving sub-second downtimes in large-scale virtual machine migrations with lisp," *IEEE Transactions on Network and Service Management*, vol. 11, no. 2, pp. 133–143, 2014.

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

[29] "Cloud computing at the tactical edge," https://resources.sei.cmu.edu/asset_files/TechnicalNote/2012_004_001_28146.pdf, 2012, Online; accessed 07-July-2020.

[30] Cisco, "Flexpod express with cisco ucs mini and vmware vsphere 5.5," https://www.cisco.com/c/dam/en/us/td/docs/unified_computing/ucs/UCS_CVDs/flexpod_express_ucsmini_esxi55_fc.pdf, 2015, Online; accessed 07-July-2020.

[31] B. Mejias and P. Van Roy, "From mini-clouds to cloud computing," in *2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop*. IEEE, 2010, pp. 234–238.

[32] M. Dano, "Edge computing mini data centers are rolling out for real. what's next?" https://www.lightreading.com/the-edge/edge-computing-mini-data-centers-are-rolling-out-for-real-whats-next/d/d-id/755493, 2019, Online; accessed 15-July-2020.

[33] I. MobiledgeX, "Cloudlets and mobiledgex," https://mobiledgex.com/product/cloudlets.

[34] S. Bicheno, "Mobiledgex launches enterprise edge computing collective seamster," https://telecoms.com/503465/mobiledgex-launches-enterprise-edge-computing-collective-seamster/.

[35] S. Inc., "Cloud sync," https://www.synology.com/en-us/dsm/feature/cloud_sync, 2020, Online; accessed 07-July-2020.

[36] C. Boberg, M. Svensson, and B. Kovács, "Distributed cloud – a key enabler of automotive and industry 4.0 use cases," https://www.ericsson.com/en/reports-and-papers/ericsson-technology-review/articles/distributed-cloud.

[37] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani, and R. Buyya, "Cloud-based augmentation for mobile devices: motivation, taxonomies, and open challenges," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 1, pp. 337–368, 2014.

[38] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[39] I. Chih-Lin, J. Huang, R. Duan, C. Cui, J. X. Jiang, and L. Li, "Recent progress on c-ran centralization and cloudification," *IEEE Access*, vol. 2, pp. 1030–1039, 2014.

[40] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[41] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[42] L. Lu, G. Y. Li, A. L. Swindlehurst, A. Ashikhmin, and R. Zhang, "An overview of massive mimo: Benefits and challenges," *IEEE journal of selected topics in signal processing*, vol. 8, no. 5, pp. 742–758, 2014.

[43] E. Commission, "5G for europe: An action plan," Communication From the Commission to the European Parliament, the Council, the European Economic and Social Committee and the Committee of the Regions.

[44] B. Bangerter, S. Talwar, R. Arefi, and K. Stewart, "Networks and devices for the 5G era," *IEEE Communications Magazine*, vol. 52, no. 2, pp. 90–96, 2014.

[45] P. Suskovics, B. Kovács, S. Terrill, and P. Wörndle, "Creating the next-generation edge-cloud ecosystem," https://www.ericsson.com/en/reports-and-papers/ericsson-technology-review/articles/next-generation-cloud-edge-ecosystems.

[46] A. Huang, N. Nikaein, T. Stenbock, A. Ksentini, and C. Bonnet, "Low latency mec framework for sdn-based lte/lte-a networks," in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017, pp. 1–6.

[47] 3GPP, "3gpp release 15 – the first full set of 5G standards," International Organization for Standardization, Standard 3GPP TR 21.915, 2018. [Online]. Available: https://www.3gpp.org/release-15

[48] A. Neal, B. Naughton, C. Chan, N. Sprecher, and S. Abeta, "Mobile edge computing (mec); technical requirements," *ETSI, Sophia Antipolis, France, White Paper no. DGS/MEC-002*, 2016.

[49] E. Foundation, "Edge native working group," https://eclipse-foundation.blog/2019/12/10/close-to-the-edge/.

[50] M. Satyanarayanan, G. Klas, M. Silva, and S. Mangiante, "The seminal role of edge-native applications," in *2019 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 2019, pp. 33–40.

[51] J. Wang, Z. Feng, S. George, R. Iyengar, P. Pillai, and M. Satyanarayanan, "Towards scalable edge-native applications," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 152–165.

[52] E. AB, "5G systems, enabling the transformation of industry and society," ERICSSON, White Paper, 2017. [Online]. Available: https://www.ericsson.com/assets/local/publications/white-papers/wp-{5G}-systems.pdf

[53] G. Even, M. Rost, and S. Schmid, "An approximation algorithm for path computation and function placement in sdns," in *International Colloquium on Structural Information and Communication Complexity*. Springer, 2016, pp. 374–390.

[54] Y. Yu, "Mobile edge computing towards 5G: Vision, recent progress, and open challenges," *China Communications*, vol. 13, no. Supplement2, pp. 89–99, 2016.

[55] J. Liu, E. Ahmed, M. Shiraz, A. Gani, R. Buyya, and A. Qureshi, "Application partitioning algorithms in mobile cloud computing: Taxonomy, review and future directions," *Journal of Network and Computer Applications*, vol. 48, pp. 99–117, 2015.

[56] S. Deshmukh and R. Shah, "Computation offloading frameworks in mobile cloud computing: a survey," in *Current Trends in Advanced Computing (ICCTAC), IEEE International Conference on*. IEEE, 2016, pp. 1–5.

[57] D. Delaney, T. Ward, and S. McLoone, "On consistency and network latency in distributed interactive applications: A survey—part i," *Presence: Teleoperators and Virtual Environments*, vol. 15, no. 2, pp. 218–234, 2006.

[58] J. Cho, K. Sundaresan, R. Mahindra, J. Van der Merwe, and S. Rangarajan, "Acacia: Context-aware edge computing for continuous interactive applications over mobile networks," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 2016, pp. 375–389.

[59] S. Nunna, A. Kousaridas, M. Ibrahim, M. Dillinger, C. Thuemmler, H. Feussner, and A. Schneider, "Enabling real-time context-aware collaboration through 5G and mobile edge computing," in *Information Technology-New Generations (ITNG), 2015 12th International Conference on*. IEEE, 2015, pp. 601–605.

[60] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.

[61] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[62] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu, "Towards a distributed platform for resource-constrained devices," in *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*. IEEE, 2002, pp. 43–51.

[63] L. Wang and M. Franz, "Automatic partitioning of object-oriented programs for resource-constrained mobile devices with multiple distribution objectives," in *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*. IEEE, 2008, pp. 369–376.

[64] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[65] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: Profile-based partitioning for sensornet applications." in *NSDI*, vol. 9, 2009, pp. 395–408.

[66] L. Yang, J. Cao, and H. Cheng, "Resource constrained multi-user computation partitioning for interactive mobile cloud applications," *Technical reort. Department of Computing, Hong Kong Polytechnical University*, 2012.

[67] T. Verbelen, T. Stevens, F. De Turck, and B. Dhoedt, "Graph partitioning algorithms for optimizing software deployment in mobile cloud computing," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 451–459, 2013.

[68] I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso, "Calling the cloud: enabling mobile phones as interfaces to cloud applications," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., 2009, p. 5.

[69] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive offloading inference for delivering applications in pervasive computing environments," in *Pervasive Computing and Communications, 2003.(PerCom 2003). Proceedings of the First IEEE International Conference on*. IEEE, 2003, pp. 107–114.

[70] D. Kovachev, "Framework for computation offloading in mobile cloud computing," *IJIMAI*, vol. 1, no. 7, pp. 6–15, 2012.

[71] M.-R. Ra, B. Priyantha, A. Kansal, and J. Liu, "Improving energy efficiency of personal sensing applications with heterogeneous multi-processors," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM, 2012, pp. 1–10.

[72] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao, "Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems," in *Proceedings of the 45th annual design automation conference*. ACM, 2008, pp. 191–196.

[73] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[74] H. Meyerhenke, B. Monien, and S. Schamberger, "Graph partitioning and disturbed diffusion," *Parallel Computing*, vol. 35, no. 10-11, pp. 544–569, 2009.

[75] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[76] F. Berman, "High-performance schedulers," *The grid: blueprint for a new computing infrastructure*, vol. 67, pp. 279–309, 1999.

[77] D. L. Long and L. A. Clarke, "Task interaction graphs for concurrency analysis," in *Proceedings of the 11th international conference on Software engineering*. ACM, 1989, pp. 44–52.

[78] M. Naghibzadeh, "Modeling workflow of tasks and task interaction graphs to schedule on the cloud," *CLOUD COMPUTING 2016*, p. 81, 2016.

[79] D. Grove and C. Chambers, *An assessment of call graph construction algorithms*. IBM Thomas J. Watson Research Division, 2000.

[80] H. Gani, "Transparent and adaptive application partitioning using mobile objects," 2010.

[81] A. Computing *et al.*, "An architectural blueprint for autonomic computing," *"IBM White Paper"*, vol. 31, pp. 1–6, 2006.

[82] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Unconventional Programming Paradigms*. Springer, 2005, pp. 257–269.

[83] M. B. Alaya and T. Monteil, "Frameself: A generic context-aware autonomic framework for self-management of distributed systems," in *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2012 IEEE 21st International Workshop on*, IEEE. IEEE, 2012, pp. 60–65.

[84] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Infocom, 2012 Proceedings IEEE*. IEEE, 2012, pp. 945–953.

[85] C. M. S. Magurawalage, K. Yang, L. Hu, and J. Zhang, "Energy-efficient and network-aware offloading algorithm for mobile cloud computing," *Computer Networks*, vol. 74, pp. 22–33, 2014.

[86] V. Developers, "Callgrind: A call-graph generating cache and branch prediction profiler," 2010.

[87] T. Imielinski and H. F. Korth, *Mobile computing*. Springer Science & Business Media, 1996, vol. 353.

[88] Haojie Fan and Yongmin Mu, "A performance testing and optimization tool for system developed by python language," in *International Conference on Cyberspace Technology (CCT 2013)*, Nov 2013, pp. 24–27.

[89] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," in *Mobile Computing*. Springer, 1994, pp. 449–471.

[90] G. Bradski, "The opencv library," *Dr Dobb's J. Software Tools*, vol. 25, pp. 120–125, 2000.

[91] M. Lutz, *Programming Python*. O'Reilly Media, Inc., 2006.

[92] D. Abrahams and R. W. Grosse-Kunstleve, "Building hybrid systems with boost.python," *The C Users Journal archive*, vol. 21, 2003.

[93] B. Hendrickson and R. W. Leland, "A multi-level algorithm for partitioning graphs." *SC*, vol. 95, no. 28, pp. 1–14, 1995.

[94] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations research*, vol. 21, no. 2, pp. 498–516, 1973.

[95] G. Karypis and V. Kumar, "Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0," University of Minnesota Twin Cities, Tech. Rep., 1995.

[96] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, "Just-in-time provisioning for cyber foraging," in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 2013, pp. 153–166.

[97] P. Raghavan, *Randomized rounding and discrete ham-sandwich theorems: provably good algorithms for routing and packing problems.* University of California. Computer Science Division, 1986.

[98] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[99] W. Scott, "Content delivery networks (cdn): Caching principles, architecture, and resource optimization," https://www.slideshare.net/hacktivism/cisco-live-content-delivery-networks-cdn, 2017, Online; accessed 29-March-2018.

[100] S. Xinyi, "Resource allocation in edgecomputing infrastructure," Master's thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, 6 2020.

[101] C. Sonmez, A. Ozgovde, and C. Ersoy, "Edgecloudsim: An environment for performance evaluation of edge computing systems," *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 11, p. e3493, 2018, e3493 ett.3493. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3493

[102] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[103] C. Sonmez, A. Ozgovde, and C. Ersoy, "Edgecloudsim: An environment for performance evaluation of edge computing systems," *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 11, p. e3493, 2018.

[104] J. Niu, W. Song, L. Shu, and M. Atiquzzaman, "Bandwidth-adaptive application partitioning for execution time and energy optimization," in *Communications (ICC), 2013 IEEE International Conference on.* IEEE, 2013, pp. 3660–3665.

[105] L. D. Pedrosa, N. Kothari, R. Govindan, J. Vaughan, and T. Millstein, "The case for complexity prediction in automatic partitioning of cloud-enabled mobile applications," *Small*, vol. 20, p. 25, 2012.

[106] M. Masdari, S. S. Nabavi, and V. Ahmadi, "An overview of virtual machine placement schemes in cloud computing," *Journal of Network and Computer Applications*, vol. 66, pp. 106–127, 2016.

[107] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.

[108] R. A. da Silva and N. L. da Fonseca, "Algorithm for the placement of groups of virtual machines in data centers," in *Communications (ICC), 2015 IEEE International Conference on*. IEEE, 2015, pp. 6080–6085.

[109] J. Chase, R. Kaewpuang, W. Yonggang, and D. Niyato, "Joint virtual machine and bandwidth allocation in software defined network (sdn) and cloud computing environments," in *Communications (ICC), 2014 IEEE International Conference on*. IEEE, 2014, pp. 2969–2974.

[110] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services," in *NSDI*. USENIX, April 2010. [Online]. Available: https://www.microsoft.com/en-us/research/publication/volley-automated-data-placement-for-geo-distributed-cloud-services/

[111] B. Malet and P. Pietzuch, "Resource allocation across multiple cloud data centres," in *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, 2010, pp. 1–6.

[112] M. Alicherry and T. Lakshman, "Network aware resource allocation in distributed clouds," in *2012 Proceedings IEEE INFOCOM*. IEEE, 2012, pp. 963–971.

[113] M. Steiner, B. G. Gaglianello, V. Gurbani, V. Hilt, W. D. Roome, M. Scharf, and T. Voith, "Network-aware service placement in a distributed cloud environment," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, 2012, pp. 73–74.

[114] J. Halpern and C. Pignataro, "Service function chaining (sfc) architecture," ietf, Tech. Rep., 2015.

[115] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*. IEEE, 2015, pp. 171–177.

[116] Y. Xie, Z. Liu, S. Wang, and Y. Wang, "Service function chaining resource allocation: A survey," *arXiv preprint arXiv:1608.00095*, 2016.

[117] T. Taleb, M. Bagaa, and A. Ksentini, "User mobility-aware virtual network function placement for virtual 5G network infrastructure," in *Communications (ICC), 2015 IEEE International Conference on.* IEEE, 2015, pp. 3879–3884.

[118] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz, "Near optimal placement of virtual network functions," in *Computer Communications (INFOCOM), 2015 IEEE Conference on.* IEEE, 2015, pp. 1346–1354.

[119] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "On orchestrating virtual network functions," in *Network and Service Management (CNSM), 2015 11th International Conference on.* IEEE, 2015, pp. 50–56.

[120] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. Ramakrishnan, and T. Wood, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," in *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on.* IEEE, 2015, pp. 1–6.

[121] M. Bouet, J. Leguay, and V. Conan, "Cost-based placement of virtualized deep packet inspection functions in sdn," in *Military Communications Conference, MILCOM 2013-2013 IEEE.* IEEE, 2013, pp. 992–997.

[122] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy, "Design and evaluation of algorithms for mapping and scheduling of virtual network functions," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on.* IEEE, 2015, pp. 1–9.

[123] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale.* 1005 Gravenstein Highway North Sebastopol, CA 95472: O'Reilly and Associates, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[124] P. Bellavista and A. Zanni, "Feasibility of fog computing deployment based on docker containerization over raspberrypi," in *Proceedings of the 18th international conference on distributed computing and networking.* ACM, 2017, p. 16.

[125] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. S. Goren, and C. Mahmoudi, "Fog computing conceptual model," 2018.

[126] E. T. S. I. 2, *Multiaccess Edge Computing (MEC); Technical Requirements*, ETSI ETSI ETSI GS NFV-IFA 011 V3.3.1 (2019-09), 2019.

[127] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Live service migration in mobile edge clouds," *IEEE Wireless Communications*, vol. 25, no. 1, pp. 140–147, 2018.

[128] B. Ibryam, "White paper: Principles of container-based application design," Red Hat Enterprise Linux, USA,NORTH AMERICA 1 888 REDHAT1, Tech. Rep., December 2018. [Online]. Available: https://www.redhat.com/cms/managed-files/cl-cloud-native-container-design-whitepaper-f8808kc-201710-v3-en.pdf

[129] S. Wang, J. Xu, N. Zhang, and Y. Liu, "A survey on service migration in mobile edge computing," *IEEE Access*, vol. 6, pp. 23 511–23 528, 2018.

[130] A. Mirkin, A. Kuznetsov, and K. Kolyshkin, "Containers checkpointing and live migration," in *Proceedings of the Linux Symposium*, vol. 2, 2008, pp. 85–90.

[131] C. Yu and F. Huan, "Live migration of docker containers through logging and replay," in *2015 3rd International Conference on Mechatronics and Industrial Informatics (ICMII 2015)*. Atlantis Press, 2015.

[132] Y. Chen, "Checkpoint and restore of micro-service in docker containers," in *2015 3rd International Conference on Mechatronics and Industrial Informatics (ICMII 2015)*. Atlantis Press, 2015.

[133] C. Feb, "Checkpoint/restore in userspace," 2019. [Online]. Available: https://criu.org/Main_Page

[134] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse, and W. Joosen, "A comprehensive feature comparison study of open-source container orchestration frameworks," *Applied Sciences*, vol. 9, no. 5, 2019. [Online]. Available: http://www.mdpi.com/2076-3417/9/5/931

[135] Mesosphere, "Mesos architecture," 2019. [Online]. Available: http://mesos.apache.org/documentation/latest/architecture//

[136] A. Laghrissi and T. Taleb, "A survey on the placement of virtual resources and virtual network functions," *IEEE Communications Surveys & Tutorials*, 2018.

[137] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[138] E. Zadok, *Linux NFS and Automounter Administration (Craig Hunt Linux Library)*. Sybex, 2001.

[139] L. Feb, “Linux container,” 2019. [Online]. Available: https://linuxcontainers.org

[140] OCI, “runc,” 2019. [Online]. Available: https://github.com/opencontainers/runc

[141] Docker, “Docker: Enterprise Container Platform.” [Online]. Available: "https://docker.com/"

[142] OpenVZ, “Openvz,” 2019. [Online]. Available: https://wiki.openvz.org/Main_Page

[143] L. Feb, “Linux container,” https://linuxcontainers.org, 2019.

[144] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[145] Virtuozzo, “Find your path to the cloud with virtuozzo infrastructure platform,” 2019. [Online]. Available: https://www.virtuozzo.com/

[146] Jelastic, “Jelastic,” 2019. [Online]. Available: https://jelastic.com/

[147] P. S. V. Indukuri, “Performance comparison of linux containers (lxc) and openvz during live migration,” Ph.D. dissertation, Master’s thesis, Blekinge Institute of Technology, Sweden, 2016.

[148] L. Ma, S. Yi, and Q. Li, “Efficient service handoff across edge servers via docker container migration,” in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM, 2017, p. 11.

[149] R. A. Addad, D. L. C. Dutra, M. Bagaa, T. Taleb, and H. Flinck, “Towards a fast service migration in 5G,” in *Proceedings of the IEEE NetSoft Conference, Montreal, QC, Canada*, 2018, pp. 25–29.

[150] L. Feb, “Linux container security,” 2019. [Online]. Available: https://linuxcontainers.org/lxc/security/

[151] Docker, “Docker:how secure they are,” 2019. [Online]. Available: https://blog.docker.com/2013/08/containers\protect\discretionary{\char\hyphenchar\font}{}{}docker\protect\discretionary{

[152] Docker, “Use volumes.” [Online]. Available: "https://docs.docker.com/storage/volumes/"

[153] rsync, “rsync features.” [Online]. Available: "https://rsync.samba.org/features.html"

[154] CRIU, “Incremental dumps.” [Online]. Available: "https://criu.org/Incremental_dumps"

[155] CRIU, “Disk-less migration.” [Online]. Available: "https://criu.org/Disk-less_migration"

[156] Docker, “Use tmpfs mounts.” [Online]. Available: "https://docs.docker.com/storage/tmpfs/"

[157] W. Scott, “Open container initiative-based implementation of kubernetes container runtime interface,” https://github.com/cri-o/cri-o/wiki, 2016, Online; accessed 13 July 2020.

[158] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, “Container migration in the fog: A performance evaluation,” *Sensors*, vol. 19, no. 7, p. 1488, 2019.

[159] P. A. Frangoudis and A. Ksentini, “Service migration versus service replication in multi-access edge computing,” in *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*. IEEE, 2018, pp. 124–129.

[160] J. Konecný, H. B. McMahan, D. Ramage, and P. Richtárik, “Federated optimization: Distributed machine learning for on-device intelligence,” *CoRR*, vol. abs/1610.02527, 2016.

[161] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” in *AISTATS*, 2017.

[162] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, “Autonomic vertical elasticity of docker containers with elasticdocker,” in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. IEEE, 2017, pp. 472–479.

[163] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, “Incremental deployment and migration of geo-distributed situation awareness applications in the fog,” in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM, 2016, pp. 258–269.

[164] F. O. Source, “A persistent keyvalue store,” 2019. [Online]. Available: https://rocksdb.org

[165] T. Taleb, S. Dutta, A. Ksentini, M. Iqbal, and H. Flinck, "Mobile edge computing potential in making cities smarter," *IEEE Communications Magazine*, vol. 55, no. 3, 2017.

[166] CRIU, "Criu: what cannot be checkpointed," 2019. [Online]. Available: https://criu.org/What_cannot_be_checkpointed/

[167] J. Petazzoni, "Using docker-in-docker for your ci or testing environment? think twice," 2015. [Online]. Available: https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci

[168] R. Emilsson, "Container performance benchmark between docker, lxd, podman & buildah," 2020.

[169] A. Kanso, M. Toeroe, and F. Khendek, "Comparing redundancy models for high availability middleware," *Computing*, vol. 96, no. 10, pp. 975–993, 2014.

[170] "Rancher", "K3s: Lightweight Kubernetes," https://k3s.io/.

[171] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand, "Firmament: Fast, centralized cluster scheduling at scale," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 99–115. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog

[172] Firmament, "Poseidon scheduler," https://github.com/kubernetes-sigs/poseidon.

[173] K. Authors, "Poseidon-firmament - an alternate scheduler," https://v1-16.docs.kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler, 2020, Online; accessed 29-July-2020.

[174] A. I. Bucur and D. H. Epema, "Local versus global schedulers with processor co-allocation in multicluster systems," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2002, pp. 184–204.

[175] V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. Á. Bañares, and O. F. Rana, "Client-side scheduling based on application characterization on kubernetes," in *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer, 2017, pp. 162–176.

[176] V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui, "Modelling performance & resource management in kubernetes," in *Proceedings of the 9th*

International Conference on Utility and Cloud Computing, 2016, pp. 257–262.

[177] Google, "CoAP — Constrained Application Protocol | Overview," https://coap.technology/.

[178] J. Dizdarević, F. Carpio, A. Jukan, and X. Masip-Bruin, "A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, p. 116, 2019.

[179] E. Horowitz and S. Sahni, "Computing partitions with applications to the knapsack problem," *J. ACM*, vol. 21, no. 2, pp. 277–292, Apr. 1974. [Online]. Available: http://doi.acm.org/10.1145/321812.321823

[180] M. Seufert, S. Lange, and M. Meixner, "Automated decision making based on pareto frontiers in the context of service placement in networks," in *2017 29th International Teletraffic Congress (ITC 29)*, vol. 1, Sep. 2017, pp. 143–151.

[181] M. Brettel, N. Friederichsen, M. Keller, and N. Rosenberg, "How virtualization, decentralization and network building change the manufacturing landscape: An industry 4.0 perspective," *International Journal of Science, Engineering and Technology*, vol. 8, pp. 37–44, 08 2014.

[182] 3GPP, *Policy and Charging Rules Function*, ser. TS. 3GPP, Sep. 2009, no. TS23.203, Rel-9 v0.4.0. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23203.htm

[183] 3GPP, *User Data Convergence (UDC); Technical realization and information flows*, ser. TS. 3GPP, Sep. 2009, no. TS23.335, Rel-9 v0.4.0. [Online]. Available: http://www.3gpp.org/ftp/Specs/html-info/23335.htm

[184] Flannel, "Flannel," https://github.com/coreos/flannel#flannel.

[185] V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. Á. Bañares, and O. F. Rana, "Client-side scheduling based on application characterization on kubernetes," in *Economics of Grids, Clouds, Systems, and Services*, C. Pham, J. Altmann, and J. Á. Bañares, Eds. Cham: Springer International Publishing, 2017, pp. 162–176.

[186] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "Keids: Kubernetes based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem," *IEEE Internet of Things Journal*, pp. 1–1, 2019.

[187] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, April 2018, pp. 1–7.

[188] V. Medel, O. Rana, J. A. Banares, and U. Arronategui, "Adaptive application scheduling under interference in kubernetes," in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, Dec 2016, pp. 426–427.

[189] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 351–359.

[190] C. Chang, S. Yang, E. Yeh, P. Lin, and J. Jeng, "A kubernetes-based monitoring platform for dynamic cloud resource provisioning," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, Dec 2017, pp. 1–6.

[191] K. Péter, R. Anna, T. Melinda, and H. Zoltán, "Designing a decentralized container based fogcomputing framework for task distribution and management," *International Journal of Computers and Communications*, vol. 13, pp. 1–7, 2019.

[192] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *Journal of Network and Systems Management*, vol. 23, no. 3, pp. 567–619, 2015.

[193] B. Kovács, "Parameter estimation of dynamical systems," *PhD Thesis dissertation*, 2011.

Acronyms

3GPP Third Generation Partnership Project.

5G 5th Generation (of Mobile Networks).

ABBE Attribute-Based Broadcast Encryption.

AI Artificial Intelligence.

AMF Access Management Function.

APA Application Partitioning Algorithms.

API Application Programming Interface.

AR Augmented Reality.

AUSF Authentication Server Function.

C-RAN Centralised or Cloud RAN.

CaaS Container or Cluster as a Service.

CC Cloud Computing.

CFG Control Flow Graph.

CG Control Graph.

CIA Continuous Interactive Applications.

CLI Command Line Interface.

CoAP (Constrained Application Protocol.

CPU Central Processing Unit.

CRD Custom Resource Definition.

CRI Container Runtime Interface.

CRIU Checkpoint Restore In Userspace.

CRUD Create Read Update Delete.

CSP Cloud Service Provider.

CV Coefficient of Variation.

DB Data Base.

DC Data Centers.

DC/OS Distributed Cloud Operating System.

DCN Data Centers Networks.

DDS Data Distribution Service.

DHCP Dynamic Host Configuration Protocol.

DinD Docker in Docker.

DNS Domain Name System.

DooD Docker outside of Docker.

EC Edge Computing.

EPG Evolved Packet Gateway.

EPS Edge Packet Service.

ETSI European Telecommunications Standards Institute.

GBA Generic Bootstrapping Architecture.

GPU Graphics Processing Unit.

HSS Home Subscriber Server.

HTTP HyperText Transfer Protocol.

IaaS Infrastructure as a Service.

IBBE Identity-Based Broadcast Encryption.

IETF Internet Engineering Task Force.

IIoT Industrial Internet of Things.

ILP nteger Linear Programming.

IoT Internet of Things.

IP Internet Protocol.

K8S Kubernetes.

KPI Key Performance Indicator.

KVM linux Kernel-based VM.

LAN Local Area Network.

LM Live Migration.

LP Linear Programming.

LTE Long Term Evolution.

LXC LinuX Container.

MBS Message Broadcast Service.

MEC Mobile or Multiaccess Edge Computing.

MEW Multiplicative Exponent Weighting.

MIMO Multiple-Input and Multiple-Output.

MLKL Multi-Level Kerninghan-Lin.

MME Mobility Management Entity.

MMORPG Massive Multiplayer Online Role-Playing Games.

MPTCP Multi-Path TCP.

MQTT Message Queue Telemetry Transport.

NAS Network Attached Storage.

NAT Network Address Translation.

NFS Network File System.

NFV Network Function Virtualization.

NNG NAtural Number Generator.

NR New Radio.

NSA Non Stand-Alone.

NSSF Network Slice Selection Function.

OCI Open Containers Initiative.

ORG Object Relation Graph.

OS Operative System.

OSBA Open Service Broker API.

P2P Peer-to-Peer.

PaaS Platform as a Service.

PCF Policy Control Function.

PCFP ath Computation and Function Placement.

PCRF Policy and Charging Rules Function.

PDU Protocol Data Unit.

PoC Proof of Concept.

QM Quine–McCluskey.

QoE Quality of Experience.

QoS Quality of Service.

RAM Random Access Memory.

RAN Radio Access Network.

RBAC Role-Based Access Control.

RC Replication Controller.

REST REpresentational State Transfer.

RNIS Radio Network Information Service.

ROI Return Of Investment.

RTT Round Trip Time.

SDK Software Development Kit.

SDN Software Defined Networking.

SFC Service Function Chaining.

SLA Service Level Agreement.

SMF Session Management Function.

TCP Transport Control Protocol.

TG Target Graph.

UC Use Case.

UDP User Datagram Protocol.

UE User Equipment.

UPF User Plane Function.

URL Uniform Resource Locator.

V2I Vehicle to Infrastructure.

V2V Vehicle to Vehicle.

vCMTS virtual Cable Modem Termination Systems.

VHEM Variational Hierarchical Expectation–Maximization.

VM Virtual Machine.

VNF Virtual Network Function.

VNFP VNF Placement.

vOLT virtual Optical Line Terminal.

VPN Virtual Private Network.

VR Virtual Reality.

vRAN virtual RAN.

WAN Wide Area Network.

XaaS Multi-cloud as a service.