Caravela: Cloud @ Edge

A fully distributed and decentralized orchestrator for Docker containers

André Filipe Pardal Pires

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Luís Manuel Antunes Veiga
Prof. José Manuel de Campos Lages Garcia Simão

Examination Committee

November 2018

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. I would also like to thanks my girlfriend, Tânia Corrales, for her understanding and support throughout all these years, specially when things did not went well.

I would also like to acknowledge my dissertation supervisor Prof. Luís Veiga for the insight, support and sharing of knowledge that has made this thesis possible. I would like to thanks for that moments where Prof. Luís Veiga tried to push the work into another level, that thrived me to do a better work.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. A special thanks to my college's colleagues, Miguel Cruz and Artur Fonseca, that endured my stubbornness during the group projects at Instituto Superior Técnico, and helped me grow into a better person. Thank you.

To each and every one of you – Thank you.

Abstract

Cloud Computing has been successful in providing large amounts of resources to deploy scalable and highly available applications. However, there is a growing necessity of lower latency services and cheap bandwidth access to accommodate the expansion of IoT and other applications that reside at the internet's edge. The development of community networks and volunteer computing, together with the today's low cost of compute and storage devices, is making the internet's edge richer and filled with a large amount of still under utilized resources. Due to this, new computing paradigms like Edge Computing and Fog Computing are emerging.

This work presents Caravela[1][2], a Docker's container orchestrator that utilizes volunteer edge resources from users to build an Edge Cloud, where it is possible to deploy applications using standard Docker containers. Current cloud platform solutions are mostly tied to a centralized cluster environment deployment. So, Caravela employs a completely decentralized architecture, resource discovery and scheduling algorithms to cope with: the large amount of volunteer devices, volatile environment and, wide area networks that connect the devices and nonexistent natural central administration.

Keywords

Cloud Computing; Edge Computing; Fog Computing; Volunteer Computing; Edge Cloud; P2P; Resource Scheduling; Resource Discovery; Docker.

[1]Caravela (a.k.a *Portuguese man o'war*) is a colony of multi-cellular organisms that barely survive alone, so they need to work together in order to function like a single viable animal.

[2]Code available at https://github.com/Strabox/caravela

Resumo

O sucesso da Computação na Nuvem deve-se ao facto de oferecer uma grande quantidade de recursos, possibilitando aos seus utilizadores o uso desses recursos para executar aplicações escaláveis e com grande disponibilidade. Apesar do crescimento da Computação na Nuvem, há uma crescente necessidade de redução das latências e aumento de largura de banda barata para acomodar a expansão de aplicações da Internet das Coisas (IoT) e de outras que residam no extremo da internet. O desenvolvimento de redes comunitárias e da computação voluntária, juntos com o custo cada vez mais baixo de dispositivos com capacidade computacional e de armazenamento, faz com que existiam grandes quantidades de recursos no extremo da internet que estão muitas vezes em sub-utilização. Devido a estes requisitos e condições, novos paradigmas de computação têm surgido como a *Edge Computing* e a *Fog Computing*.

Este trabalho propõe a Caravela[3][4], um orquestrador de contentores Docker que utiliza dispositivos oferecidos voluntariamente pelos utilizadores para construir uma nuvem no extremo da rede, onde é possível executar aplicações usando contentores Docker. Os orquestradores existentes utilizam arquitecturas centralizadas e algoritmos de escalonamento que não conseguem lidar com ambientes voluntários, como as redes comunitárias, onde existem grandes quantidades de dispositivos conectados por redes dispersas geograficamente onde não existe nenhuma administração central natural para os gerir.

Palavras Chave

Computação na Nuvem; *Edge Computing*; *Fog Computing*; Computação Voluntária; P2P; Escalonamento de Recursos; Descoberta de Recursos; Docker.

[3]Caravela (a.k.a *Caravela-portuguesa*) é uma colónia de organismos multicelulares que não conseguem viver sozinhos, por isso trabalham juntos como um único animal para poderem sobreviver.

[4]Código disponível em https://github.com/Strabox/caravela

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
CC	Cloud Computing
CLI	Command Line Interface
CLR	Common Language Runtime
DHT	Distributed Hash Table
EC2	Elastic Compute Cloud
EC	Edge Computing
FC	Fog Computing
GUID	Global Unique Identifier
IPFS	InterPlanetary File System
ISP	Internet Service Provider
IaaS	Infrastructure-as-a-Service
IoT	Internet of Things
JVM	Java Virtual Machine
LoC	Lines of Code
MEC	Mobile Edge Computing
MNO	Mobile Network Operator
NAT	Network Address Translation
OS	Operating System

P2P	Peer-to-peer
PaaS	Platform-as-a-Service
QoS	Quality of Service
RAN	Radio Access Network
REST	Representational State Transfer
RPI	Raspberry PI
SDK	Software Development Kit
SLA	Service Level Agreement
SPoF	Single Point of Failure
SVM	System-level Virtual Machine
SaaS	Software-as-a-Service
TOML	Tom’s Obvious, Minimal Language
TTL	Time To Live
VC	Volunteer Computing
VM	Virtual Machine
WAN	Wide Area Network

1 Introduction

Contents

Cloud Computing (CC) is in a mature stage with heavily usage due to its advantages as resource elasticity, no upfront investment, global access and many more [1]. In order to meet these attractive properties, normally CC is implemented with a set of a few geo-distributed energy hungry data centers at the internet's backbone. This makes CC operate far from the internet's edge with Wide Area Network (WAN) latencies and expensive bandwidth to reach it.

The Internet of Things (IoT) is expanding at huge rate, as stated by CISCO [2], it is filling the network's edge with a lot of data production. Trying to push all of this data into the Cloud, in order to process is costly (in terms of bandwidth) and it will saturate the internet's backbone. Pre-processing data at the edge would reduce the amount of data needed to be transmitted to the Cloud (to be stored in permanent storage and/or performing heavier computations), thus reducing the transmission cost. Latency sensitive applications, e.g. self driving car, smart cities, cannot tolerate WAN latencies. Community networks are also growing, with today's storage and compute power inexpensiveness: laptops, desktops, Raspberry PI (RPI) [3, 4], computing boxes, routers and others. The edge of the network is filled with compute power and storage that most of the times are under utilized. A movement, commercial and academic, is ongoing to leverage this "Edge Power" to provide services with smaller latencies and cheaper bandwidth to the end users.

1.1 Edge Computing Vs Fog Computing Vs Mobile Edge Computing

The current literature still does not have yet come up with consensual definitions for many of these terms since it is still a field in its infancy. Here we present two of the most widely accepted definitions. CISCO [5] and Vaquero et al. [6] use Fog Computing (FC) and Edge Computing (EC) interchangeably as seen in the following definitions.

1.1.1 CISCO Definition

Fog/Edge Computing is a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data Centers, typically, but not exclusively located at the edge of network. [5]

1.1.2 Vaquero et al. Definition

Fog/Edge computing is a scenario where a huge number of heterogeneous (wireless and sometimes autonomous) ubiquitous and decentralized devices communicate and potentially cooperate among them and with the network to perform storage and processing tasks without the intervention of third parties. These tasks can be for supporting basic network functions or new services and applications that run in a sandboxed environment. Users leasing part of their devices to host these services get incentives for doing so. [6]

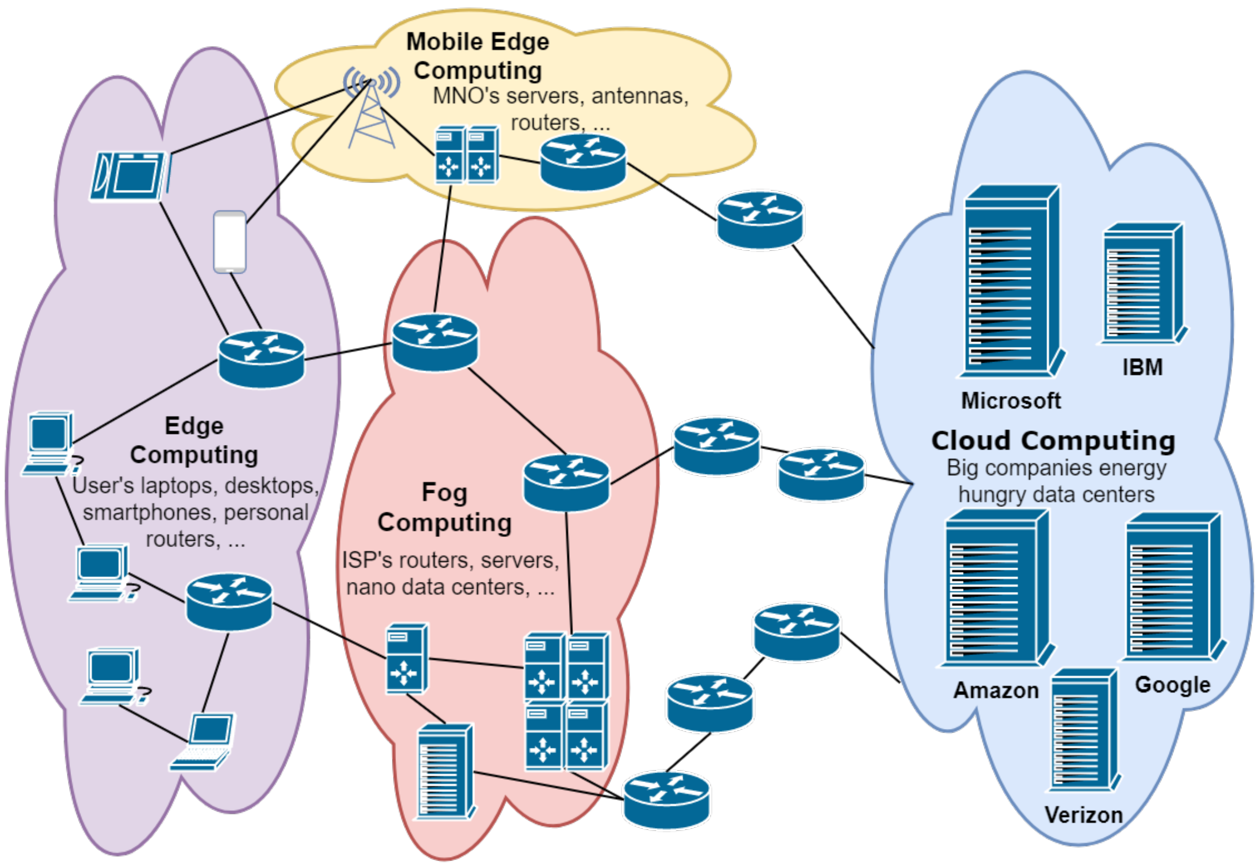

Mobile Edge Computing
MNO's servers, antennas, routers, ...
Edge Computing
User's laptops, desktops, smartphones, personal routers, ...
Fog Computing
ISP's routers, servers, nano data centers, ...
Microsoft
IBM
Cloud Computing
Big companies energy hungry data centers
Amazon
Google
Verizon

Figure 1.1: Today's relevant computing paradigms and its resource owners.

As we can see there is a main difference between them which is the ownership of the resources that compose this "new" layer. Vaquero et al. suggests that users lease part of their devices (e.g. Laptops, Workstations, etc) to build the Fog/Edge layer. While CISCO definition does not explicitly say but it is known that they provide their own specific solutions (routers, computer boxes, ...) [7] for Fog/Edge Computing. The real target of CISCO is mainly to support IoT specific applications using their own solutions, in order to build a layer that is near the IoT devices providing low latencies and compute power to help with more intensive computing tasks.

Mobile Edge Computing (MEC) has some common goals with the Fog/Edge Computing, e.g. in reducing latency of services, but it targets the support of mobile devices and its specific constraints, e.g. providing location awareness services. Most of the work in MEC targets Radio Access Networks (RANs) owned by Mobile Network Operators (MNOs). Cloudlet is another term coined, it appearis in some literature [8, 9] that normally consists in cloud solutions to support mobile computing but not focused in RANs deployments.

A graphical view of the computing paradigms is pictured in Figure 1.1. As seen, mobile devices can connect to different types of networks. We separate the EC from the FC due to the physical ownership of the resources. In some sense FC is an extension of the traditional CC to the edge/fog frontier, while EC is mostly powered by

EC users own devices. We will discuss the ownership of the resources in depth in Section 2.1.1.

1.2 Edge Cloud

In a lack of a specific widely accepted definition for Edge Cloud in the current literature (to the best of our knowledge), we will introduce here one based on Definition 1.1.2. EC can be regarded in part as a successor to Volunteer Computing (VC), as deriving from it. VC is, in essence, a paradigm where users offer their own devices computing power joining a distributed system in order to use some desired functionality or execute some workloads. It started with SETI@Home [10] where users lent their computers to perform computations to discover signs of extra terrestrial life. Recently, and more similar to an Edge Cloud, Cloud@Home [11] appeared, employing the users own resources to power a Cloud.

1.2.1 Definition Proposal

Edge Cloud is a synthesis of Cloud Computing, Edge Computing and Volunteer Computing. It provides an environment similar to Cloud Computing with seemingly unlimited set of resources, managed by a virtualized platform, where users deploy their applications easily. It is powered by the computing, storage and network resources provided by the volunteer and widespread users' own personal devices (laptops, workstations, routers, ...) that reside at the internet's edge.

1.3 Edge Cloud Challenges

The environment of Edge Clouds brings up great challenges [6, 12, 13] that are summarized below. From here onwards in this document, the terms device, machine and node are used interchangeably to represent workstations, laptops, etc.

- **Scalability**: The architecture design should be very scalable to accommodate large number of devices that can participate to provide increasing power.
- **Wide Area**: The devices in this type of cloud are geographically spread suffering from moderate latencies and poor bandwidth connections between them.
- **Self Management**: Avoid the need of administrators to manage it since it is built from all users' resources and does not exist any natural central administration.
- **Fairness**: It should enforce fairness mechanisms to usage because the resources are contributed by multiple users, and no user should be able to abuse from the other users' resources.
- **Support of device Heterogeneity**: Since different users contribute with their resources, it should support hardware and software heterogeneity.

- **Isolation**: Users' applications will run in other users' machines so it is necessary to isolate the cloud platform from the underlying private user resources.
- **Multi-Tenant Support**: To maximize the use of the resources it should be possible to consolidate applications from different users in the same device.
- **Ease to Use**: Make it simple to contribute with resources and deploy applications because the success of its volunteer part depends on the user interest.
- **Usage Flexibility**: Give the users the possibility to specify requirements for theirs applications in terms of resources needed, using Service Level Agreements (SLAs)[1].
- **Churn Resilience**: The edge devices are not very reliable, and users can put and take away their devices from the cloud at anytime, so it should adapt to this by degrading its performance gracefully.

1.4 Cloud Containerization

To build any cloud, multi-tenancy and isolation are fundamental requirements, normally being provided by virtualization techniques. Traditional clouds are powered by System-level Virtual Machines (SVMs), managed by hypervisors (e.g. Xen, QEMU, ...), but Operating System (OS) level virtualization (e.g. LXC containers, Docker[2]) is now entering in the game, mainly with Docker containers leading it. Below there is is a brief comparison between both technologies [14–16].

- **System Virtual Machines**
  - **Full isolation** of instances in same node making it the perfect choice in multi-tenant environments like CC. Containers have less isolation and can be attacked.
  - One node can contain SVMs **instances with different OSs and versions** which is more flexible to the application developers and to the cloud provider deployment algorithms.
- **Containers**
  - **Performance is equal or better** [14] at instance launch, startup time, stop time and node resources usage, due to the fact that each container does not need a specific OS layer that consumes resources. At the downside deployable images are limited to compiled binaries for the host's OS and CPU architecture.
  - **Container images are much smaller** than SVM counterparts because they do not require having an OS kernel inside, making it more scalable, faster and cheaper to move around and maintain.

[1]Since offering the typical performance SLAs is difficult in a volunteer environment we consider SLAs to be only resources necessary to run the application.
[2]https://www.docker.com/

There are other possibilities, e.g. using OSGI framework[3] that uses as resource isolation application components, but these offer less isolation (even when enhanced [17, 18]), using a Java Virtual Machine (JVM), compared to the previous ones.

Today's personal devices are very powerful, yet the gap to the more powerful devices in data centers still occurs, so, the container's performance and size seem more suitable for an Edge Cloud environment. The size makes it easy to transfer in the wide area and low bandwidth scenario of the Edge Clouds compared to SVMs.

1.5 Current Shortcomings

Most of current Open Source solutions to cloud platforms: OpenStack[4], OpenNebula[5], Swarm[6] and others have very centralized internal architectures and algorithms that mostly fit only in small-medium, homogeneous and controlled environments like clusters, not in volunteer and edge environments. The current literature in Edge Clouds has few functional and deployable prototypes, and as in Open Source solutions centralized management prevails in most of the works. The fairness in volunteer systems has been studied for a while in volunteer P2P systems, but real attempts to introduce it in a cloud solution are rare [12].

1.6 Contributions and Goals

Our fundamental contribution is the development of Caravela: a distributed and decentralized Edge Cloud that can leverage volunteer resources from multiple users (e.g. laptops, workstations, spare RPIs[7], ...), allowing them to deploy their applications on it, just by using standard Docker containers. The individual work goals are:

- Investigate the state of the art and previous researches in Edge/Fog Computing (more concretely in Edge Clouds), scheduling and discovery resource algorithms in CC and fairness mechanisms in CC and Distributed Systems;
- Produce a taxonomy to classify the works in Edge Clouds, Resource Management and Usage Fairness;
- The implementation of Caravela, a Docker's container orchestrator prototype with the following properties:
  - **Decentralization**: We propose a distributed and decentralized architecture, resource discovery and scheduler algorithms to avoid Single Point of Failure (SPoF) and bottlenecks to cope with the large number of volunteer resources and wide area of deployment;
  - **User Requirements**: Users should be able to specify the class of CPU (reflects CPU speed), number of CPUs and RAM they need to deploy a container. It should be possible to deploy a set

[3]https://www.osgi.org/developer/architecture/
[4]https://www.openstack.org/
[5]https://opennebula.org/
[6]https://docs.docker.com/engine/swarm/
[7]Raspberry PIs

of containers that form an application stack, specifying if the user wants the containers in the same node, promoting co-location, or spread them over different nodes, e.g. promoting resilience.

- The implementation of a simulator called CaravelaSim with the following properties:
  - **Scalability**: Allows to test Caravela's architecture and algorithms with hundreds of thousands of nodes in a reasonable amount of time;
  - **Parameterizable**: Allows to parameterize the simulator with different request profiles, and different request flows, to test the system under several situations, e.g. steady state, high load.
- Produce a synthetic benchmark for CaravelaSim to test the system under a realistically Edge Cloud scenario;
- Evaluate the prototype, present and analyze the results.

1.7 Document Structure

The rest of the document is organized in the following way: Chapter 2 presents our analysis of the related work. Chapter 3 presents Caravela's architecture and algorithms. Chapter 4 describes the major implementation details of Caravela and CaravelaSim. In Chapter 5 we present the evaluation to our work. Lastly, Chapter 6 concludes the document and wraps up with the important remarks and future work.

2

Related Work

Contents

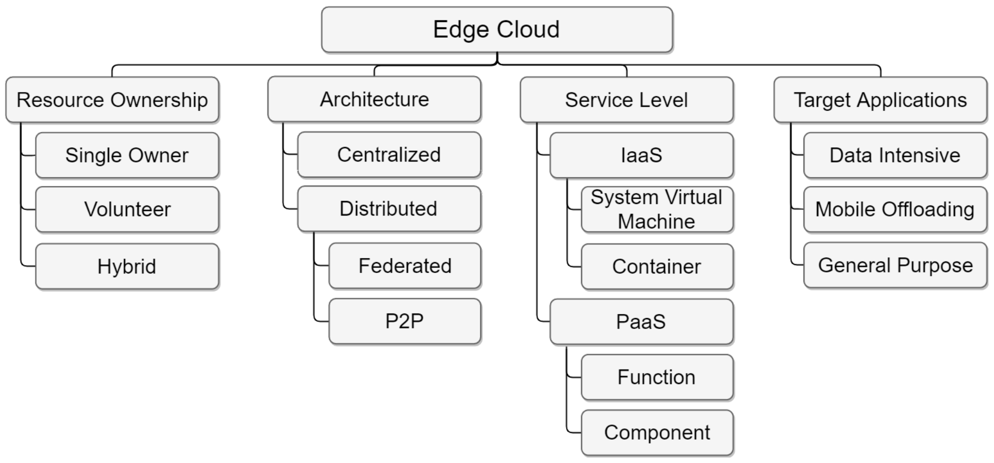


Figure 2.1: Edge Clouds taxonomy.

In this chapter we present the fundamental and state of art, academical and commercial, work in the development of Edge Clouds in Section 2.1, Cloud Resource Management in Section 2.2 and System Usage Fairness in Section 2.3. Lastly we present Relevant Related Systems in Section 2.4.

2.1 Edge Clouds

A definition for Edge Clouds was already introduced in Section 1.2. In this section we present a taxonomy to classify them. Since this is still a recent research area, we did not find any in the current literature (to the best of our knowledge). We present here the main characteristics that distinguish Edge Clouds: **Resource Ownership**, type of **Architecture**, **Service level** and **Target Applications**. The taxonomy is pictured in the Figure 2.1.

2.1.1 Resource Ownership

Resource Ownership distinguishes **who owns** the physical devices that power the Edge Cloud. There are three types: Single Owner, Volunteer and Hybrid.

Single Owner Edge Clouds are build with devices owned by a single entity (individual or collective). It can be easily envisioned that a large retailer chain use all the under utilized computing power spread across offices and shops to offer a cloud like environment, where they can deploy data mining workloads in order to discover buying tendencies. This approach would have the advantage of no information disclosure over the traditional CC because the infrastructure belongs to them. Single Edge Clouds are here to represent the extreme case of an Edge Cloud because the most common case consists in using volunteer devices.

Volunteer Edge Clouds is where CC and VC intersect. Resources are provided by users' personal devices (e.g. Cloud@Home [11], Satyanarayanan et al. [9], Cloudlets [8], Babaoglu et al. [19] and Mayer et al. [20]). The users have incentives to join, e.g. with the possibility to deploy their own applications. Volunteer Edge

Clouds have a potential to amass a large number of widespread resources resulting in virtually unlimited computational and storage power.

Hybrid Edge Clouds are a mixture of the single owner and volunteer types. In these clouds a large slice of the infrastructure belong to a single entity (usually but not mandatory a FC entity like Internet Service Providers (ISPs)), normally the more powerful nodes that operate at management layer belong to this single entity while the remaining devices are volunteer resources from multiple edge users. The volunteer resources are hooked to the management layer providing the computational and storage power to the cloud (e.g. Nebula [21], Chang et al. [22] and Mohan et al. [23]).

2.1.2 Architecture

Edge Cloud's Architecture reflects **how the nodes are structured and managed**, because in terms of computation placement, by definition, all Edge Clouds are highly distributed due to the widespread area of deployment. We have two main architectures types: Centralized and Distributed.

Centralized Edge Clouds have dedicated nodes (normally the ones that have management responsibilities) in a central physical place while the widespread resources at the Internet's edge connect to them providing the computational and storage power to where the user's tasks are offloaded (e.g. Cloud@Home [11] and Nebula [21]). These systems tend to not scale well because all the resources are managed from a centralized location, by a small set of nodes compared to the number of edge nodes.

Distributed Edge Clouds are divided into two sub-types: *Federated* and *P2P*. These types tend do scale better for large amount of resources than centralized ones because the management and coordination is independently distributed across the system nodes. Due to this they do not suffer from SPoF too. A *Federated* Edge Cloud has a built-in notion of autonomous smaller clouds (also called zones in some literature [12]) that can provide services alone but cooperate with other nearby autonomous cloud to provide even more powerful services, for example in cases of high loads (e.g. Nebulas [24]). On the other hand, a *P2P* Edge Cloud is a decentralized network of widespread nodes that connect among themselves and provide a cloud platform as a whole entity without using central nodes for coordination. It makes all the nodes equal in terms of responsibilities in the system. So if some nodes fail the cloud can continue operating (e.g. Babaoglu et al. [19]).

2.1.3 Service Level

As in the traditional CC world, Edge Clouds also provide different levels of services depending on the **type of resources provisioned** to users (e.g. System Virtual Machine, Container, Programming Runtime), and **how automatic is its management**. At service level we can distinguish Edge Clouds as Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS). There are examples of Software-as-a-Service (SaaS) but normally they only exploit FC nodes in order to extend the CC range to the edge, maintaining some control over the nodes that execute games client locally. The work of Choy et al. [25] studied if deployment of cloud games with constrained latency requirements was improved using fog servers instead of Amazon's EC2[1] service alone. Due to the focus in the FC we did not consider it in our taxonomy.

IaaS Edge Clouds provide an infrastructure that provides CPU cycles, RAM, disk storage and network capabilities to deploy arbitrary stacks of software. Users can control the deployment of applications using a simple Application Programming Interface (API), e.g.: launching more instances of the application when needed. In IaaS the management is application independent, e.g. user should deal with the application auto scaling explicitly. The resource provisioned comes with two flavors: *System virtual Machines* and *Containers*. Normally IaaS provides full SVMs to the users where they can choose the favorite OS and install any kind of software. As discussed in section 1.4, Containers are starting to be used to power cloud, specially because the images are lighter and tools build around them like Docker provides cross platform use, application packaging and easy application stacks deployment.

PaaS Edge Clouds usually provide language runtimes (e.g. .NET Common Language Runtime (CLR), JVM and more) and application frameworks to deploy users application code hiding the manual application management and deployment that users need to do explicitly in IaaS (e.g. Verbelen et al. [8]). The main sub-types identified are: providing *function* code and *application components* (e.g. OSGI bundles) that usually run computational heavy tasks.

2.1.4 Target Applications

Edge Clouds by definition already have good properties for some specific applications, e.g. lower access latencies because they are at the internet's edge. But some of the works have design choices that can be leveraged by specific applications workloads. Here we have three main categories identified in our research: Data Intensive, Mobile Offloading and General Purpose.

Data Intensive Edge Clouds are built to support workloads that consist in processing large amounts of data, usually files. They evidence characteristics that optimize it, for example Nebula [21] tries achieve co-location of the data files and the processing code in the same node to avoid transfer large amounts of data through the network. Other example is the work of Costa et al. [26], where a map-reduce framework is implemented over volunteer resources.

Mobile Offloading Edge Clouds provide environments where the resource constrained mobile devices can easily (and usually transparently to the mobile applications) offload tasks or part of applications that are computational heavy to run, e.g. a face recognition module. Cloudlet work [8] tries to offer a transparent way of offloading mobile application components to Edge Clouds in order to leverage the seemingly infinite amount of resources. Martins et al. [27] work goes even further, building a cloud with the own mobile devices power.

General Purpose Edge Clouds did not fit in the previous categories, because they do not specify particular characteristics and features that would enhance the execution of specific types of applications/workloads.

[1] https://aws.amazon.com/pt/ec2/

System/Work	Resource Ownership	Architecture	Service Level	Target Application
Cloud@Home [11]	Volunteer	Centralized	SVM	General Purpose
Satyanarayanan et al. [9]	Volunteer	Federated	SVM	Mobile Offloading
Cloudlets [8]	Volunteer	Federated	Component	Mobile Offloading
Babaoglu et al. [19]	Volunteer	P2P	SVM	General Purpose
Mayer et al. [20]	Volunteer	P2P	Component	General Purpose
Nebula [21]	Hybrid	Centralized	Function	Data Intensive
Chang et al. [22]	Hybrid	Federated	Containers	General Purpose
Edge-Fog Cloud [23]	Hybrid	P2P	?	General Purpose

Table 2.1: Edge Cloud works classification.

Table 2.1 contains the Edge Cloud works identified in our research and the respective classification considering the taxonomy presented above. Entries with ? means that it is not explained in the paper or it can not be extracted from the description.

2.2 Resource Management

The management of distributed resources consist in two main stages: Resource Discovery and Resource Scheduling [28]. Resource discovery (a.k.a Resource Provisioning) focus in discover the resources for a given request, obtaining the addresses (e.g. IP addresses) and its characteristics (e.g. RAM available). Resource schedulers redirect the user requests to a subset of the resources discovered. In this section we present the fundamental and latest work on **Resource Discovery** and **Resource Scheduling** mainly applied to Cloud Computing but in a few cases Grid Computing. In the rest of this section SVM, Virtual Machine (VM) and Container are used interchangeably to represent an IaaS deployable image.

2.2.1 Resource Discovery

In order to take decisions on how to schedule users requests among the system's resources it is vital know the **amount** of free resources, **where** they are (e.g. IP address), the **current state** (e.g. 80% CPU utilization) and its **characteristics** (e.g. dual core CPU). From here onwards a user request consists in a VM image with a list of requirements (e.g. necessary CPU and RAM power) for its execution, this is the typical case in IaaS. With accurate and up-to-date resource data, the schedulers can take better decisions that benefit the user and the system itself, e.g. high resource utilization with good Quality of Service (QoS). Resource discovery strategies have several main differentiating features [29] that we discuss below in detail: **Architecture**, **Resource Attributes** and **Query**. The Figure 2.2 presents a taxonomy for the works in resource discovery. The majority of the works presented in this Section are from the Grid Computing [30] and P2P environments because the traditional CC did not have the resource discovery problem due to the natural

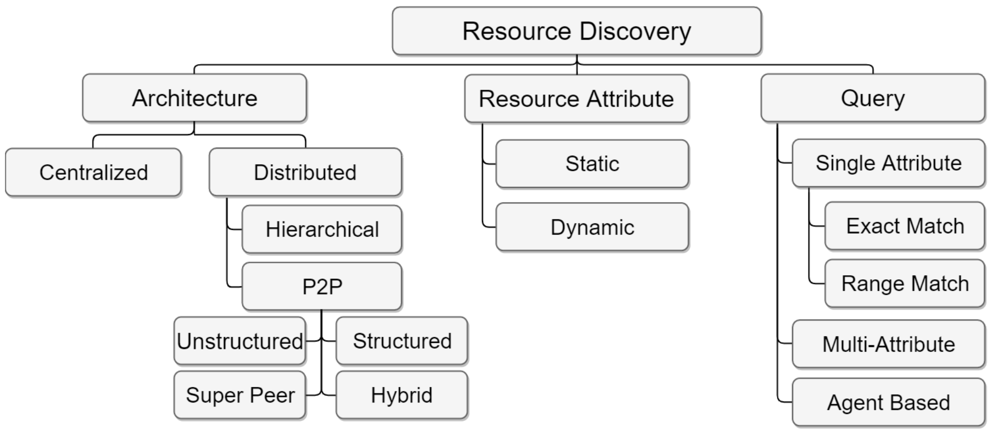


Figure 2.2: Resource Discovery taxonomy.

centralized architecture.

2.2.1.1 Architecture

The architecture choice has a large impact on **how the search for resources is made** which determines the scalability, robustness, reliability and availability of the mechanism. It is highly coupled with the network topology of the system nodes. There are two main classes of resource discovery architectures: Centralized and Distributed.

Centralized resource discovery mechanisms consolidate in a single node (or set of replicated nodes) the knowledge of all the system resources and its state using a client-server approach. This has the advantage of easily implementing algorithms that find the optimal resource for a request. The downside is that it does not scale for large systems because that node(s) is/are a bottleneck and a SPoF for the entire system.

Distributed resource discovery mechanisms split the knowledge of the system through multiple nodes that cooperate with each other to find the resources. Distributed approaches have higher scalability, robustness and availability because the knowledge is spread among multiple independent system nodes. This approach includes two sub-types: *Hierarchical* and *P2P*. *Hierarchical* resource discovery usually split the system's nodes in managers and workers. Workers form static groups that are managed by a manager. The managers form a tree. Managers advertise to the meta-managers (managers of managers) its resources normally in a digest form in order to increase the scalability while loosing data precision. It has the drawbacks of SPoF in root node and for each group in the respective manager. *P2P* discovery mechanisms tend to be even more scalable and fault tolerant because the discovery mechanism is responsibility of all nodes (or a large subset).

There are four sub-types of P2P mechanisms: *Unstructured*, *Super Peer*, *Structured* and *Hybrid*. *Unstructured* discovery mechanism do not use a specific peer overlay topology. The discovery normally is made with flooding strategies or informed search plus random walkings. It usually generates a massive number of messages that flood the network. It suffers from false-negatives because such strategies use Time To Live (TTL) mechanisms to avoid crushing the system, making possible the existence of the searched resource somewhere in the system. It has the advantages of being highly tolerant to peer churn.

Super Peer is a mixture of *Centralized* and *Unstructured*. A Super Peer node is responsible for a set of regular peers' resources, acting as a centralized server. They cooperate between each other to find the resources, traditionally they use flooding strategies. This approach generates less false-negatives than unstructured because there is a small space to search (super peer network). It has the drawback of SPoF in the super peers.

Structured enforce a overlay topology of the peers (e.g. ring with Distributed Hash Table (DHT) on top, Euclidean Spaces and more). These approaches tend to have no false-negatives, e.g. DHT approaches tend to find the resources typical in $O(log(N))$ hops, with N being network size. In order to achieve the completeness property (always find the resource if it exists) we need to pay the price to maintain the structure in the presence of peer churn. These approaches are only scalable if we can distribute the load of the discovery process throughout the structure. *Hybrid* approaches are combinations of the three previous ones, e.g. Super Peer architecture with super peers participating in a DHT. These approaches try to take the advantages of multiple P2P approaches.

2.2.1.2 Resource Attributes

The resource attributes **specified by the users in the requests** can be: total RAM memory installed in the resource, total RAM memory available in a specific point in time, number of CPU cores, amount of disk storage available and more. The resources attributes can be: Static or Dynamic.

Static resource attributes never change, for example the total memory installed in a node or the maximum clock speed. Discovery mechanisms for static attributes are easier to make due to its static nature because once the system know the resource value it is never needed to take actions in order to update it.

Dynamic resource attributes change during the execution, e.g. the total memory available on a node in a moment or the amount of CPU being used. Dynamic attributes are harder to incorporate in a discovery mechanism, because the necessity of updating its value is always a overhead to the mechanism, and make it scale to a large amount of nodes is a difficult task. This is the more interesting type of attributes in CC in order to provide a view on how loaded are the nodes in terms of CPU, RAM, disk space and even network quality.

2.2.1.3 Query

A query consists in **a request for a resource** with a given set of attributes, e.g. finding a node that has a Linux distribution installed, minimum of three CPU cores and at least 500MB of memory available. The queries supported by the discovery mechanisms are split in three main categories: Single Attribute, Multi-Attribute and Agent Based.

Single Attribute queries are the ones that match only one attribute of the resource. Most of the resource discovery system from the great era of file distribution using P2P systems, like Gnutella system, only used the file name as attribute to find the file in the system. A single attribute query can be sub-divided in two types:

System/Work	Architecture	Resource Attribute	Query
OpenStack	Centralized	Dynamic	Multi-Attribute Range Match
Cardosa et al. [31]	Hierarchical	Dynamic	?
Iamnitchi et al. [32]	Unstructured	Dynamic	Multi-Attribute Exact Match
CycloidGrid [33]	Super Peer	Static	Multi-Attribute Exact Match
Hasanzadeh et al. [34]	Super Peer	Dynamic	Multi-Attribute Range Match
Kim et al. [35]	Structured (DHT)	Static	Multi-Attribute Range Match
Kargar et al. [36]	Structured (Nested DHTs)	Dynamic	Multi-Attribute Range Match
Cheema et al. [37]	Structured (DHT)	Dynamic	Multi-Attribute Range Match
SWORD [38]	Structured (DHT)	Dynamic	Multi-Attribute Range Match
HYGRA [39]	Structured (Euclidean Field)	Dynamic	Multi-Attribute Range Match
NodeWiz [40]	Hybrid (Super Peer + Distributed Tree)	Dynamic	Multi-Attribute Range Match
Papadakis et al. [41]	Hybrid (Super Peer + DHT)	Dynamic	?
Kakarontzas et al. [42]	?	Static	Agent Based

Table 2.2: Resource Discovery works classification.

Exact Match and *Range Match*. *Exact Match* means that the query only supports $=$ operator over the attribute, e.g. find a node with exactly four cores. This is restrictive since sometimes user may accept a range of values for the attribute. *Range match* queries allow user to use operators like $<$, $>$ to attributes that have ordered value sets. An example is to find a resource with at least 250MB of memory available. It is more difficult to implement these queries in a scalable and distributed way because it is necessary maintain the resource knowledge ordered.

Multi-Attribute queries allow users to specify several attributes constraints over a resource. These queries are a set of single attribute queries united with logical operators like AND, OR, NOT and more depending on the logical algebra supported by the system. This type gives a great flexibility to the user but increases the complexity of the discovery mechanism because it is necessary search for a resource in different dimensions at same time. An example is finding a node that has at least 500MB of memory available and 3 CPUs free. This type of queries is not well supported by all the architectures presented above, e.g. P2P structured architectures using DHTs that support this type of queries are not trivial.

Agent Based queries use smart agents (pieces of code) that are deployed to the system (usually with a P2P architecture) in behalf of users. They are programmed with logic to find and negotiate the resources that user wants. The agent's logic travels through the system making multiple queries to the nodes. They are normally able to do multi-attribute exact match queries since they interact directly with the nodes that have the resources. This type is different from the traditional queries that contains the resource attributes constraints

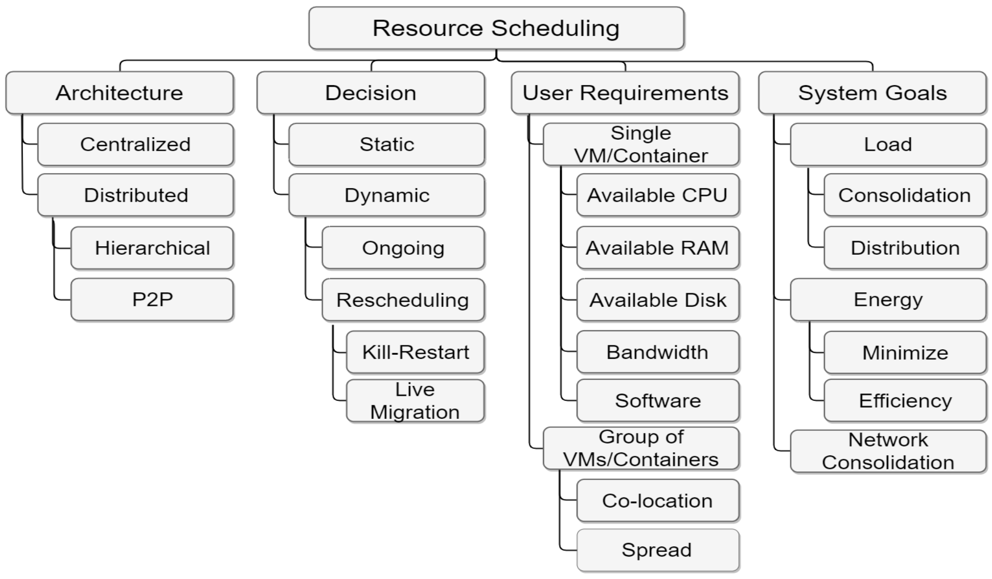


Figure 2.3: Resource Scheduling taxonomy.

only. The traditional queries are interpreted by the nodes' code. Agent based have security issues due to the potential to inject malicious code that run on the nodes. Furthermore, deciding the path the agent travels is hard and can affect scalability and completeness.

Table 2.2 presents a list of works in resource discovery classified using the taxonomy presented above.

2.2.2 Resource Scheduling

Resource scheduling is composed of three processes: Resource Mapping, Resource Allocation and Resource Monitoring [28]. The discovery mechanism provides several suitable resources (according to the SLAs) for a request. The scheduler implements strategies to decide, from all of the suitable resources, which one receives the request (a.k.a Resource Mapping). After the mapping is done, the scheduler reserves the resources and redirects the request to them (a.k.a Resource Allocation). In dynamic systems where the user applications running have a dynamic consumption of resources, the schedulers monitor the resources in order to know if applications are under/over the consumption settled in the SLAs in order to take actions like VM migration (a.k.a Resource Monitoring). Below we present a taxonomy to classify the work done in resource scheduling more concretely in Cloud Computing [28] with some mentions to P2P environment [43]. The resource scheduling is divided in four main dimensions: **Architecture**, **Decision**, **User Requirements** and **System Goals**. The resource scheduling taxonomy is pictured in Figure 2.3.

2.2.2.1 Architecture

The architecture of a system characterizes **what** system nodes participate in the scheduling decision and how they are **organized**. Our research revealed that the scheduling mechanism architecture is most of the times the same as the discovery mechanism architecture underneath. There are two main types of architectures: Centralized and Distributed.

Centralized resource scheduling only uses a single scheduler node (or coherently replicated set of scheduler nodes) to take the decision. All the scheduling requests go through the same scheduler causing a bottleneck and a SPoF. It has the advantage of capturing all the requests that enter in the system. Therefore it can easily enforce global strategies and goals because it schedules all the requests.

Distributed resource scheduling uses multiple independent nodes that make the scheduling decisions. Due to the independent scheduling decisions between the nodes it is harder to enforce global level strategies but in other hand the scalability and fault tolerance increases. The distributed resource scheduling comes with two significantly different sub-types: *Hierarchical* and *P2P*. *Hierarchical* resource scheduling normally is used to split the workers nodes in smaller and static groups for the respective scheduler, it increases the scalability over the centralized type because the requests are split among several schedulers. Normally this is implemented in a hierarchical tree shape where the leaves are the worker nodes and the rest are schedulers. It continues to suffer from SPoF and bottleneck at the root node and the partitions are usually static limiting the scalability. *P2P* resource scheduling is where the majority of the system nodes are schedulers that communicate with each other (usually with a small subset of the schedulers called neighborhood) in order to forward the requests to the suitable resources. This type of resource scheduling is highly scalable and does not have a SPoF inheriting the properties from the P2P systems. It has the disadvantages of being difficult to implement and enforce system level goals since the decisions are locally taken. It is important that the scheduling requests are evenly split across the system schedulers in order to be truly scalable.

2.2.2.2 Decision

This characterizes **when** the scheduling decision is taken. There are two main types: Static and Dynamic.

Static decisions are taken at "compilation time" which means the decision does not depend on the current system state. An example of a static scheduling strategy is the Round-Robin algorithm in Eucalyptus cloud [44] platform that assigns VMs to the nodes in a round robin fashion regardless the current system state. This type is not good for systems that have very heterogeneous requests in terms of resources needs.

Dynamic decisions are taken online based on the current system state (built with resource discovery mechanisms). This type of decisions is fundamental to handle heterogeneous requests in terms of resources usage. They are fundamental to handle load variations in the system. There are two sub-types of dynamic decisions: *Ongoing* and *Rescheduling*. *Ongoing* decisions are the ones made by schedulers that after assigning a request to a node, it is never attempted to move the assignment to other node. Moving the request here means moving a VM to other node. This kind of decisions has the disadvantage of not trying to correct systems that tend to become unbalanced due to suddenly high loads, e.g. e-commerce web site during "Black Friday".

Rescheduling decisions are the ones that can be changed after a first assignment. This is usually done to re-balance the load in the nodes in order to fulfill the SLAs and/or maximize resources usage. It uses the information of the resource monitoring and resource discovery processes to make the decision. There are two main sub-types: *Live Migration* and *Kill-Restart*. *Live Migration* decisions are a specific type used in SVM allocation that support live migrations. This type of migrations allows the VM to execute while is being moved to other node. The work of Farahnakian et al. [45] is an example of a scheduling architecture and algorithm that leverage hypervisors abilities to do live migrations in order to save energy. *Kill-Restart* decisions terminate the instance in the node it was placed for the first time. After that it starts a new instance (from the same image as the first one) in another node. This last type of migration loses the state of the the first instance and it stops service requests while the new instance does not start.

2.2.2.3 User Requirements (SLAs)

Scheduling requests carries **resources requirements that user want to be fulfilled**, e.g. the VM should have at least 200MB of RAM to use during 95% of the time. The user requirements are also known as SLAs. There are two main types of user requirements: applicable to a Single VM or to a Group of VMs.

Single VM requirements are applicable to a single VM. There are five main requirements[2] that can be requested when scheduling a VM: *Available CPU*, *Available RAM*, *Available Disk Storage*, *Bandwidth* and *Software*. *Available CPU* is requested when a user wants a minimum specific amount of processing power to run the VM. *Available RAM* is requested by a user when he wants a minimum specific amount of available RAM that can be used by the VM. *Available Disk Storage* is requested when a user wants a minimum amount of disk storage that can be used by the VM. *Bandwidth* is a requirement for the amount of bandwidth a VM has in order to communicate. Since bandwidth is an end-to-end connection characteristic it is difficult to provide. *Software* can be requested by the user to accommodate code that needs a specific software to be presented in order to work correctly, e.g. when deploying the container the node needs to have Windows installed.

Group of VMs/Containers are usually required when user want to deploy a application that is composed by a set of components that run in a distributed fashion on different VMs. A typical case is a microservices bundle application. There are two sub-types of group requirements: *Co-location* and *Spread*. *Co-location* requirement is requested when user wants the several components of the application to be physical close. It is useful when a user wants to deploy an application that has a high intra-communication between the distributed components. *Spread* requirement is the opposite of the co-location and is requested when the user wants the application components to be physically spread. It is usually required when maximum availability of the application is necessary, to do so the components must be spread to tolerate spacial located failures in the underlying infrastructure.

[2]There are more variations that can be asked in some systems, like the number of machine cores and maximum installed RAM, but here we are focused in the most used in CC.

System/Work	Architecture	Decision	User Requirements	System Goals
Lin et al. [46]	Centralized	Static	?	Minimize Consumption
OpenNebula (Packing) [23]	Centralized	Ongoing	CPU & RAM	Load Consolidation
OpenNebula (Striping) [23]	Centralized	Ongoing	CPU & RAM	Load Distribution
OpenNebula (Load Aware) [23]	Centralized	Ongoing	CPU & RAM	Load Distribution
OpenStack (Filter Scheduling) [23]	Centralized	Ongoing	CPU, RAM & Disk	None
Lucrezia et al. [47]	Centralized	Ongoing	CPU, RAM & Disk	Network Consolidation
Selimi et al. [48]	Centralized	Kill-Restart	Co-location	Network Consolidation
Eucalyptus (Round Robin) [23]	Hierarchical	Static	CPU & RAM	Load Distribution
Eucalyptus (Greedy) [23]	Hierarchical	Ongoing	CPU & RAM	Load Consolidation
Jayasinghe et al. [49]	Hierarchical	Ongoing	CPU, RAM, Bandwidth & Co-location or Spread	Load Consolidation
Sampaio et al. [50]	Hierarchical	Kill-Restart	CPU	Power Efficiency
Snooze [51]	Hierarchical	Ongoing or Kill-Restart	CPU, RAM & Bandwidth	Load Consolidation or Distribution
HiVM [45]	Hierarchical	Live-Migration	CPU	Load Consolidation
Messina et al. [52]	P2P	Ongoing	CPU	Load Consolidation or Distribution
Feller et al. [53]	P2P	Kill-Restart	?	Power Efficiency

Table 2.3: Resource Scheduling works classification.

2.2.2.4 System Goals

While users want SLAs to be fulfilled, the **system provider** usually enforce goals into the system that must be met at the same time using different scheduling strategies. These scheduling strategies run during the mapping phase of the scheduling process. Good systems have the possibility to change these strategies/policies (on reboot). These systems are interesting because they decouple the architecture and the resource discovery mechanism from the scheduling, allowing this flexibility. Open source solutions, like Swarm, offer this flexibility to allow users to fine tune the system for their needs. There are three main categories of system goals: Load, Energy and Network Consolidation.

Load goals consist in controlling the amount of load (CPU usage, RAM usage, disk usage and even network usage) in the system nodes. We identified two main strategies to control the load: *Consolidation* and *Distribution*. *Consolidation* strategy is used to accommodate the maximum number of VMs in a single node while supporting the request's SLAs. This is used to explore the maximum potential of the node without compromising the user requirements. *Distribution* strategy is the opposite of the consolidation consisting on distributing the load evenly among all the system nodes. This strategy can be used to leverage all the available resources in order to offer better services to the users.

Energy goals consist in taking scheduling decisions considering the energy spent by the physical machines. These strategies come from the fact that a physical machine without any user's application running, consumes a lot of energy decreasing the system provider profits. There are two main strategies in energy saving goals: *Minimize Consumption* and *Power Efficiency*. *Minimize Consumption* of energy basically consists in turning off or putting in a deep sleep mode as many machines as we can in order to save energy. Strategies here tend to neglect the users SLAs in order to save money from the system provider or simple because the main focus is the ecological foot print. *Power Efficiency* strategy is similar to the minimize consumption but here we do not neglect users' SLAs. The idea is having turned on the minimum number of machines while satisfying the users SLAs. This is normally a combination of rescheduling and load consolidation strategies.

Network Consolidation goal consists in the allocation of VMs that communicate a lot in the same node, or in the physical nearby nodes, in order to reduce the traffic inside the system.

Table 2.3 presents the list of works in our research and the classification using the taxonomy of scheduling resources presented above.

2.3 System Fairness

The subject of system fairness has been studied along the years in the P2P and VC systems due to its collaborative nature [12, 54]. The majority of these systems have two main roles for the users: when a user offer resources, he is called a **Supplier**, and when the user wants to use/buy resources from other, he is called a **Buyer**. An example of these systems is the volunteer Edge Clouds. These systems need to employ control mechanisms, such as virtual currency in order to trade the resources between users or even reputation

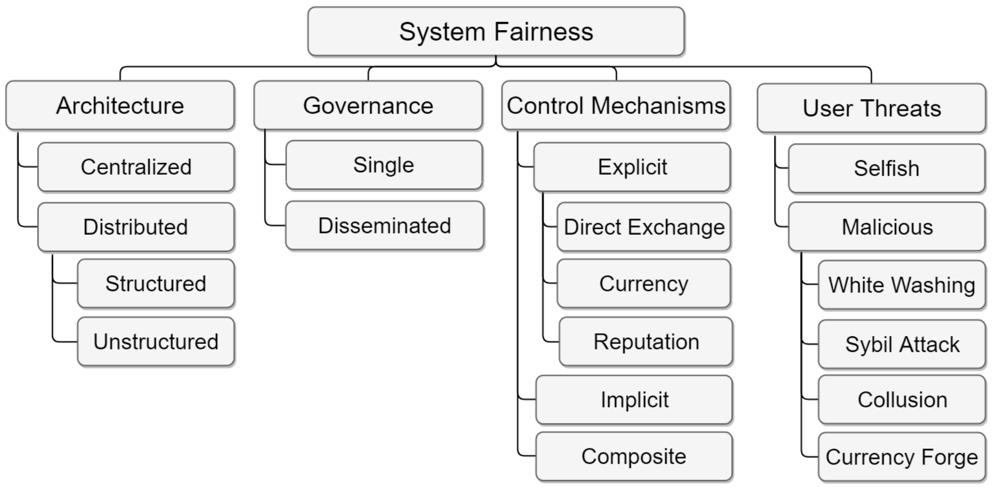


Figure 2.4: System Fairness taxonomy.

scores per user (that represent the trustworthiness) in order to discourage bad behavior. An example of bad behavior is when a user does not pay for the resources he used from other users. As a crude major goal it is desirable to maintain a system where a user can use it proportionally to what he contributes to it. In other hand it is important that these control mechanisms do not impose a significant overhead in the system, defeating its main purpose. In this section we classify these mechanisms using the four following dimensions: **Architecture**, **Governance**, **Control Mechanism** and **User Threats**. Figure 2.4 pictures the taxonomy.

2.3.1 Architecture

The architecture characterizes **how the system is organized**, there are two main approaches: Centralized and Distributed.

Centralized architecture is the most widely used. The fairness mechanism is imposed in a central location. Usually the clients interact with the system in a client-server fashion. This approach has the advantage of easily access all the information from all the users. The disadvantage is the maintenance of all the information in a central place making it hard to scale and susceptible to a SPoF.

Distributed architecture approaches have the control mechanism and information spread over all the nodes that participate in the system (e.g. Rodrigues et al. [55]). In this area the most studied systems are P2P so in this particular case distributed means P2P. P2P approaches are much more scalable since the knowledge of reputation scores and virtual currencies are spread over all the system nodes. In P2P systems a node only knows a small subset of nodes. The interaction with unknown nodes is a problem, because the node does not have information about them. Strategies like asking trustworthy nodes if they know the unknown nodes are used [56].

2.3.2 Governance

Governance characterizes **how many entities have the control /authority** over the system. There are only two types: Single and Disseminated.

Single type means that there is only one entity that controls the system (e.g. Ebay[3]). Users tend easily accept systems that have a trustworthy entity controlling it. This entity is responsible for controlling all transactions between nodes assuring its validity, similar to what a banks do with real money transactions. Single controlled systems are not correlated with centralized architecture since the entity could implement the system in a distributed fashion in order to scale.

Disseminated type means that multiple entities in the system cooperate, usually without central management, in order to control the fairness mechanisms (e.g. Rodrigues et al. [55]). This is the typical case of VC where the system is built with the user's own resources. Here we have multiple entities **using** and **controlling** the system. Therefore implementing mechanisms in this environment is far more difficult since the control is spread. Blockchain is a technology where the transactions between users are approved in a disseminated way (without any central control) among the network nodes.

2.3.3 Control Mechanisms

This dimension characterizes what are the main specific mechanisms that are used to **enforce the fairness** in the systems. We have two main categories of mechanisms: Explicit and Implicit.

Explicit mechanisms are specially designed and implemented in the system with the solely purpose of enforcing usage fairness. There are four main types: *Direct Exchange*, *Currency*, *Reputation* and *Composite*. *Direct Exchange (a.k.a Bartering)* mechanisms means that if a user wants a resource from another user it should give a resource with similar value back. This type of mechanisms is highly restrictive because with heterogeneous resources it is very difficult mapping the values between them, and the supplier user may not want/need the resource that the buyer offers back making the negotiations difficult. *Currency* mechanisms were introduced to solve the direct exchange problems. With this mechanism when a user wants a resource from other user it pays for it with a currency (virtual or real money). The resource prices could be fixed or dynamic. Fixed prices are not flexible since when a resource is scarce and the demand is high its value should increase in order to compensate the user for providing it. In order to rent resources from other users it is necessary rent the own resources in order to accumulate currency. This creates incentives for contributing to the system. *Reputation* score mechanisms have a different purpose than the previous two. It provides to the users a way to know how trustworthy is a given user. Before trading resources the user can decide to proceed or not based on that score. This score reflects the user behavior according to the system model behavior. Well behaved users tend to have higher scores than users that try to cheat the system. *Composite* mechanisms are a combination of the previous mechanisms (e.g. Rodrigues et al. [55]). Collaborative systems tend to

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

use currency and reputation at the same time. They use the currency to control the resources trade and the reputation to instigate well behavior in the users.

Implicit mechanisms were not created with the purpose of enforcing system fairness instead they are system constructs that serve other purposes but that can be leveraged to extract valuable information for the system fairness. A good example, is the page rank mechanism of the Google search engine. It uses the hyperlinks (that were created to navigate between pages) counting in each web page in order to calculate a score for the page importance.

2.3.4 User Threats

This dimension is a brief analysis to the most **common threats to the systems and the control mechanisms themselves**. We are not focused in doing a formal threat model analysis because that is out of our work scope. There are two types of bad behaviored users: Selfish and Malicious.

Selfish users try obtain resources from others without contributing/paying back, e.g. a user obtains resources from other users and always denies requests from them. The currency and reputation mechanisms combat this behavior because there is a necessity of having currency to obtain resources, and to obtain currency it is needed share resources. The reputation is usually used to combat users that do not respect the system rules. Users with low reputation scores tend to be be excluded from future transactions and interactions because nobody trusts them.

Malicious users try to exploit weaknesses in the control mechanisms of the fairness system in order to take advantage over others. Next we describe the most common exploits/attacks from this kind of users: *Currency Forge*, *White Washing*, *Sybil Attack* and *Collusion*. *Currency Forge* (a.k.a Double Spending [57]) is the most basic attack and consists in an user claiming that he has more currency than he really has. In order to solve it, usually there is a single trustworthy authority that controls the amount of currency available in the system, similar to banks and real money, it guarantees that there is no false currency in the system. A fully distributed and decentralized approach is the blockchain and distributed ledger used in the Bitcoin virtual cryptocurrency avoiding the need of a single authority making the currency regulate itself. *Sybil Attack* consists in a single user forging multiple system identities in order to take advantage of a larger participation in the system. A typical example is making successful transactions between the several forged entities increasing the reputation of them. The other users are tricked to interact with the attacker due to the high reputation of the fake identities. *Whitewashing* attacks consist in a user with low reputation to exit and enter with a new identity in order to have a new start. Systems that have an easy way to create new identities are susceptible to this attack. *Collusion* is similar to the sybil attack, but instead of a user creating multiple identities, multiple different malicious users try to take advantage of a larger participation in the system using similar strategies as the sybil attack.

Table 2.4 lists the researched systems and its classification considering the taxonomy presented above. The last column represents the attacks that the system can mitigate.

System/Work	Architecture	Governance	Control Mechanisms	User Threats
SocialCloud [58]	Centralized	Central	Implicit (friends connections) & Currency	Currency Forge & Whitewashing
Karma [59]	Structured	Disseminated	Currency	Selfish, Currency Forge, Whitewashing & Sybil Attack
Gridlet Economics [60]	Structured	Disseminated	Currency & Reputation	Selfish & Currency Forge
Rodrigues et al. [55]	Structured	Disseminated	Currency & Reputation	Selfish, Currency Forge & Whitewashing
VectorTrust [56]	Unstructured	Disseminated	Reputation	Sybil Attack & Collusion
Ebay	?	Central	Currency & Reputation	Selfish & Currency Forge

Table 2.4: Fairness works classification.

2.4 Relevant Related Systems

2.4.1 Docker Swarm (Commercial System)

Swarm[4] is a state of the art commercial Docker container orchestrator where it is possible to deploy Docker containers. It has a centralized architecture where a set of actively replicated nodes (called managers), using the Raft consensus algorithm [61], manage the other nodes (called workers). All the requests to the Swarm (schedule containers, stop containers, gather node information and more) must go through a manager in order to maintain a consistent view of the cluster status.

This centralized approach of Swarm allows a perfect enforcement of system-level scheduling policies. Swarm provides a container consolidation policy (a.k.a Binpack) and a spread container policy (a.k.a Spread). In the perspective of the resource discovery and scheduling algorithms, a centralized node running them is called an oracle approach because it has a complete vision of the system state.

Due to this architecture, its scalability and fault tolerance are limited. Swarm creators added the possibility of having multiple manager nodes replicated in order to increase the fault resilience and availability of Swarm. In an Edge Cloud scenario there are higher latencies between nodes because they are geographically widespread. So, if the manager nodes are far apart the consensus algorithm will take much more time to converge. Since most of user's requests change the cluster status (e.g. deploy container and stop container, among others) the consensus algorithm will run many times since all this type of requests from all users trigger the consensus algorithm.

OpenStack and OpenNebula suffer from similar limitations due to a cluster like environment target. More-

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

over, none of these solutions are concerned with the system fairness since they assume a trustworthy environment where all the users respect each other and do no try to abuse from the system resources.

2.4.2 P2P Cloud System (Academic work)

Babaoglu et al. [19] present a prototype for a full P2P IaaS Cloud system. This early work targets the volunteer computing world (applied to Cloud Computing), trying to mitigate the problems of centralized solutions like Swarm.

Their prototype uses a gossip protocol (called Peer Sampling Service) to maintain all the nodes connected in a logical overlay without requiring any structure. With this protocol each node knows a sub-set of the system nodes, called local view. This local view of a node is constantly changing and being computed according with the last nodes that interacted with it. This makes it highly fault tolerant and peer churn resilient.

It has the drawback of allocating each set of nodes (called slices) only exclusively to a user. So, the system does not support multi-tenancy in the nodes, which is not good for extracting the maximum potential from the resources and, arguably, not truly cloud-like. In the presented prototype it was not possible to specify at least the characteristics of the wanted nodes during a request, the only option was random nodes.

Similar to most of the works in the Edge Computing and Fog Computing it does not present an evaluation section that clearly demonstrates the scalability properties of the resource discovery and scheduling with large volunteer networks like GUIFI.net ($\approx$ 35K nodes), or even beyond that to approximate BitTorrent network size. It also does not incorporate or specify a way to integrate system usage fairness to avoid abuses by the users.

2.5 Discussion

Most of the solutions, special the commercial ones (Swarm, OpenStack and more), are tied to centralized solutions (do not scale well) because they are much easier to build, maintain and extend, which makes sense because most of them were created by open source movements where an easy architecture to extend is much better. They also target a cluster like environment where machines are physically closer with good networks connecting, and composed by a set of homogeneous machines in terms of hardware capabilities and architectures. Which is not the case of the Edge Computing scenarios where the machines are physically widespread and are completely heterogeneous.

Academic works tend to use decentralized and distributed architectures, that scale far better than centralized ones. Most of the works are still primitive (e.g. Babaoglu et al. [19]) without fully working prototypes and with evaluations that do not evidence the system scalability with the size of real volunteer/community network. Also there are few works that consider the heterogeneity of the machines capabilities, which is common in the Edge Computing.

3 Caravela

Contents

In this chapter, we present Caravela**, a decentralized Docker's Container orchestrator** to be used in a Edge and Volunteer Computing scenario. After the analysis of the state-of-the art, presented in Chapter 2, we extracted many desirable properties for an Edge Cloud. We present Caravela's architecture with all the entities that interact in the system and the respective protocols that run on top of it. Finally, we present a classification of Caravela using our own taxonomy, presented in Chapter 2.

3.1 Desirable Properties

The size of Volunteer Networking and Computing platforms, like the GUIFI.net [62] community network, is constantly growing, e.g. GUIFI.net, at the time of writing, has around 35k active nodes and a steadily growth rate of 2k nodes/year since 2006. So Caravela must have a **scalable** design in order to handle a large number of nodes and user's requests. From here onward, a node is a device provided by the users that supplies its own resources, CPU, RAM, disk and network into Caravela, enriching its resource pool.

Caravela must have some degree of **usage flexibility** in terms of the resources a user can chose to deploy a Docker Container, e.g. **number of CPUs and RAM** necessary to run the container, these two resources are available in Docker Swarm for the user to specify. We introduce a **notion of CPU class** (not currently present in Docker Swarm) in a container deployment in order to give the user a better idea of the node's performance. In Docker Swarm, the user can't specify anything about the CPU speed because it targets homogeneous clusters with identical CPUs, we could not accept that in an Edge Cloud scenario. This also allows to maximize the resources usage of the system and providing a **minimum degree of QoS**. Since Edge Clouds are usually inserted in Volunteer Computing environments, guarantee QoS is extremely difficult because there is little or no control over the physical devices as they are most of the time managed by the users that contribute with them.

Volunteer Computing devices normally have different types of hardware, in terms of CPU architectures, operating systems and hardware capabilities. We don't try to solve the problem of different CPU architectures and Operating Systems, but we try to deal with the **heterogeneity in terms of device capabilities**, e.g. slow vs fast CPU, number of cores and memory available.

Volunteer and Edge Computing nodes are susceptible to failures and even unexpected downtime because the user that donates the resources may take them away. Therefore it is fundamental for Caravela to have some degree of **churn resilience**: it must adapt and degrade its performance gracefully during churn.

We want to maintain a fair level of **compatibility with Docker/Swarm** environments, thus we want to provide a similar CLI tool with identical commands semantics in order to easily move from one into the other.

Some of the properties mentioned here tackle a subset of the challenges referred in Section 1.3. We do not tackle all of them due to being outside the scope of the work and to time constraints. Most of the left out properties are orthogonal to the ones tackled by our work and can be seen in the Section 6.1, where we describe future work, e.g. the usage fairness is very important to a multi-user system like an Edge Cloud but

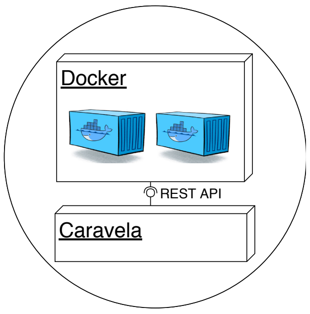


Figure 3.1: Node's mandatory components

we left it as an extension to our prototype and work.

3.2 Prerequisites

Caravela is a Docker's Container orchestrator, so it is **mandatory** that each node that joins Caravela has (see Figure 3.1):

- A standard and unmodified **Docker engine** configured and running in order to run the containers.
- Our Caravela's **middleware** running as a daemon in order to receive the user's requests.
- To simplify the prototype development we assume that each node has a **public IP address** or all of the nodes reside inside the same private network with a unique IP address. We leave as future work the development of a extension where nodes can be behind Network Address Translations (NATs) and Firewalls.

The following prerequisites are not mandatory to demonstrate Caravela's functionality but they are mandatory for real life deployment of Caravela as an Edge Cloud (they are out of scope for our work, check future work Section 6.1):

- A client for a **highly distributed file system** (e.g. InterPlanetary File System (IPFS)[1] [63] or BitTorrent[2] [64]) in order to upload, download and persist container's images in a distributed and decentralized way.
- A client for a distributed and decentralized **virtual currency system**, like Bitcoin [57], in order to manage a virtual currency. This client would help tackle some of the problems studied in Section 2.3, like the double spending attack.
- A client for a distributed and decentralized **reputation system**, like Karma [59], that maintains the users' reputation, e.g. it could be used to avoid accepting requests from users that after receiving the currency, do not provide the promised resources. This client would complement the virtual currency client in order to tackle the rest of the problems studied in Section 2.3.

[1] https://ipfs.io/
[2] http://www.bittorrent.com

3.3 Caravela Operations

A Caravela's user contributes to it providing devices into the system in order to enrich the global resource pool, as a benefit the user can also use the resources from others. The user can send **three main requests** to Caravela:

- **Deploy** - Deploy a Docker container or a semantically linked set of Docker containers into Caravela. The user can request an amount of CPUs/Cores and RAM for a container to run. We also let the user choose the class of the CPU where the container will be deployed, which is highly influenced by the CPU's speed. As we will show we will use a simple binary system for CPU class: class 0 are low tier CPUs and class 1 are higher tier CPUs. The user has the possibility to specify a scheduling policy per request, e.g. the user can request that two containers must run in the same node (**co-location**) or that the two containers must run in different nodes (**spread**). This scheduling policy per request is not present in the current Docker Swarm.
- **Stop** - Stops a container or a set of containers, releasing its resources to be reused.
- **List** - List all the containers the user has running in Caravela and its respective details.

3.4 Distributed Architecture

Caravela sits on top of a Chord's [65] ring overlay which implements a DHT. We chose Chord because it implements a lookup protocol that scales logarithmically, in terms of network hops, with the number of nodes participating in it, which we will leverage in order to efficiently discover the nodes that have the resources necessary to deploy a container.

Another interesting property of the Chord is its churn resilience. Authors claim that Chord can solve correctly all the lookup queries even in the presence of continuously peer churn but with a price in the lookup performance. All of the nodes have the same (or very similar) responsibilities participating in Caravela (see Section 3.5.2) so it is a full P2P system where all nodes are equal, which is interesting to enhance fault tolerance and resilience. Chord is a perfect fit for a loosely coupled peer system such as an Edge Cloud. The relationship among Caravela's nodes, devices contributing to Caravela and Chord's nodes is one-to-one. A picture of the overall distributed architecture is represented in Figure 3.2.

3.4.1 Network Management

Before explaining Caravela's algorithms we explain briefly how Chord works because it is the foundation for Caravela's distributed architecture.

Chord is typically used as a DHT for storing (put) and retrieving (get) content based on a key, e.g. storing and retrieving movie files based on its names. Chord maps the content identifier in a k-bit key, and then

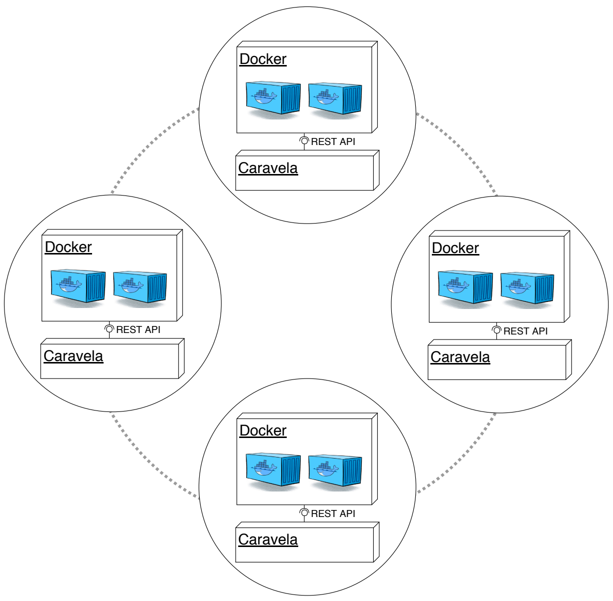


Figure 3.2: Caravela's distributed architecture.

maps the key in a node's overlay of size N which is normally much smaller than the key space. Each Chord node has a Global Unique Identifier (GUID) that belongs to the key space. When a new node joins Chord, it is assigned a random GUID that belongs to the key space, usually the GUID is obtained hashing the IP address using a consistent hashing algorithm [66] like SHA-1. The join protocol, to enter the Chord overlay, is detailed in Section 3.5.1. The consistent hashing is important, as we will show next, because it distributes the node's GUID uniformly in the ID/key space. From now on the node's ID space and the key space are used interchangeably. Each node becomes responsible for a set of keys. With the use of consistent hashing to map the content identifier to the key space, each node is responsible for roughly $\frac{2^k}{n}$ keys since the nodes are also uniformly distributed in the id space. This uniform distribution of the amount of keys per nodes grants two highly relevant properties: (a) decentralized load balancing, which is important in order to distribute the load of the searches by all nodes; (b) small amount of relocated keys when a node leaves. Finally Chord maps a key to the node that has the nearest higher GUID of the key. The great thing about Chord is the scalable and efficient routing mechanism that given a random key to a random Chord's node, it will find the node responsible for the key in average $log_2(N)$ network hops, with each node only knowing $log_2(N)$ nodes (a.k.a Chord's Finger Table).

In $\mathrm{Caravela}$ we want to find the resources to deploy a container, with each node providing the #CPUs of one CPU Class and an amount of RAM for $\mathrm{Caravela}$ via the Docker engine. We will use Chord to find a node that has sufficient available resources to deploy a container. Note that the **node's resources availability is dynamic** and changes over time when: the node receives a request to launch a container (#CPUs and RAM are used by the container) and when a container running in it stops (#CPUs and RAM used by the container are released). We will map the following resources in a single Chord's ring: CPU Class, #CPUs/cores free in the node and the amount of free RAM in the node, therefore for us the content's key is the pair $<< CPUClass; \#CPUs >$

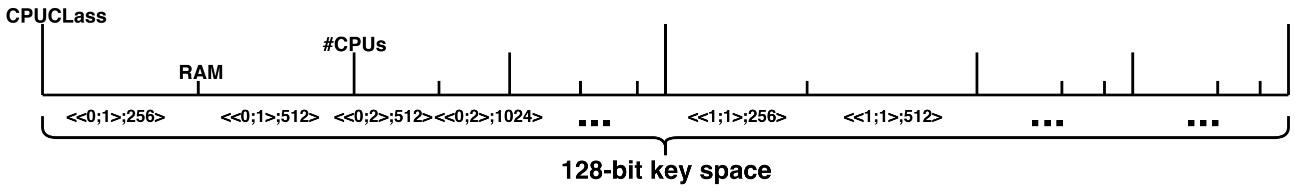


Figure 3.3: Example of the resources mapping, $<< CPUClass; \#CPUs >; RAM >$, into node's GUID/key space.

$; RAM >$ and the content is the IP address of the node that can serve this request.

The typical use of Chord is a equality based search, it uses consistent hashing of the content's identifier to find the node that holds that particular content. This is unsuitable for Caravela because the node's available resources and the necessary resources to deploy a container can't be matched with an equality based search, because that would reduce the chances of finding a suitable node, in order to find a node with exactly the same resources available as the request needs. Two similar requests, e.g. $<< 0; 1CPU >; 200MB >$ and $<< 0; 1CPU >; 300MB >$, would be mapped to completely different nodes. Basically we need to perform a kind of range-query. So instead of using consistent hashing we use a kind of **locality-preserving hashing** [67] in order to have nodes that can serve similar requests (in terms of resources needs) becoming neighbor nodes in the Chord's ring.

We decided to divide the ID/key space in contiguous regions of IDs where each one maps to a specific value of resources, an example of a division for the key space (considering only the #CPUs) would be four regions that would represent, 1CPU, 2CPUs, 4CPUs and 8CPUs. This, way nodes that have at least 4 CPUs available can register its resources in the 4CPUs region and in the ones below because it can satisfy the requests for 1, 2, 3 and 4 CPUs. As we will discuss later (see Section 3.5.3) a node could register its resources availability exclusively in the highest region it can (in terms of resources), but that did not prove to be a good solution. We need to map two resources #CPUs and RAM into the flat key space used by Chord. We also need to map the CPU class in the key space. So we used a **hierarchical division of the id space** to map the resources, see Figure 3.3.

Summing up each node has a GUID that represents a unique combination of CPU class, #CPUs and RAM, that represent the resources we want to find during a user deployment request. But each unique resource combination can be represented by multiple nodes. In Section 3.4.2, we detail why we need such region divisions.

3.4.2 Resources Mapping in Chord's Ring

As detailed previously, the ID space is split in several regions, in order to have nodes that can serve a certain kind of request, in a contiguous region of the Chord's ring. As we can see in Figure 3.3 the regions are not uniformly in terms of size. A graphical visualization of the resources division by regions can be seen in Figure 3.4.

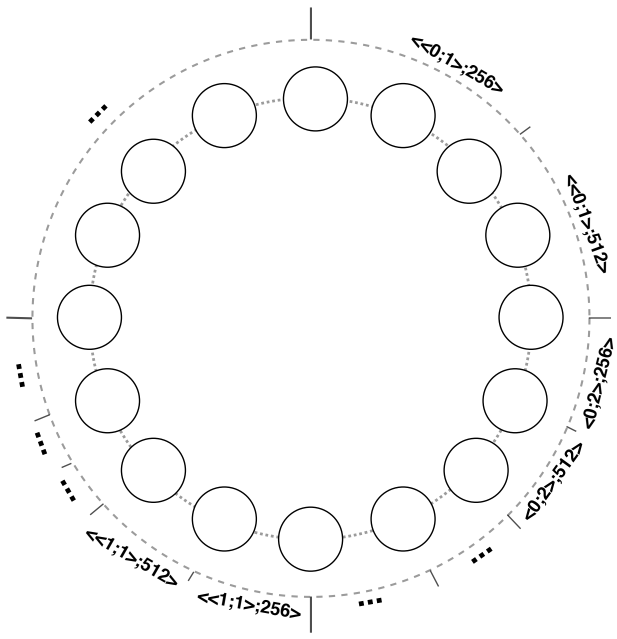


Figure 3.4: Resources regions mapped in Chord's ring.

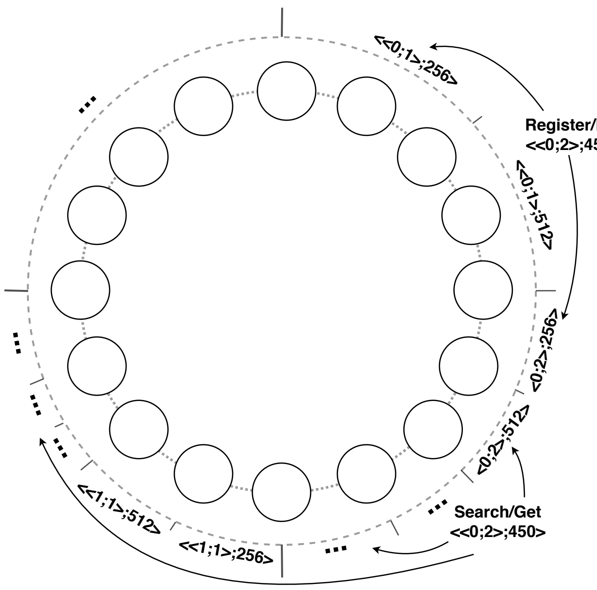


Figure 3.5: Examples of put and get operations over Chord's ring with the regions of resources mapped in it.

The rationale behind the regions configuration presented is straightforward. We want to have more nodes handling resources combinations that can be offered by the great majority of nodes, and also have more nodes that can serve the hottest resources requested by the users. This is important in order to distribute the load of Chord's put/get operations among the system nodes. The regions division configuration is static and provided in Caravela's bootstrap, for simplicity.

We could envision a section division configuration that would be dynamically adjusted depending on the amount of nodes registered in the region, in order to have more nodes handling the resource combinations that had more nodes providing it at the moment. As a downside, this dynamic reconfiguration would introduce a great overhead in a system with so many nodes as an Edge Cloud. Since the resources combination encodes the current available resources of a node and the nodes are constantly receiving deploy and stop requests, the dynamic reconfiguration would happen too often.

It is important to notice that we don't have all the possible combinations of resources mapped/encoded in the regions. The first reason why we have a smaller number of regions is due to scalability, as we present in Section 3.5.4.2. Finding resources are much more inefficient, in terms of performance, if we have too many regions. In the end, we mitigate that problem at the cost of a bit of fragmentation in node's used resources. But even if we had no problems in regions scalability, it would be impossible to cover all of the combinations of $<< CPUClass; \#CPUS >; RAM >$. Some combinations are also not useful such as, $<< 0; 4CPUs > ; 256MB >$. This combination is near impossible to happen so we would have nodes in this region that would never have node's registered offering that combination. This would hinder the load balancing in the system.

The Figure 3.5 shows two examples of a put operation where a node register its resource availability $<< 0; 2CPUs >; 450MB >$ in the Chord and and a get operation where someone was searching for a registered node with a resource availability of at least $<< 0; 2CPUs >; 450MB >$. As we can see in the figure the put/register operation can be sent into two regions, the ones that represent the same or less resources, or it could only be sent to the highest one of the two only (in our example), this is two different approaches into how we can use the resources regions. The get/search operation searches for $<< 0; 2CPUs >; 450MB >$, so it could be sent into any region that represents the same or a higher combinations of resources, here we have again several ways into how we can develop an algorithm to search for the resources necessary. The way we leverage these mapping of resources into the Chord's ring via the regions are explained in Section 3.5.4 where we describe our resource's discovery algorithms.

3.5 Algorithms

Now that we have introduced the node's network management using the Chord and the mapping of resource combinations into the node's ID/key space, we describe the Caravela's algorithms that use those mechanisms to drive container orchestration. We start by describing the **join algorithm**, then the **resource's discovery algorithms** and finally the **container's scheduling algorithms** that drives the user deployment requests. All the following protocols and distributed algorithms are built using a Representational State Transfer (REST) API. Each Caravela's node daemon runs a Web Server that handles the REST API requests; this is used by other nodes to consume the node's services.

3.5.1 Join Algorithm

The join algorithm consists on the use of **two** algorithms the Caravela's join algorithm and the Chord's join algorithm. The Caravela's algorithm at some point need to request Chord's overlay to join it in order to join the nodes' overlay becoming available to all other nodes via the overlay. The following sections detail these two algorithms and its interactions.

Algorithm 3.1: Caravelas's join algorithm.

```
Data: $remoteClient$
Function OnJoinCommand($caravelaKnownIP$):
    $caravelaConfigs \leftarrow remoteClient.ObtainConfigs(caravelaKnownIP)$
    $newNode \leftarrow NewNode(caravelaConfigs)$
    $newNode.Initialize()$
    $joinSuccess \leftarrow newNode.Chord.Join(caravelaKnownIP)$
    if $joinSuccess = true$ then
        $newNode.Start()$
        return
    else
        return $NewError$("ChordJoinError")
```

3.5.1.1 Caravela's Join Algorithm

The Algorithm 3.1 describes the full join algorithm. Notice that the objects that appear in the `Data` line of our pseudo-code examples consist in global objects of the node that is calling the function with the algorithm. When a node wants to join Caravela the user must send a **join command** to the node's Caravela daemon, providing the IP address of a node that already belongs to the Caravela. This command triggers the join algorithm. The node that is joining the system requests the system's configurations (line 2) of Caravela in order to initialize its structures (line 4) in compliance with all the other nodes that belong to Caravela. The `remoteClient` object consists in a gate to contact other Caravela's nodes via the REST API, it will appear in all the algorithms that we will show in the next sections. After that, the joining node provides to its Chord's client the IP of the known Caravela's node and the Chord's level join protocol starts (line 5). Once the node is already in the Chord's overlay the node starts the operation of its Caravela components (line 7), like the REST API Web Server, and becomes ready to receive requests.

This joining, protocol as all the other ones that follows it (presented in the following sections), does not need any central node for coordination, hence it increases the scalability and resilience of Caravela.

Since sometimes a user could not know any user with a node in Caravela we should provide an out-of-band mechanism to be able to join Caravela. We suggest that the Caravela's community should handle a web site to present a set of IPs of nodes that are already in Caravela. This facilitate the join of new users' nodes to Caravela.

3.5.1.2 Chord's Join Algorithm

The Chord's join algorithm is straightforward, its paper [65] can be consulted for details. The node that receives the Chord's join message generates a random GUID for the new node, and does a lookup using that GUID as the key, in order to find the place in the ring where the new node should be inserted. The node's GUID generation is done by applying a consisting hashing algorithm to the node's IP address producing a good random GUID. If the node that handles that key, known as the key successor, has a different GUID, then this means it has a higher GUID (due to Chord's lookup protocol). So the new node will be inserted as the predecessor of the key's successor and it will inherit the finger's table from its predecessor. The key successor becomes the new node's successor. If the key's successor has the GUID equal to the key, the key, hence the new node's GUID is incremented and the join protocol restarts.

3.5.2 Node's Roles

Before the description of the our algorithms that guide the container's deployment, we introduce the roles that a Caravela node has depending on the task/request it is processing. These roles are important to understand the next algorithms. The Figure 3.6 pictures the three roles that a node has while operating, Buyer,

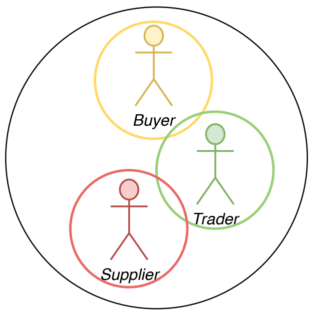


Figure 3.6: Caravela node's roles.

Supplier and Trader. From here onwards if we mention that a Buyer, Trader or Supplier sent or received a messages it is the same as saying it was a node, but operating in the context that the interest role was specified.

3.5.2.1 Buyer

The node acts as a Buyer when it receives a **deploy command** from the user specifying a container's deployment request. When acting as a buyer, the node's main goal is to find Caravela's node(s) with sufficient resources available to fulfill the user's container deployment request.

3.5.2.2 Supplier

The node acts as a Supplier when it is managing the Caravela's node own resources. The supplier responsibilities are to guarantee that the node's available resources are visible to other nodes in order to be used.

3.5.2.3 Trader

The Trader receives requests from suppliers with the resources being provided and receives requests from buyers that want to find nodes where to deploy the containers. In the end, its job is to match buyers and traders. The necessity of this level of indirection (via trader) is explained in section 3.5.4.

3.5.3 Data Structures

Before detailing the algorithms that make the resources discovery and scheduling phases of a deployment request, we introduce the main data structures (and its fields) we use:

- **Offer** contains the *available resources* of supplier, the *used resources* too, the *supplier's IP address* and an *unique local's offer ID*. The suppliers publish/put offers in the system and the buyer's search/get them to find a node to deploy a container.
- **Trader's Offers Map** is a map that each trader has where it register *supplier's offers* that put them available in the system.

- **Supplier's Offers Map** is a map that each supplier has where it saves the *offers it put available* in the system and the the *IP address of the trader* that is responsible for each one of them.
- **Regions Resource's Map** is used by traders and supplier to help them access the configurations of the chord regions pre-defined, e.g. obtain a random GUID in a certain resources region.
- **Container's Configuration** contain all the configurations necessary to deploy a container: *container's image key*, *container's name*, *arguments*, *port mapping* (HostPort:ContainerPort) and the container's *group policy*.
- **Container's Status** contains the *container's configurations*, the *IP of the supplier* that is running the container and the *container's local ID*.

3.5.4 Resource Discovery - Multiple Offers

Our resource discovering approach is called **multiple offers** (**multi-offer** to simplify) because each supplier will publish its available resources (via offers) into multiple traders across the ring overlay. We also developed another set of discovery algorithms, called *single-offer* (presented in the Appendix B) that utilize the Chord in a different way. The single offer algorithms had some advantages over the multiple offer but in the overall performance the multiple offer was the best one. We present the single offer algorithms in appendix in order to show that it is not the best choice for the resources discovering using Chord.

Algorithm 3.2: Supplier's create offer algorithm is called by a supplier to create/put an offer in the system.

```
Data: $resourcesRegionMapping, supplierOffersMap, remoteClient, chord, supplierIP$
Function CreateOffer($freeResources, usedResources, targetResourcesRegion$):
    $newOffer \leftarrow NewOffer(GenerateOfferLocalID(), freeResources, usedResources, supplierIP)$
    $destTraderGUID \leftarrow resourcesRegionMapping.RandomGUIDOffer(targetResourcesRegion)$
    $traderIP \leftarrow chord.Lookup(destTraderGUID)$
    $ok \leftarrow remoteClient.CreateOffer(traderIP, LocalNodeInfo(), newOffer)$
    if $ok = true$ then
        $newOffer.SetTraderIP(traderIP)$
        $supplierOffersMap[newOffer.ID] = newOffer$
        return
    return $NewError$("offerCouldNotBeCreatedError")
```

3.5.4.1 Supplying

The supplying is done by each node (as a supplier) when it needs to make its resources available to the other nodes. When a node decides to provide its resources it will encode (i.e. "hash") its free resources into a Chord's key in the following manner: look for the all the resources regions configured for the Caravela and find the **regions with resources combinations equal or smaller than the node's current available resources**. Then it calls function `CreateOffer` pictured in Algorithm 3.2, for each one of them with the targetRegionResources parameter being the region minimum handled resources (the ones specified in the region's label explained before). Algorithm 3.2 is in charge of creating an offer in the system. It starts with the creation of an offer object with its current available resources, its currently used resources, its IP and an local unique offer ID (line 2). After that, it generates a random GUID that belongs to the target resources region (line 3). In the end, our "hash" consists in selecting a random node inside a resources region, in order to distribute the offers of that region uniformly between the nodes. After this, we provide this key to Chord's lookup mechanism that will return us a node's IP address that handles the key (line 4). With this IP addresses we send directly to the node a `CreateOffer` message with our offer (line 5). The receiving node will handle the message as a trader. It will register the offer in a local's map called **trader's offers map**. When the Trader acknowledges the offer, the supplier also registers the offer in its local **supplier's offers map**, in order to know what it is offering, and what is the trader that is handling its offer (lines [7-8]).

When an offer is registered in a Trader it is available for all the nodes in the system. In Section 3.5.4.2 we explain how the other nodes can find the offers and use them. This was the description of how the supplying algorithm works when a supplier has no offers in the system because it recently joined it and it is the first time.

The complete supplying protocol is detailed in Algorithm 3.3. A Supplier runs it when:

1. The node joins Caravela and wants to provide its resources (simple case explained above).
2. The node launches a container in its Docker Engine consuming some of its resources, requiring the node/supplier to update its offers in order to match its new resource availability.
3. A container running in its Docker Engine stops and some of its resources are released, requiring the node/supplier to update its offers in order to match its new resources availability.

When the supplying algorithm is triggered due to the item (2) and (3) listed above, we need to execute another type of actions beyond the ones we explained before. When the algorithm is triggered due to item (2) the node sends an `UpdateOffer` messages to all the traders that maintain its offers (lines [5-8]). This update contains the new resource availability of the supplier's resources and it will be used by the scheduler to enforce global scheduling policies detailed in Section 3.5.5. It also sends asynchronous `RemoveOffer` messages to the traders that handle resources combinations that the supplier's currently free resources can't handle (lines [10-11]). When the supplying algorithm is triggered due to item (3) instead of sending the `RemoveOffer` message the node might send `CreateOffer` messages to traders in regions where the node did not have offers before (lines [12-13]). This happens due to the increase of resources available in the supplier. The `UpdateOffer` messages are also sent because the available resources in the node also changed. If an `UpdateOffer` message to a trader fails, the node will create a new offer for another trader in the former's trader region. It is important to note that all of these messages (except `CreateOffer`) mentioned are sent directly to the traders because the supplier saves the IP address of it in the first contact (usually after a Chord's get/lookup). With this, we avoid using Chord's lookup which is a costly operation ($log_2(N)$ with N being the network size). Summing up each supplier publishes one offer for each resources combination region where its free resources can handle the

Algorithm 3.3: Supplier's resources supplying complete algorithm.

```
Data: $suppliersOfferMap, resourcesRegionMapping, remoteClient$
1 Function SupplyResources($freeResources, usedResources$):
2     $regions \leftarrow resourcesRegionMapping.SuitableResourcesRegions(freeResources)$
3     foreach $offer$ in $suppliersOfferMap$ do
4         $offerRegion \leftarrow resourcesRegionMapping.Region(offer.TraderGUID)$
5         if $regions.Contains(offerRegion)$ then
6             $updatedOffer \leftarrow NewOffer(offer.ID, freeResources, usedResources)$
7             $remoteClient.UpdateOffer(offer.TraderIP, updatedOffer)$
8             $regions.Remove(offerRegion)$
9         else
10            $suppliersOfferMap.Remove(offer.ID)$
11            $remoteClient.RemoveOffer(offer.TraderIP, offer.ID)$
      /* It only creates offers in new regions, when its resources available increased to
      the point that it can now offer its resources in at least one higher resource
      region.                                                                        */
12    foreach $region$ in $regions$ do
          /* Algorithm 3.2 describes how an offer is created in the system.              */
13        $CreateOffer(freeResources, usedResources, region)$
```

requests that search on that region.

Since the nodes can fail due to the Edge Computing volatile environment, the trader could be publishing offers from dead suppliers. Conversely, a supplier could believe that an offer was published but its trader died. In order to diminish the unavailability windows we introduced a simple **refresh offer algorithm** to complement the supplying algorithm. With a refresh interval $Refresh_{interval}$, each trader will send a `RefreshOffer` message to the suppliers of the offers it has in its offer's table. With this, the supplier can acknowledge the presence of the trader and the trader also knows that the supplier is alive in order to provide the promised resources (like a lease renewal). This ensures liveness, garbage collects outdated information, while still ensuring completeness of accessibility to the resources in the long run. We also have a $MaxRefreshes_{failed}$ configuration parameter that is the maximum number of refreshes failed, for an offer, that a trader tolerates before removing it from its offer's table. The maximum refreshes missed $MaxRefreshes_{missed}$ is the complementary parameter and is used by the suppliers to figure out when the trader probably crashed. When $MaxRefreshes_{missed}$ is reached for an offer, the supplier will run the supplying protocol in order to put its resources online. In the case that the trader's crash is a transient failure, if someone tries to use that offer to claim the supplier resources, it will be denied by the supplier because it knows what are the active offers local IDs. If a trader tries to refresh an offer that is no longer valid, the supplier will respond with an error and the trader will remove the offer from its offer's table.

This refresh protocol alone introduced a lot of overhead regarding messages exchanged in the system, and the computational power expended to update the offer's refresh timestamps. So we decided to update the refresh timestamps and timeout counters in trader and supplier (i.e. renew these leases) piggybacked when `UpdateOffer` messages are exchanged between a supplier and a trader. Since each node is registered in

Parameter	Description
$Schedule_{policy}$	Schedule policy used by the scheduler component.
$ResourcesRegion_{mapping}$	The configuration for the mapping resources<->GUIDs (regions).
$Refresh_{interval}$	Refresh periodicity (from trader to supplier) to check if a offer is still valid.
$MaxRefreshes_{failed}$	Maximum failed refreshes to a supplier before removing the offer.
$MaxRefreshes_{missed}$	Maximum refreshes missed by trader before the supplier re-advertises.
$Offer_{timeout}$	Timeout for an offer being acknowledged by a trader.
$MaxRetries_{discovery}$	Maximum number of retries to the discovery strategies.
$Discovery_{strategy}$	Discovery strategy used by the nodes.
$SuperTraders_{factor}$	Factor of super traders to all nodes.

Table 3.1: Caravela's main configuration parameters.

multiple regions, it is difficult for a supplier to become completely unreachable due to all the traders' crashes. This way we can increase the refresh algorithm trigger interval (e.g. 15 minutes) to reduce the overhead of refreshing offers.

Table 3.1 contains the main parameters we enforce in Caravela's nodes. These configuration parameters are equal for all of the nodes and are used to initialize the boot node. All the nodes that join Caravela receive a copy of these configuration parameters, as we described in Caravela's join algorithm 3.5.1.

Algorithm 3.4: Resource's discovery algorithm.

```
Data: $resourcesRegionMapping, remoteClient, chord, configs$
1  Function DiscoverResources($resourcesNeeded$):
2      $retry \leftarrow 0$
3      while $retry < configs.MaxDiscoverRetries()$ do
4          $destTraderGUID \leftarrow resourcesRegionMapping.RandomGUIDDiscover(resourcesNeeded)$
5          $traderIP \leftarrow chord.Lookup(destTraderGUID)$
6          $resultOffers \leftarrow remoteClient.GetOffers(traderIP)$
7          if $resultOffers$ != $\varnothing$ then
8              return $resultOffers$
9          $retry \leftarrow retry + 1$
10     return $\varnothing$
```

3.5.4.2 Discover

Now that we explained how each node publishes/advertise its resources in multiple resources regions, via offers. The question left is, how can a *buyer* find a node to deploy a container?

Algorithm 3.4 describes how resource discovery is carried out. The buyer receives a request for a container deployment with the resources constraint for its deployment (line 1). The buyer node encodes (i.e. "hashes") the resources necessary, generating a random key, for the fittest region available (i.e. the region that represent the fewest resources combinations but that can still handle the ones required). Again, we try to distribute the discover requests load among all the region's nodes to not overly stress some nodes (line 4). With the key, we ask Chord to get the node's IP that handles the key (line 5). After we obtain the IP address we send a `GetOffers` message to the node (line 6). The trader will handle the message and return all the offers it has registered. If the trader returns a non empty set of offers, our discovery algorithm returns the offers to the caller, which is the buyer's scheduler algorithm. The selection of the offer to be used in the deployment is described in the scheduling Section 3.5.5.

We have a retry mechanism that is configured by the $MaxRetries_{Discovery}$ configuration parameter in order to allow us to try another random node in order to search for more resources (line 3). The use of retries has the cost of expending more messages, due to the an extra Chord's lookup, but increases efficacy (the possibility for finding the necessary resources). This is beneficial when we have few offers in the system (due to its high usage) in order to increase our chance of finding some.

When the system is low on resources available, we can fail to find the resources in a region even with some non empty traders. Our cost searching for resources now is bounded by the retries and with a very interesting performance in terms of requests served with success (see Section 5.6). Finding all the resources "crumbs" would cost us a great overhead in a completely decentralized system as ours.

3.5.5 Scheduling

Algorithm 3.5: Buyer's single container deployment algorithm.

```
Data: $scheduler$
1 Function OnDeploymentRequest($containerConfig$):
2     $resourcesNeeded \leftarrow containerConfig.Resources$
      /* Algorithm 3.6 describes how container(s) is/are scheduled into a node.    */
3     return $scheduler.Schedule(containerConfig, resourcesNeeded)$
```

Now that we presented our discovery strategy (multi-offer), which is the backbone of the system in terms of resource management, in this section we describe the scheduling algorithms that run during the scheduling phase. The scheduling phase conceptually comes after the discovery phase as we presented in Section 2.2 of the related work.

Scheduling is conducted by the buyer where the deployment request was injected. An user can inject a request in the node using a CLI application (see Section 4.3) that interacts with node's Caravela daemon. The buyer executes the `OnDeploymentRequest` function (see Algorithm 3.5) when receives the user's deploy request with the configurations for the container, it simply calls the `Schedule` function with the necessary resources for the container (lines [2-3]). This is the pseudocode to deploy **only one container per request**, later we describe an improved version of this algorithm to deploy a set of containers at the same time, called an application stack deployment (see Section 3.5.5.1).

Algorithm 3.6 describes the `Schedule` function, that given a container and its resources needs it schedules the container to run on a system's node/supplier. It starts by calling the `DiscoverResources` function from the

Algorithm 3.6: Algorithm to schedule containers in a Caravela's node.

```
  Data: $discoveryStrategy, schedulingPolicy, remoteClient$
1 Function Schedule($containersConfigs, resourcesNeeded$):
      /* Discovery strategy is backed up by our multi-offer discovery.  See Alg. 3.4.   */
2     $offers \leftarrow discoveryStrategy.DiscoverResources(resourcesNeeded)$
3     $offers \leftarrow schedulingPolicy.Rank(offers)$
4     foreach $offer$ in $offers$ do
5         $containersStatus \leftarrow$
            $remoteClient.LaunchContainer(offer.SupplierIP, offer.ID, containersConfigs)$
6         if $containersStatus$ != $\varnothing$ then
7             return $containersStatus$
8     return $NewError$("CouldNotScheduleContainersError")
```

discovery strategy (multi-offer), it returns a set of suitable offers from nodes that can handle the request (line 2). With this set of offers the buyer only needs to choose one to use according to a predefined system's global policy. This policy is based on Caravela's configuration parameter $Schedule_{policy}$. We have two policies, as Docker Swarm, **binpack** and **spread**, but applied in our decentralized architecture, which is more challenging to implement since the knowledge about the node's state is distributed in order to scale.

The **binpack** policy tries to consolidate the containers into the minimum number of nodes. It tries to find the node that has the minimum amount of resources available, while satisfying the user's request resources constraints. This policy tries to minimize the fragmentation of node's available resources, at the cost of having less load balancing in terms of nodes resource consumption.

The **spread** policy tries to spread the containers over all the nodes. Basically, it tries to balance the load across all the nodes trying to make the load, in terms of resource consumption, mostly equivalent in all of them. This policy distributes the load at the cost of fragmenting the nodes' available resources.

Resuming the description of the `Schedule` function 3.6, the scheduling policy ranks the offers, according to the policy, from the best to the worst (line 3). After that, the buyer iterates over the offers (line 4) and sends a `LaunchContainer` message with the container configurations (`imageKey`, `containersName`, `portMapping`, `resourcesNecessary`, etc) and the offer that will be used (line 5). If all the suppliers from all the offers are not able to launch the container, e.g. due to concurrency between buyers targeting the same trader and offers, the schedule function returns an error that bubbles up to the user saying the deployment was not possible. It then could repeated manually by the user.

When a supplier accepts a container to be launched, it uses the `imageKey` field of the container configuration structure to find the container's image in the images storage backend and starts downloading it in order to put the container running. Then a supplier acknowledges the launch of the container, returning all the information about the deployed container (`containersStatus` contains the `suppliersIPAddress`, `containersLocalID` and `containersConfigs`) that is saved in the buyer's in order to be consulted later by the node's user (using the **list command**) presented in Section 3.3.

The algorithms that rank the offers according to the schedule policy configured are presented in the Appendix C because they were not developed by us. We used the node ranking algorithms of Docker Swarm for the spread and binpack policy adapted to our offer data structure. We only adapted them to support the CPU class attribute that does not exist in Swarm.

These policies are enforced at the system level but, since our buyer only has access to a small sub-set of the suitable nodes/offers (the ones that were registered in the random trader we visited) for the deployment, its decision is not optimal and will not lead to a perfect global policy enforcement.

3.5.5.1 Containers Stack Deployment and Scheduling

All of our previous examples were based on an user's request to deploy one container at time, i.e. 1 request = 1 container. Nowadays with multi-tier distributed applications and micro-services architectures being the norm, most of container orchestrators, like Docker Swarm, offer the possibility to deploy a distributed application that has components running in different containers but that cooperate between them. This means that a user can make a single request specifying multiple containers to run. In Docker Swarm this is called a **stack deployment** and from here onward we also use this terminology. We also support this in Caravela, an user can build a request with multiple containers that must be deployed. In a stack deployment, if at least one container cannot be deployed, automatic `StopContainer` messages are launched to stop the ones that were already launched.

We went further and introduced the notion of group scheduling/placement policy at the request level. This is novel in Caravela; it does not exist in Docker Swarm. These policies are completely orthogonal to the global scheduling policies presented in Section 3.5.5, e.g. the global scheduling policy can be bin-packing/consolidating the containers while the user ask for all the containers to be spread in a stack deployment and vice-versa. We have two request level scheduling policies: **co-location** and **spread**.

The **co-location** policy can be specified in a stack deployment to ensure that a sub-set of the containers must be deployed in the same node. This is useful for containers that have latency sensitive interactions between them.

The **spread** policy is used by the user to specify that a subset of containers must be deployed in different nodes. This is useful for specifying robustness properties of the deployment, e.g. tolerate nodes failures.

Figure 3.7 contains an example of a stack deployment file where the user specifies that the two containers must be deployed in the same node using the group policy parameter. The files we use in Caravela have identical syntax and semantics to the ones used to deploy application stacks in Docker Swarm. We ignore some parameters that Docker Swarm accepts, since we don't use them in our prototype, and we added some that did not exist in Docker Swarm like the containers group policy.

Algorithm 3.7 is the extension of the Algorithm 3.5 already presented, it describes how a buyer deploys a container's stack (it can also be used to deploy only one container too). The buyer starts by dividing the containers that must be spread and the ones that must be co-located (lines [5-10]), it also aggregates the resources necessary for all the co-located containers (line 7). After that it provides the co-located containers

```yaml
database:
  group_policy: "co-location"
  image: "redis:alpine"
  cpus: 1
  memory: 1024
  ports:
  - "9000:6000"
web:
  group_policy: "co-location"
  image: "webserver"
  cpus: 2
  memory: 512
  ports:
  - "8080:80"
```

Figure 3.7: Example of a YAML stack deployment file.

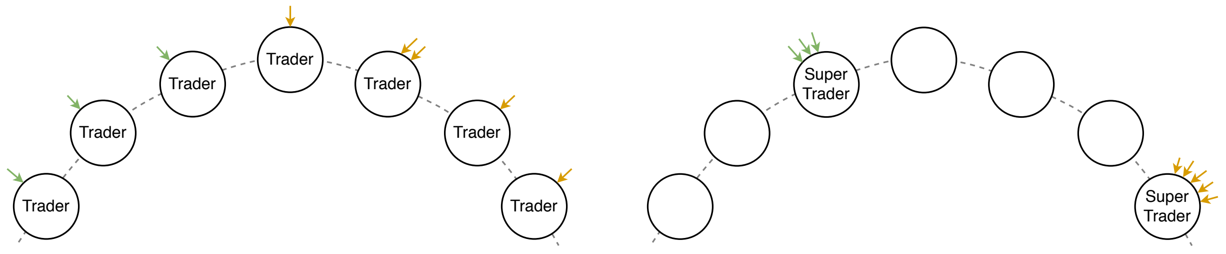
Algorithm 3.7: Buyer's extended deployment algorithm for: single container's deployment and container's stack deployment.

```
Data: $scheduler, remoteClient$
1  Function OnDeploymentRequest($containersConfigs$):
2      $deploymentFailed \leftarrow false$
3      $colocatedResourcesSum \leftarrow NewResources(0, 0)$
4      $colocatedContainers, spreadContainers, deployedContainersStatus \leftarrow \varnothing$
5      foreach $containerConfig$ in $containersConfigs$ do
6          if $containerConfig.GroupPolicy$ = "Co-location" then
7              $colocatedResourcesSum.Add(containerConfig.Resources)$
8              $colocatedContainers \leftarrow Append(colocatedContainers, containerConfig)$
9          else
10             $spreadContainers \leftarrow Append(spreadContainers, containerConfig)$

       /* Schedule the co-located containers in one node.  Skipped if there are not.      */
11     $containersStatus \leftarrow scheduler.Schedule(colocatedContainers, colocatedResourcesSum)$
12     if $containersStatus = \varnothing$ then
13         return $NewError$("DeployFailedError")
14     $deployedContainersStatus \leftarrow Append(deployedContainersStatus, containersStatus)$
       /* Schedule the remaining containers, spreading them over different nodes.  Skipped
       if there are not.                                                                  */
15     foreach $spreadContainerConfig$ in $spreadContainers$ do
16         $spreadContainerResources \leftarrow spreadContainerConfig.Resources$
17         $containerStatus \leftarrow scheduler.Schedule(spreadContainerConfig, spreadContainerResources)$
18         if $containerStatus = \varnothing$ then
19             $deploymentFailed \leftarrow true$
20             break
21         $deployedContainersStatus \leftarrow Append(deployedContainersStatus, containerStatus)$

       /* Rollback the deployment if any of the containers could not be deployed.         */
22     if $deploymentFailed = true$ then
23         foreach $deployedContainer$ in $deployedContainersStatus$ do
24             $remoteClient.StopContainer(deployedContainer.SupplierIP, deployedContainer.ContainerID)$
25         return $NewError$("DeployFailedError")
26     return $deployedContainersStatus$
```

configurations and its resources necessary for the `Schedule` function that will schedule them in a node (lines [11-14]). In the end, the system treats these co-located containers as a single larger container in terms of resources needs. When the user does not specify the group policy for a container, we assume by default it is spread. To enforce the spread policy we treat each container as an independent one-container request, so the buyer calls the `Schedule` function several times and, due to the nature of our random search in a region (always select a random trader), we will end up with a high probability of finding different offers/suppliers for the deployments (lines [15-21]). As we already explained, if one or more containers cannot be deployed/scheduled all the other ones are immediately stopped (lines [22-25]). Note that when we aggregate the resources for the co-located containers we assume the CPU class of the destination node to be the highest one specified in all of them.

3.5.6 Optimizations

In this section we describe some optimizations to the previously described algorithms.



(a) Without Super Traders. **(b)** With Super Traders.

Figure 3.8: Nodes of a single resource region receiving `CreateOffer` and `GetOffers` messages (represented by the arrows).

3.5.6.1 "Pseudo" Super Traders

When we select a random trader where we create or search for offers, we have the probability to find one with no offers or with very few; this is specially true when the system is low on free resources. This has two performance implications in our system: (1) if we do not find offers in the trader, our algorithm retries the search into other node, which is costly due to Chord's lookup usage (efficiency problem), or rejects the user's request (user satisfaction problem); (2) if the node has few offers, our global policy will suffer from efficacy problems because it is being applied to a very small subset of the total offers.

So, in order to counter these problems, we implemented a modified random trader selection. Instead of choosing a completely random key in the resources region ID/key space, we choose a random key, but from a small sub-set of random keys that are evenly distributed over the resources region key space. We try to consolidate the offers and the search for them in fewer traders in order to counter the problems previous explained. Now we are giving up full load balancing among all the traders: some will be responsible for more offers (Super Traders) than before, and others will have no offers to manage. This is a trade-off we must pay. We can control the consolidation factor of traders with the parameter $SuperTraders_{factor}$ (see Table 3.1), which defines the amount of traders that will now have its job done by one super trader. Figure 3.8 pictures the messages distribution by the nodes with and without the super trader optimization.

3.6 User's Commands

Here, we briefly explain how the user's stop container(s) and list container(s) commands work. When a deployment request is fullfilled the user's node where the he/she injected the request saves locally all the data about the deployed containers. It saves for each container: the configurations used to deploy it, the ID of the container in the supplier and the IP address of the supplier. A Caravela's deployed container is unequivocally identified by the pair $< supplierIP; containerID >$. When a **stop command** is issued by the user in the node (containing the ID/IDs) of the containers to stop, the node simply sends a `StopContainer` message directly to the supplier(s). When a **list command** is issued by the user in the node it returns the data saved in the node.

3.7 Survey and Discussion

To summarize the architecture of Caravela, we will assess it according to the taxonomies provided in Chapter 2. Since our work did not developed any fairness focused property due to the scope of the work, we do not have any classifications in that category.

3.7.1 Edge Cloud

Caravela's classification as an Edge Cloud based in Section 2.1 of the related work:

- **Resource Ownership:** Intended to be used as a completely volunteer Edge Cloud where each user donates its devices as nodes in Caravela.
- **Architecture:** It is clearly a Peer-to-peer (P2P) Edge Cloud due to the Chord and our algorithms.
- **Service Level:** Caravela is an IaaS (Container) Edge Cloud that deploys standard Docker containers.
- **Target Applications:** It is designed as a general purpose platform without any particular benefits for a specific kind of applications.

3.7.2 Resource Management

3.7.2.1 Resource Discovery

Caravela's discovery strategy classified according to Section 2.2.1 of the related work:

- **Architecture:** Our resource discovery strategy (multiple offer) is P2P structured since they are built on top of the structured Chord's overlay. This was intended to have an efficient overlay to search resources in large networks.
- **Resource Attribute:** The resource attributes specified in our containers deployment (CPUs and RAM) are dynamic, which means we look for the current availability of them in each node. The CPU class attribute is static, but since the user can specify the three of them in the same request, the classifications is dynamic.
- **Query:** The user can specify multiple resource attributes in its containers deployment, CPU class, CPUs and RAM.

3.7.2.2 Resource Scheduling

Caravela's scheduling classified according to Section 2.2.2 of the related work:

- **Architecture:** P2P due to the chord's overlay and our set of algorithms.
- **Decision:** The decisions of scheduling are "Ongoing" for each request that arrives the system tries to place it in the best node according to the current nodes state.
- **User Requirements:** The users can specify the CPU class, #CPUs and RAM for one container deployment. They can also specify request-level scheduling group policies, **co-location** and **spread**.
- **System Goals:** The system-level resource consolidation is achieved by our binpack policy and the resource distribution can be achieved with our spread policy.

3.8 Summary

We started this chapter by stating the desirable properties for our Caravela's prototype: scalability to handle large network, usage flexibility in terms of resources requested by the user, handle different capabilities of the nodes in terms of CPU speed, #CPUs and RAM available, churn resilient and finally providing a minimum degree of QoS when deploying containers. We also described how we did the network management using Chord as the base overlay for efficient and scalable searches. After that we described how we encoded the resources in the key space in order to know where we need to search when searching for resources to deploy a container. Then we detailed the completely decentralized and scalable algorithms for: joining Caravela, discovering resources (Multi-Offer) and scheduling containers.

To wrap up the Caravela's architecture description we assessed our prototype using the taxonomies presented in Chapter 2.

4 Implementation

Contents

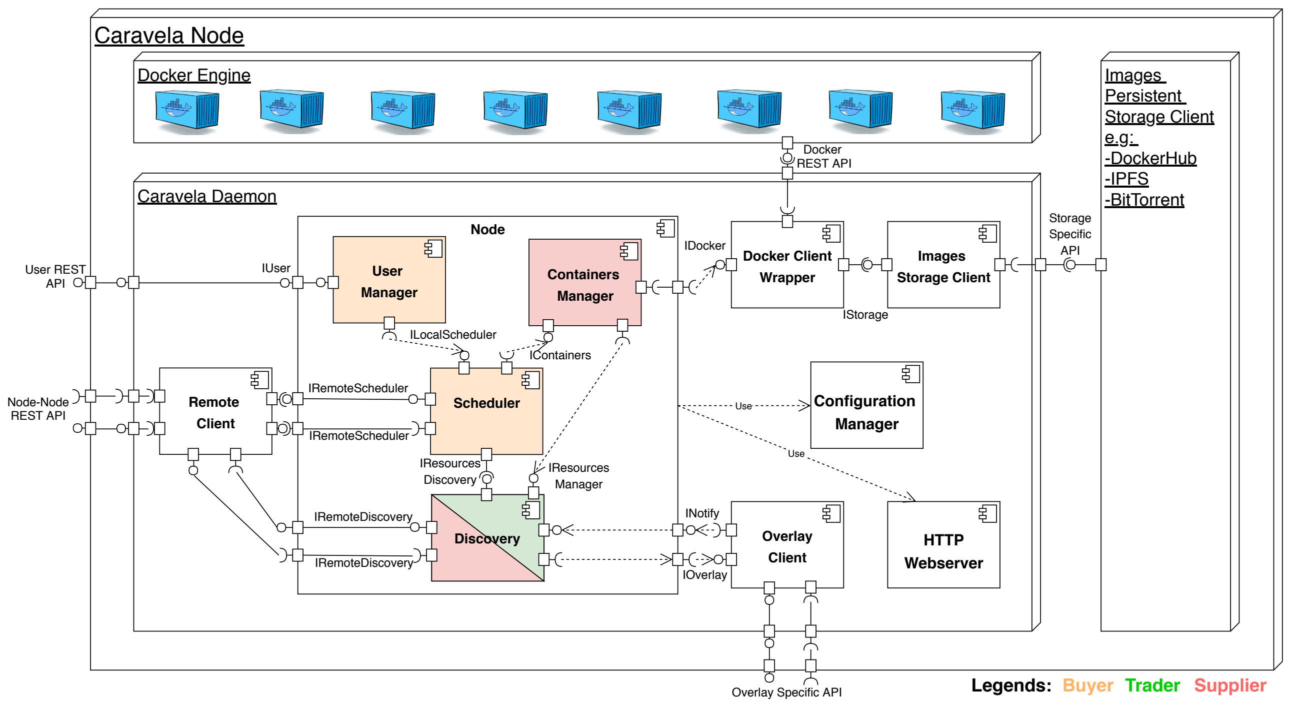


Figure 4.1: Caravela's components and interfaces relationships.

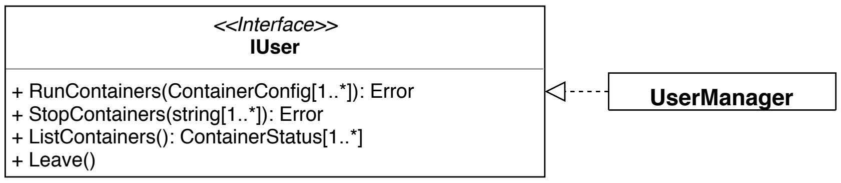
Our Caravela prototype is written in Go, it contains ≈7.5K of Go's Lines of Code (LoC). We decided to choose Go, because Docker, Swarm and all the solutions around containerization are written in Go, so we could leverage libraries that are used by these projects and more important the integration with them in the future. This is our ultimate and more ambitious goal (as we refer in the future work, in Section 6.1). In the following sections we describe the major details of Caravela's. We detail the CaravelaSim which is used to evaluate our prototype and we also describe the **CLI** tool developed to use Caravela.

4.1 Caravela Prototype

In this section we detail the major implementation details about Caravela's prototype.

4.1.1 Software Architecture

As already mentioned, Caravela is a P2P system, so each node contains the same components and runs the same code. Here, we address the inside of Caravela in order to understand what components are responsible for the algorithms and protocols described in Section 3.5. Caravela is composed by **eleven** components (totally developed by us): **Node**, **User Manager**, **Remote Client**, **Docker Client Wrapper**, **Scheduler**, **Discovery**, **Images Storage Client**, **Containers Manager**, **Overlay Client**, **Configuration's Manager** and the **HTTP WebServer**. Figure 4.1 pictures Caravela's node components, its interfaces and relationships. It also pictures the relations with the external components like the Docker Engine. This picture is used as reference



<<Interface>>
IUser
+ RunContainers(ContainerConfig[1..*]): Error
+ StopContainers(string[1..*]): Error
+ ListContainers(): ContainerStatus[1..*]
+ Leave()
UserManager

Figure 4.2: IUser interface exposed by the User Manager.

for the next sections, where we detail each one of the components explaining its main functionalities.

4.1.1.1 Node

The Node component is the super component that contains the main logic implementation of our algorithms. It exposes the public services of the User Manager (IUser interface), Scheduler (IRemoteScheduler interface) and Discovery (IRemoteDiscovery interface). These interfaces are exposed to other nodes via a REST API with the help of the HTTP WebServer component which we call Node-Node REST API. This Node-Node API contains all the services available for a node to contact another in order to cooperate to discover resources, schedule containers and more. As mentioned before, Caravela has a set of configuration parameters that all the nodes must agree on and are defined during the system bootstrapping. The Node component uses the Configuration Manager to obtain all the configuration parameters and passes them to its internal components in order to configure them accordingly.

4.1.1.2 User Manager

The User Manager component manages the user's information and requests. It exposes to the outside an interface called IUser that is used as the front-end of Caravela for a user. It handles the requests mentioned in Section 3.3. As already mentioned, it is exposed outside via REST services, that are consumed by our CLI tool presented Section 4.3. It validates the requests and forwards them to other components if necessary. It forwards container's deployment requests on behalf of the user into the Scheduler component. Figure 4.2 illustrates the IUser interface methods provided by User Manager. The UML diagram pictured in Figure 4.3 illustrates the main entities used across the system; it is here as a reference to all the following UML diagrams that we will show.

The User Manager could also be responsible for the user's authentication, currency and reputation control. Since the scope of the work would be large with all of these features, we left the extension of User Manager component to support them into future work as we mentioned before.

4.1.1.3 Scheduler

The Scheduler component drives the containers deployment. It exposes two interfaces: ILocalScheduler and IRemoteScheduler. The ILocalScheduler interface is used by the User Manager component to inject user

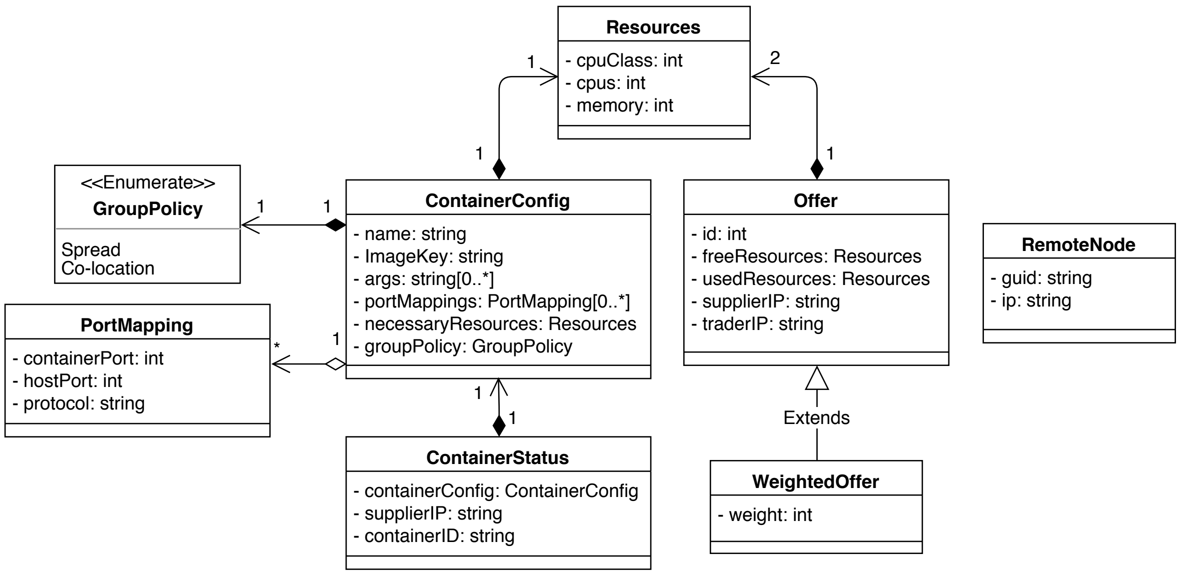


Figure 4.3: Common classes used across all the system components.

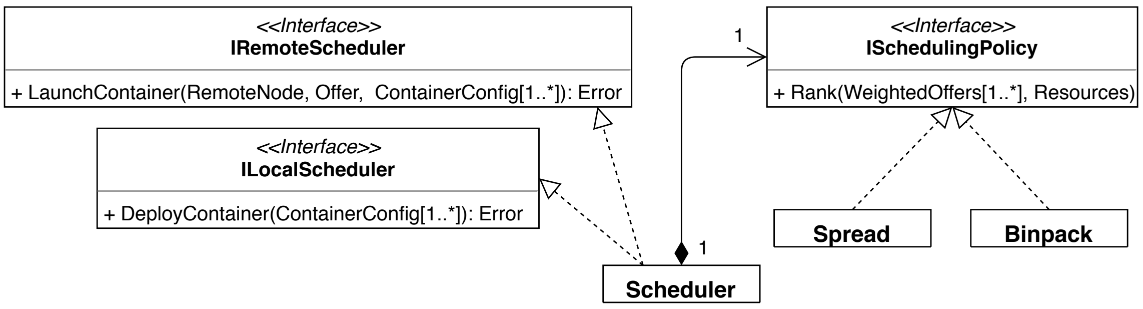

<<Interface>>
IRemoteScheduler
+ LaunchContainer(RemoteNode, Offer, ContainerConfig[1..*]): Error
<<Interface>>
ILocalScheduler
+ DeployContainer(ContainerConfig[1..*]): Error
1
<<Interface>>
ISchedulingPolicy
+ Rank(WeightedOffers[1..*], Resources)
Spread
Binpack
1
Scheduler

Figure 4.4: Scheduler component and its interfaces.

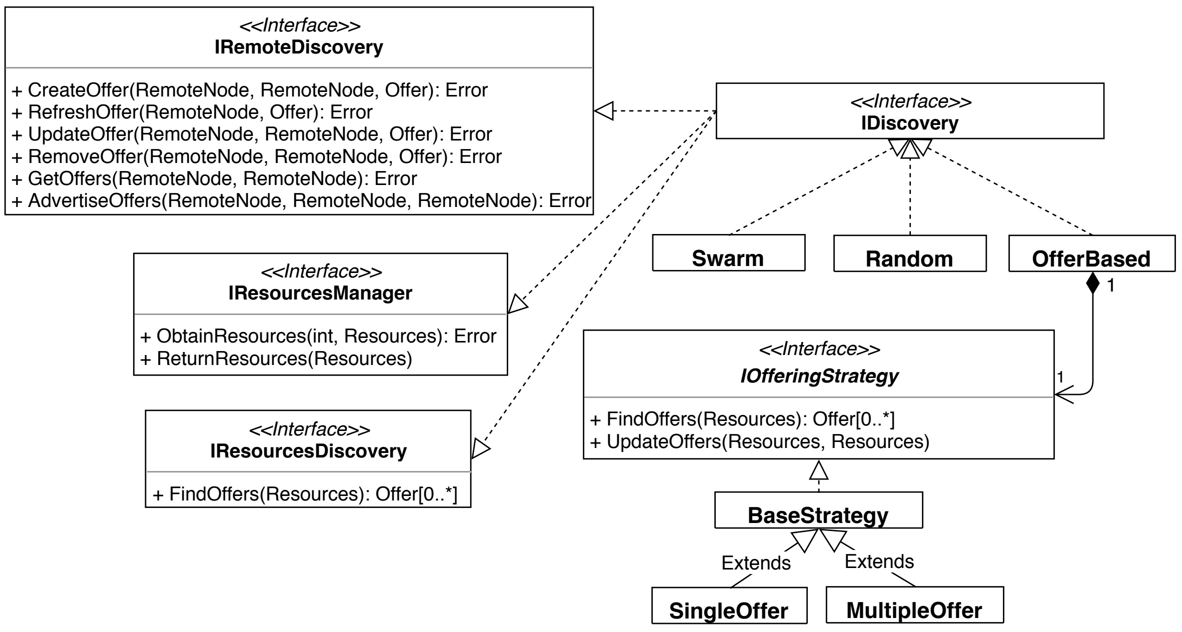

<<Interface>>
IRemoteDiscovery
+ CreateOffer(RemoteNode, RemoteNode, Offer): Error
+ RefreshOffer(RemoteNode, Offer): Error
+ UpdateOffer(RemoteNode, RemoteNode, Offer): Error
+ RemoveOffer(RemoteNode, RemoteNode, Offer): Error
+ GetOffers(RemoteNode, RemoteNode): Error
+ AdvertiseOffers(RemoteNode, RemoteNode, RemoteNode): Error
<<Interface>>
IDiscovery
Swarm
Random
OfferBased
1
<<Interface>>
IResourcesManager
+ ObtainResources(int, Resources): Error
+ ReturnResources(Resources)
<<Interface>>
IOfferingStrategy
1
+ FindOffers(Resources): Offer[0..*]
+ UpdateOffers(Resources, Resources)
<<Interface>>
IResourcesDiscovery
+ FindOffers(Resources): Offer[0..*]
BaseStrategy
Extends
Extends
SingleOffer
MultipleOffer

Figure 4.5: Discovery component interfaces and implemented backends.

requests in the system (as a local buyer). IRemoteScheduler is exposed to other nodes in order to allow them (as remote buyers) to send messages to launch containers in other nodes.

The Scheduler is responsible for choosing the best node(s) for the container(s)'s deployment. To achieve that, it uses the Discovery component (via IResourcesDiscovery interface) to find suitable nodes for the requests that the User Manager component forwards to it. The set of suitable nodes (via offers) that it receives depends on the Discovery component's implementation. Some of them are capable of analyzing all the system nodes to find the best suitable nodes; other are more restrictive hence scalable, e.g. our (Multi-Offer) algorithms can only find a small sub-set of all the suitable nodes. Our global policies (**binpack** and **spread**) for the scheduling algorithm are implemented in Scheduler using the scheduling policy object that implements the ISchedulingPolicy interface. As previously mentioned the algorithms that implement the binpack and spread policies are an adaptation from the Docker Swarm, and they are present in the Appendix C. The request-level policies (**co-location** and **spread**) are also implemented in the Scheduler, more concretely the deployment algorithm (Alg. 3.7) is implemented in the `DeployContainer` method of ILocalScheduler. Figure 4.4 illustrates the Scheduler and its interfaces.

4.1.1.4 Discovery

The Discovery component has three main responsibilities: (1) manage the node's local resources, as a supplier; (2) manage other node's offers, as a trader; (3) implement the resource discovery algorithms to efficiently find the resources. It relies on the Overlay client component (for us backed up by Chord) to guide its search for resources over the nodes' overlay. We implemented different backends of the Discovery component: Docker Swarm based, Random, and Offer based. Figure 4.5 pictures the discovery backends and

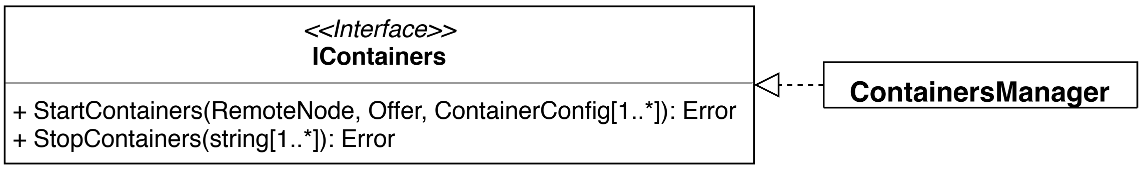

<<Interface>>
IContainers
+ StartContainers(RemoteNode, Offer, ContainerConfig[1..*]): Error
+ StopContainers(string[1..*]): Error
ContainersManager

Figure 4.6: IContainersManager interface exposed by the Containers Manager.

the Discovery component interfaces. The Swarm and Random backend are used in Section 5 to compare with our Multi-Offer backend.

The IRemoteDiscovery interface is provided for other nodes to contact the node's Trader in order to manage those offers depending on its resources region in the overlay (defined by its GUID).

The Discovery component must control the local node's available resources, exposing them to other nodes via the offer system (in the case of our Multi-Offer backend). It implements the supplying algorithm (Alg. 3.3) that we presented before. The IResourcesManager interface is exposed by the Discovery component to the Containers Manager because the Discovery component must verify whether the offers used by container's manager to deploy containers in the local node are valid.

The ILocalDiscovery interface is used by the Scheduler component of the same node to discover the resources it needs to deploy containers. The resource discover algorithm (Alg. 3.4) that we presented implements IResourcesDiscovery interface. The resource discovery algorithm could be different, depending on the Discovery backend we use. The Random discovery backend implements another algorithm to find the resources. This way we can easily change the discovery backend and experiment with different variations to evaluate them as we will see in Section 5.6.

This Discovery component provides a composition of interfaces that we call IDiscovery, it consists in the union of the interfaces: IRemoteDiscovery, IResourcesDiscovery and IResourcesManager. The methods of IResourcesDiscovery and IResourcesManager must be implemented by any Discovery backend, because the Scheduler and Containers manager rely on them. The IRemoteDiscovery methods are the ones exposed to other nodes, so some Discovery backends can use all of them, while others are simple enough that do not need all of them, e.g. the Random backend does not create offers nor use them in its operation, so all the methods of format `*Offer` are not implemented by it.

4.1.1.5 Containers Manager

The Containers Manager component manages the containers that run in the local's node Docker engine. It exposes the IContainers interface for the Scheduler component, see Figure 4.6. This interface is used by the Scheduler component to launch a container in the node on behalf of other nodes. The Containers Manager use the IResourcesManager interface in order to validate (with the local supplier) that the offer used to deploy the container is valid. If it is valid, the Containers Manager claims its resources (method `ObtainResources`), and the Discovery Component needs to update its state because the node's resource availability has changed.

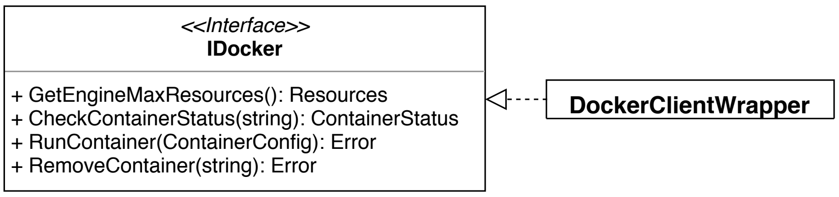

<<Interface>>
IDocker
+ GetEngineMaxResources(): Resources
+ CheckContainerStatus(string): ContainerStatus
+ RunContainer(ContainerConfig): Error
+ RemoveContainer(string): Error
DockerClientWrapper

Figure 4.7: IDocker interface exposed by the Docker Client Wrapper.

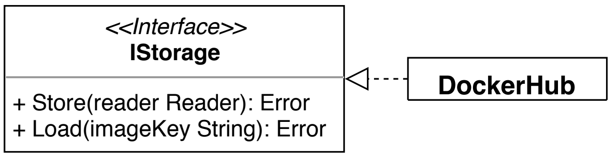


Figure 4.8: IStorage interface exposed by the Image Storage Client and our DockerHub backend.

The IDocker interface is used by the Containers Manager to issue commands to the Docker engine like: run container, stop container and even listening for containers events. If it receives an event from the Docker engine telling that a container (previous launched) crashed or exited somehow, it immediately informs the Discovery component via the `ReturnResources` method of IResourcesManager informing the supplier that the resources claimed by the container can be re-utilized.

4.1.1.6 Docker Client Wrapper

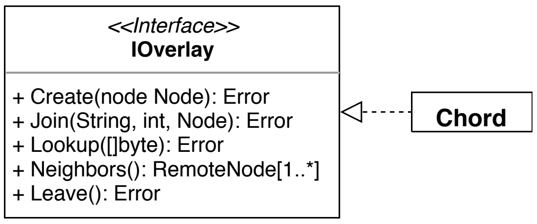
The Docker Client Wrapper, as the name indicates, wraps the Go's Software Development Kit (SDK) for the Docker Engine exposing a very simple interface to be used by our Containers Manager component. We did this to isolate the containers management at Caravela's level from the Docker's API details.

When issuing a command to the Docker engine, to launch a container, it specifies in the request the resource constraints that the user specified. The Docker Engine can enforce the maximum utilization of CPU and RAM per container. This is very helpful to provide good QoS for the containers scheduled in the Docker engine. It avoids a container from abusing the resources requested for it.

Docker Wrapper is also responsible for providing the information about the maximum resources available ($< CPUs; RAM >$) for the Docker Engine, which will be the maximum resources we consider for a node. It also provides the $CPUClass$ using a simple string matching algorithm against the CPU information provided by the Go's runtime. We consider that all x86 Intel CPUs with clock greater than 2GHz are from class 1 and all the others are from class 0. This could be enhanced running the linpack [68] benchmark during some seconds in order to classify the CPU speed more accurately.

4.1.1.7 Images Storage Client

The Images Storage Client is a component that encapsulates the details of dealing with different storage backends exposing an independent interface called IStorage. This interface is used by the Docker Client Wrapper to load images into the Docker engine (using the image's key passed in the container's configuration)



<<Interface>>
IOverlay
+ Create(node Node): Error
+ Join(String, int, Node): Error
+ Lookup([]byte): Error
+ Neighbors(): RemoteNode[1..*]
+ Leave(): Error
Chord

Figure 4.9: IOverlay interface exposed by the Overlay client.

Parameter	Description
$\#CPU_{overcommit}$	Percentage of overcommit over the #CPUs available.
$RAM_{overcommit}$	Percentage of overcommit over the RAM available.
CPU_{slices}	#Slices of a single CPU.
$Storage_{backend}$	Image's storage back end used.

Table 4.1: Caravela's additional configuration parameters. Complements Table 3.1

when it needs to deploy a container. It is also used to submit an image into the storage. We currently only have the DockerHub implementation that shares the images via a public centralized repository of images, but other backends could be easily developed and plugged in, e.g. BitTorrent and IPFS.

4.1.1.8 Overlay Client

The Overlay Client is also a component that encapsulates the details of an overlay exposing an interface independent of the implementation called IOverlay. Our current implementation is the Chord overlay (using an open source implementation for Go), but there are more implementations for overlays with similar DHTs interfaces e.g. Pastry [69], Tapestry [70] and Kademlia [71]. It would be easily to integrate them in order to test other possibilities.

4.1.1.9 Configuration Manager

The Configuration Manager component reads a Caravela's configuration file providing them to the Node component in order to configure the node. The main configuration parameters that affect our algorithms were already presented in Table 3.1, but there are more that we did not discussed yet, they are presented in Table 4.1.

The $\#CPU_{overcommit}$ and $RAM_{overcommit}$ parameter specify a percentage of overcommit that the Caravela enforces in the number of CPUs and in the RAM available from the node, e.g. if a user specifies 150% for $\#CPU_{overcommit}$ it means that if a node normally provides 2CPUs it will now provide 3CPUs this is usually used due to the knowledge that even the resources requested for a container are not fully utilized. Docker Swarm provides only one general resource overcommit parameter that applies to all the resources #CPUs and RAM, we are more flexible and allow different degrees of overcommit for different resources.

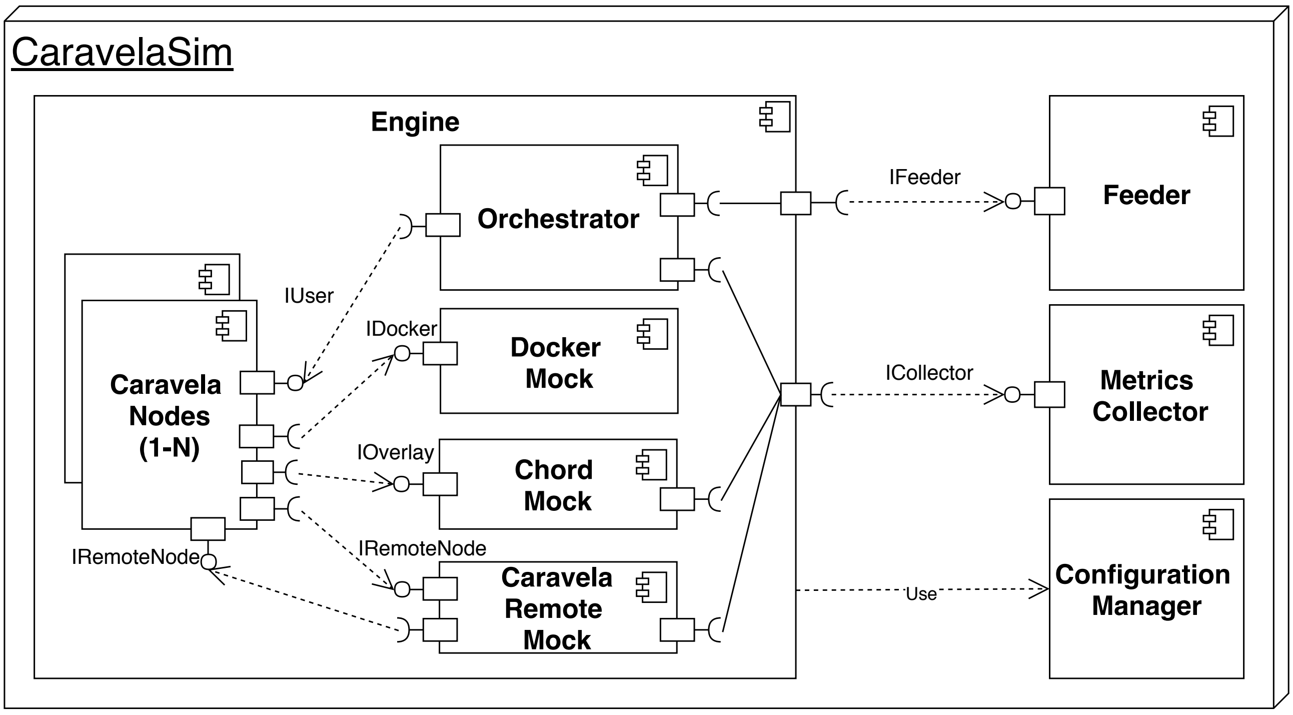


Figure 4.10: CaravelaSim component's diagram.

The CPU_{slices} is a simple way of configure Caravela to unfold 1CPU in multiple division e.g. if we use 2 for CPU_{slices} this means that the system now offer multiples of 0.5CPU, if we use 1 the multiples are 1CPU as we have been explaining throughout the document to simplify. The $Storage_{backend}$ is used to configure what is the Images Storage backend to be used in Caravela.

An example of a complete Tom's Obvious, Minimal Language (TOML) configuration file can be seen in Appendix A, it contains even more parameters that we did not mention due to space limitations, like the port where Caravela listen and other debug and auxiliary parameters.

4.1.1.10 HTTP WebServer

The HTTP WebServer Component is used to handle the Caravela's REST endpoints and forwarding them to the respective components.

4.2 CaravelaSim - Caravela's Simulator

To evaluate our Caravela's prototype with a large number of participating nodes, we also developed a cycle-based simulator called CaravelaSim[1]. It has ≈2.5k Go's LoCs and provides a CLI interface to interact with it. We use it as a testbed and debug helper for our Caravela's prototype. We decided to build the simulator due to the necessity to verify our prototype scalability properties. To do that, we needed a large amount of devices

[1]Code available at https://github.com/Strabox/caravela-sim

Parameter	Description
$Number_{nodes}$	Number of nodes participating in Caravela.
$Tick_{interval}$	The time duration for each tick.
$Maximum_{ticks}$	The maximum number of ticks for the simulation.
$Discovery_{backends}$	Which Caravela discovery backends should be tested.
$Request_{feeder}$	Which request feeder is used to feed the simulation.
$Requests_{rates}$	Amount of deploy/stop containers requests during the simulation duration.
$Requests_{profiles}$	Profile of the injected requests, in terms of resources needed.
$Multithreaded$	Used to take advantage or not from multi-core processors.

Table 4.2: CaravelaSim's main configuration parameters.

running a Docker Engine which was not foreseeable due to time/budget constraints. We decided to build our own simulator instead of using a state of the art simulators, like CloudSim [72] or PeerSim [73], because:

- With Caravela's prototype written in Go it would be difficult to export its functionality into PeerSim or CloudSim, since they are both written in Java. Maintaining two code bases, one for simulation and other for the prototype would be to cumbersome.

- With the simulator written in Go, we could re-use our main Caravela's components without maintaining an extra implementation for the simulation, but we took it further and we **utilized our complete prototype in the simulation**. This means that the code that runs in the simulation is exactly the same that would run in a real deployment (except a concurrency issue explained in the Section 4.2.2.1). We also **used the simulator for debugging** purposes instead of deploying the prototype in a hand-full of machines because it was way faster and easy to detect performance bugs that only manifest with the increased scale of the nodes participating.

4.2.1 Software Architecture

To understand how our simulator works, we present its software architecture in order to show how it integrates with Caravela's prototype. The simulator is composed by **four** components: the **Engine**, the **Request Feeder**, the **Metrics Collector** and the **Configuration Manager**. Figure 4.10 depicts the simulator's components and their relationships and it is here also as a reference for the following sections where we detail each component.

4.2.1.1 Configuration Manager

Our simulator is parameterizable providing us flexibility to test Caravela under different network sizes, request flow rate, requests profiles and more. The Configuration Manager is used by the simulator's engine to

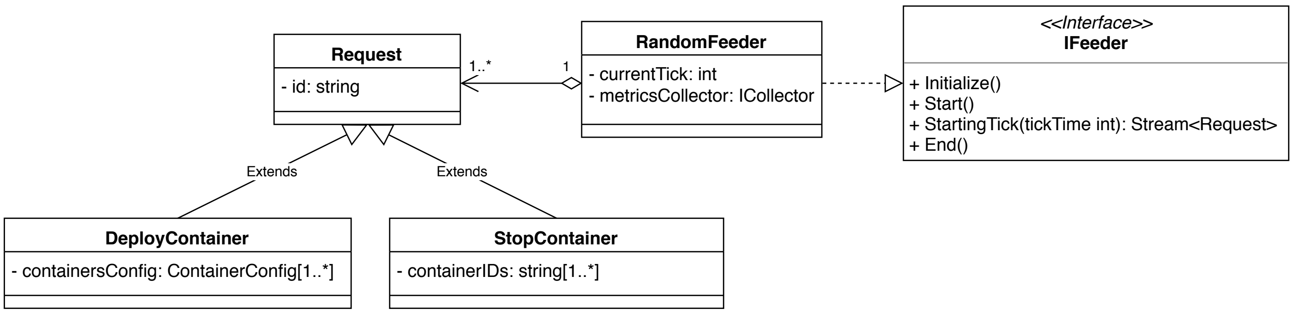


Figure 4.11: Request Feeder component and its interfaces.

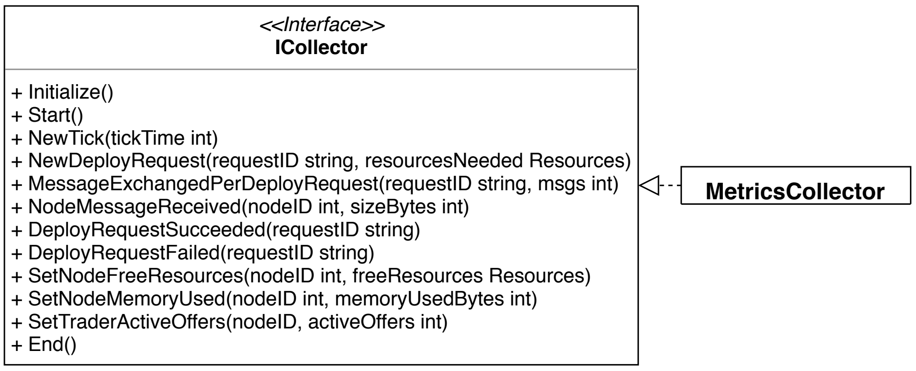

<<Interface>>
ICollector
+ Initialize()
+ Start()
+ NewTick(tickTime int)
+ NewDeployRequest(requestID string, resourcesNeeded Resources)
+ MessageExchangedPerDeployRequest(requestID string, msgs int)
+ NodeMessageReceived(nodeID int, sizeBytes int)
+ DeployRequestSucceeded(requestID string)
+ DeployRequestFailed(requestID string)
+ SetNodeFreeResources(nodeID int, freeResources Resources)
+ SetNodeMemoryUsed(nodeID int, memoryUsedBytes int)
+ SetTraderActiveOffers(nodeID, activeOffers int)
+ End()
MetricsCollector

Figure 4.12: Metric's Collector interface.

extract the parameters in order to configure itself accordingly. Table 4.2 contains the main parameters that we can specify for the simulator.

4.2.1.2 Request Feeder

Request Feeder is the component that generates a stream of user-level requests (deploy container and stop container) and sends them to the simulator engine. It generates the requests against the IUser interface which is the same interface consumed by the user's REST API we have in Caravela. This way the request feeder impersonates users consuming Caravela's public services. It provides the IFeeder interface for the engine. The engine asks a stream of requests from the request feeder for each tick, in order to be injected in the system.

4.2.1.3 Metrics Collector

Metrics Collector is used by the engine to store the metrics we need to evaluate Caravela. It stores request-level metrics like the number of messages necessary until the container is deployed and node-level metrics like the number of messages received by the node, the amount of bandwidth used by the node, the current resources usage and more. Figure 4.12 pictures its interface used to gather the simulation's metrics. The Metrics Collector also persists the metrics gathered to JSON files in order to be analyzed later.

4.2.1.4 Engine

Engine is the main component of the simulator. It is composed by *N* Caravela's Node component (see Section 4.1.1.1) which represent the nodes of the system. It also contains a mock instance of each external component that Caravela's nodes, need namely: (a) the Docker Engine (Docker Mock); (b) the nodes' Overlay (Chord Mock); (c) and the REST API WebServers (Caravela Remote Mock).

The Docker Mock simulates the Docker Engine implementing the interface IDocker. It maintains the containers running per node (and its details specified in the container's configuration), as if each node has an independent Docker Engine where it deploys containers.

The Chord Mock simulates the Chord overlay functions, i.e it implements the IOverlay interface. This Chord Mock is like a super fast Chord overlay between the nodes because it is implemented locally. We simulate the Chord's lookup protocol as it is (using the finger's tables per node) except the Chord's stabilization protocol executed in the background that is more important to test the system with node's failures but would introduce unnecessary extra overhead in the simulation. We used this mock to collect all the metrics information with respect to Chord's messages, e.g. collect how many messages a node received and crediting the messages to the respective container deployment request.

The Caravela Remote Mock simulates calls to our Caravela's HTTP wrapper; it implements interface IRemoteNode. Its responsibility is to forward the remote calls that each node does to the respective destination node. It also collects the metrics in regard of our Caravela's algorithms messages, similar to what we do in the Chord's Mock.

These mock components implement the interfaces we specified for Caravela's components in Section 4.1.1, it makes the change from the real ones to the mocks quite simple and easy.

Finally the last component is the Orchestrator. It is responsible for initializing all the remaining engine's components and coordinating the simulation. Its main function is to receive the requests from the request feeder and inject it in the nodes. It also gathers metrics like the number of requests injected, requests succeeded, accepted and rejected. It sends the metrics to the Metrics Collector.

4.2.2 Simulation

Now that we covered the simulator's structure we detail how the simulation takes place. The engine has the concept of time interval called a **tick**. Inside each tick, some requests from the requests stream are injected in the system. A tick duration is based on a real time window: we choose 20 seconds in our simulations because it was the best trade-off between granularity and performance.

Algorithm 4.1 represents the simulator's engine loop. First, the engine initializes the ticks counters (lines [2-3]), it runs the simulation in the main loop until the current tick is equal to the maximum ticks configured (line 6). In the 1st part of the main loop the engine requests the feeder a user's level request stream for this tick, and injects them in the nodes (lines [8-11]). The `SelectNode` function selects a random node from all the system's nodes. This simulates realistically the user's interacting with its own nodes. Our tick time is the "real time" that would be considered if the deployment was real; so, in the 2nd part of the loop the engine injects in the nodes the current tick time in order to run the node's actions that are dependent on real timers (line 12). An example of these actions is our Caravela's refresh algorithm that is triggered from time to time. In the 3rd part of the loop the engine updates the metrics collector with the information of the nodes (resources available, memory occupied and number of active offers in the traders) after all of the requests and actions of the current tick were completed (lines [13-14]). The 4th and final phase of the main loop consists in updating the tick counters and proceed to the next tick (lines [15-16]). When all ticks are completed the engine signals the feeder and the metrics collector that the simulation has ended (lines [17-18]).

Algorithm 4.1: Engine's simulation loop.

```
Data: $simulatorConfigs, nodes, requestFeeder, metricsCollector$
1  begin
2      $currentTickTime \leftarrow 0$
3      $currentTick \leftarrow 0$
4      $metricsCollector.Start()$
5      $requestFeeder.Start()$
6      while $currentTick < simulatorConfigs.MaximumTicks()$ do
7          $metricsCollector.NewTick(currentTickTime)$
           /* 1st - Inject the requests from the feeder in the system's nodes.   */
8          $requestStream \leftarrow requestFeeder.StartingTick(currentTickTime)$
9          while $requestStream.HasNext()$ do
10             $userLevelRequest \leftarrow requestStream.Next()$
11             $SelectNode(nodes).Inject(userLevelRequest)$
           /* 2nd - Run nodes' tasks that are dependant of real timers.          */
12         $FireRealTimeDependantActions(nodes, currentTickTime)$
           /* 3rd - Update the metric's collection with the nodes' information.  */
13         foreach $node$ in $nodes$ do
14             $metricsCollector.SetNodeFreeResources(node.FreeResources())$
           /* 4th - Advance to the next tick.                                     */
15         $currentTickTime \leftarrow currentTickTime + configs.TickInterval()$
16         $currentTick \leftarrow currentTick + 1$
17     $requestFeeder.End()$
18     $metricsCollector.End(currentTickTime)$
```

4.2.2.1 Performance Issues and "Parasimulation"

Caravela's Node component was developed as a real deployable one where performance matters. Thus, several actions in the node (e.g. remote REST API calls) were made using the Go's thread construct called Goroutine. The Goroutine is a kind of a lightweight thread from the Go's runtime. Go's runtime multiplexes the Goroutines in a small set of OS-level threads. Each Goroutine, when instantiated, occupies only 2KB-4KB so in idiomatic Go's code we can spawn hundreds of them. With the extensive use of them in Caravela's Node component, when we started injecting requests inside we had two problems: (a) thousands of Goroutines start to spawn in multiple nodes so our simulator started using several GB of memory for simulations with a moderate amount of nodes (35k); (b) we could not control when all actions unleashed by a request had finished, due to the Goroutines concurrency that we could not control from the simulator, this had impact when we wanted to advance a to the next tick but we did not know if there was something being processed or not.

To solve these two problems we introduced in the Caravela's Node component a mode where the Node would know that it was running under the simulator, a kind of "Parasimulation" operation mode (as with paravirtualization in Xen). In this mode, the Node does all the actions with one single Goroutine, more concretely the simulator's Goroutine that injects the request in the node. This way we only have one Goroutine that runs all the simulation. This reduced the memory consumption drastically and had as a side benefit that we knew when all the actions triggered by a request finished because the simulator's Goroutine processed all of them sequentially.

After this we started to notice that the simulation was very slow due to the the fact that we only used one Goroutine. In order to speed up the simulation we modified the simulator's engine to use a pool of Goroutines to process the 1st, 2nd and 3rd part of the simulation main loop Alg. 4.1. This maintained the benefits of our "Parasimulation" and sped up the simulations by a factor of 2.5, using all the power of a multi-core processor.

Listing 4.1: Example of an user's interaction with a Docker's daemon through the CLI.

```
$ docker run -cpus 1 -memory 512m -p 9001:80 yeasy/server
$
$ docker container ps
CONTAINER ID        IMAGE              COMMAND
CREATED             STATUS              PORTS                   NAMES
7ed63bc4ebe9        yeasy/server    "/bin/sh -c python"
48 seconds ago      Up 43 seconds       0.0.0.0:9001->80/tcp    peaceful
$
$ docker container stop 7ed63bc4ebe9
```

Listing 4.2: Example of an user's interaction with a Caravela's node daemon through the CLI.

```
$ caravela run -cpuClass "low" -cpus 1 -memory 512 -p 9001:80 yeasy/server
$
$ caravela container ps
CONTAINER ID                IMAGE                           STATUS
PORTS                       NAMES
7ed63bc4ebe9                yeasy/server                    Running
192.168.1.6:9001->80/tcp    peaceful
$
$ caravela container stop 7ed63bc4ebe9
```

4.3 CLI

As we mentioned in the beginning of this chapter, if we did not have time to fully integrate Caravela with Docker Swarm. We would like to have exactly similar interfaces and tools in order to maintain the consistency with the Docker Swarm ecosystem and allow people that use/develop them easily understand our work and prototype. We developed a Go's client which can be used as a Go's **SDK for** Caravela.

We also developed a **CLI application** that used our Go's client to interact with the Caravela daemon running in the node. Docker and Swarm also offer a CLI to consume their services. In the Listings 4.1 and 4.2 we show the same interaction of deploy a container, listing the containers and stop the container using the CLIs from Docker and from Caravela. As we can see they are very similar, while have some differences, e.g. the container listing in Docker has 2 more fields with time information about the creation time and up time. This would be easy to add to Caravela but it was not our main priority to mimic in perfection the command results. In line 1 of both Listings 4.1 and 4.2 we can see that Caravela has one more flag called `cpuClass`. This one was introduced by us to deal with the processor speed heterogeneity something that is not available in Docker ecosystem at the time of this work production.

So, in the end we try to have a subset of Docker commands and flags that were interesting for us, and use them with the same syntax and semantics as Docker. At the same time we introduced new commands or flags to handle things that current Docker could not and that are relevant to Edge Cloud deployments.

4.4 Unit Testing

When the code base grew with some of our advanced features, the number of bugs found in our prototype increased and it was affecting the consistency of our evaluation results, and our perception about the prototype performance. So we decided to write a set of Unit Tests for the crucial data structures that had critical impact in our algorithms decisions. We wrote around **150 unit tests** for the data structures that handled the GUID management, the GUID mapping with regions and the resources management. This helped to find many bugs and we believe has increased the robustness of our prototype.

4.5 Summary

In this chapter we started by describing Caravela's software architecture with all the components we developed connecting them with the data structures and algorithms presented in Chapter 3, where we described Caravela's distributed architecture.

Then we described CaravelaSim, a cycle-based simulator that we developed to evaluate Caravela's prototype scalability (with tens of thousands of nodes), without doing a second code base in Java to use PeerSim. We presented CaravelaSim's software architecture, and how we re-utilized Caravela's components to test them inside the simulator. We briefly detailed how the simulation occurs in the simulator's engine and how we solved some performance issues.

Finally, we compared our CLI tool syntax and semantics with Docker's to prove that they are very similar. It would be easy for an user from Docker deploy a container in Caravela.

5 Evaluation

Contents

In this chapter, we evaluate Caravela using the simulator we developed comparing it with other approaches (Docker Swarm adaptation and a random based naive approach) for the resources discovery and scheduling. We start by stating how we configured CaravelaSim to do the simulations in order to be efficient and realistic. After that, we detail how we generate a realistic dataset of requests to feed the simulator. Next, we present the metrics we gathered during the simulations to evaluate our approach and the others. Later, we presented Caravela's configurations we used during the simulations. After this, we present the results obtained. Finally, we dicuss what are the main advantages of Caravela to the users of Edge Clouds when compared to the other approaches.

5.1 CaravelaSim's Configurations

We used CaravelaSim which is our cycle-based simulator. Therefore the first thing we needed to set for all of the simulations was the duration of a simulation tick and the length of the simulation. We decided to have ticks of **20 seconds** and the simulations with a fixed duration of **360 ticks** which means a simulation of **2h** in the system's real time. We also run all the simulations with CaravelaSim's $Multithreaded$ parameter set to $true$ to speed up the simulation, so there is concurrency in requests that are submitted in the same tick.

We also needed to decide how to distribute the maximum resources available for each node when setting up the Caravela's nodes in the simulator. For this, we used the distribution represented in Figure D.1 to assign the maximum available resources for each node. This distribution is exactly the same we use for Caravela's resources regions configuration parameter. We did this according to what usually happens in volunteer computing environments, where each person only donates the resources he/she has available; so donating only 1CPU and 512MB of RAM is something normal because the user probably needs the rest for his/her own personal computations. We used that distribution because it is realistic to consider that in an Edge Cloud there are fewer highly capable nodes than weaker nodes. Even with the maximum resources being picked with a pseudo random generator from the presented distribution, we guarantee that **all the simulations have exactly the same nodes with the same amount of maximum resources** because we provide the same seed for all of them.

5.2 Dataset Details

We generated our testing dataset using our parameterizable **random request feeder** (shown in Section 4.2.1.2) to feed the simulator. Our random request feeder is simple and flexible in order to test the system under any desirable circumstance. It receives two parameters that influence: (1) the **rate of deployment and stop requests** to inject in the system during the simulation; (2) the **profiles of requests** that are used to generate the stream of real requests. In the next sections we present how we configured the random request feeder to generate realistic streams of requests.

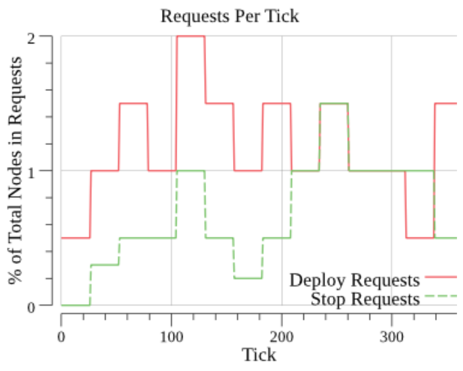


Figure 5.1: Distribution of Deploy/Stop requests per tick, over time proportionally to total nodes configured for simulation (in percentage).

5.2.1 Rate of Requests

In order to generate a stream of requests we needed to configure the rate of requests (deploy and stop) that would be submitted in each tick of the simulation. We tuned the parameters with demanding but realistically rates in mind, so we have: (1) $[0.5\% - 2\%]$ of the total nodes involved in deployment requests per tick (20s); (2) $[0\% - 1.5\%]$ of the total nodes involved in stop requests per tick (20s).

The complete request rate distribution during the simulation is pictured in the Figure 5.1. We used this rate of requests for all the simulations because the system starts with all the nodes without any container running (all the resources available to be used). Therefore in the first phase of the simulation we have more deploy requests than stops in order to fill the system. In the middle of the simulation we have the stream of requests with nearly the same deploy and stop containers. Our rationale in this rate of requests was to compare the approaches when the system had at least 50% of resources used, because a real well designed system is always at least 50% utilized, otherwise it would be overdimensioned or poorly used system. In a system with low utilization, finding resources would be much easier and it would not be difficult for any naive strategy. Since the amount of requests submitted is a **percentage of system's total nodes**, when we scale (up or down) the network size, it also scales the amount of requests being submitted.

5.2.2 Requests Profiles

The second and last parameter that we must pass to our random request feeder is the request profiles for the container's deployment requests. The requests profiles define the **amount of resources requested** and their **frequency**. The requests profiles distribution are pictured in Figure D.2. The profiles are divided in two equal halves of requests in terms of its frequency. We have 50% of the requests requesting low resources which would be equivalent to small services and micro services deployments, while the other 50% of requests consume much more resources and are equivalent to larger applications deployments or background tasks that consume more resources.

The feeder will generate the number of requests, decided by the request rate for the tick and injecting it in the nodes. The resources necessary for each request will be generated from the profiles we presented. We guarantee that even with the pseudo random generator being used to choose the resources for each request from the profiles, **all the simulations are run exactly with the same stream of requests in the same order** because the initial seed is equal for all of them. The requests are initially **injected in a random node** simulating the realistically interactions of the node's owners submitting requests into the system via their nodes.

5.3 Metrics

We want mainly to assess the scalability of the approaches when we grow the size of the network with more nodes. So, we want to verify that there are not any kind of bottlenecks in terms of memory, network usage and computational power usage in the nodes. We also want to assess the efficiency and efficacy of the scheduling and discovery, specially when we scale up the network. Thus, we gathered the following metrics during the simulation:

- **Bandwidth used per node:** used to check if some nodes are using too much bandwidth to maintain their functions;
- **Messages received per node:** complements the previous bandwidth metric;
- **Memory used per node:** used to check if some nodes are using too much RAM to hold Caravela's data structures;
- **Messages exchanged (hops) in the system to find resources (per request):** used to check how many messages we need, in order to find the resources and deploy the container(s) so that we can check the scheduling and discovery algorithms efficiency. This is highly correlated with the latency between the submission of the request and the container(s) being deployed. The latency problem is specially important in a WAN scenario such as an Edge Cloud;
- **Amount of deployment requests successfully fullfilled:** to check the user's satisfaction using the system;
- **Scheduling algorithms efficacy:** according to the global policy used (binpack or spread). It is used to check what is the degree of consolidation (when binpack is active) or load balancing (when spread is active);
- **Trader's offers (only applied to our solution):** used to check if some nodes/traders are managing too many offers.

5.4 Competitor Approaches

We implemented in our prototype two other approaches to discovery and resource scheduling to test against ours, namely: (a) **Swarm** adapted for a Chord overlay; (b) **Random** also over a Chord overlay. These approaches were implemented by simply creating a new resource Discovery backend in our prototype due to its flexibility and plugable nature.

5.4.1 Docker Swarm (Adaptation)

This approach implements the Docker Swarm algorithms with a single peer node (called master node) receiving all the requests to deploy the containers. Its has the job to schedule all of them onto the other nodes. The only difference in our adaptation is that the master node is the Chord's node responsible for the key 0, instead of a pre-defined node with a static IP address known by all the other nodes. This adaptation is particulary accurate to the case of Edge Clouds because there is no natural central administration. When a node receives a request from the user, it sends it to the master node in order to deploy it in the system. The master node knows all the nodes in the system and chooses the best node to schedule the requests directly depending on the global policy. This approach represents the centralized solution used by the Docker Swarm scaled up to Edge Cloud size. To simplify, from here onward we refer to this approach as **Swarm**. This approach provides to us a near "oracle" approach to compare with, because it takes the decision based in all the nodes state, so we can see how our approach compares with a near optimal solution regarding efficacy and efficiency of the resource discovery and scheduling algorithms.

5.4.2 Random

This approach is straightforward: when a node receives a user request to deploy a container, it generates a random Chord key, it does a Chord lookup with that key and asks the key's handler node if it has enough resources to deploy the request. If the discovered node does not have resources, Chord restarts the algorithm generating another random key, which is usually called a retry. $\mathrm{Caravela's}$ parameter $MaxDiscRetries$ defines the maximum number of retries that Random can do. If the contacted node has enough resources, it launches the container on it. If it does not find any node (until the maximum retries are reached) to deploy the container(s), it is considered failed and the user must retry manually later. To simplify, from here onward we refer to this approach as **Random**. This approach also uses Chord and it its presented here as a possible naive solution in order to check that a solution for our problem was not trivial. Since it only uses Chord, it does not do any more operations in the background. It will scale in terms of messages with $MaxDiscRetries * log_2(N)$ which is a constant defined by the number of retries and the scalability of Chord's lookup. This approach helps us compare our solution in terms of overhead and scalability because Random has a minimum overhead and good scalability factor.

Parameter	Random	Multi-Offer	Swarm
$Schedule_{policy}$	-	Binpack	Binpack
$\#CPU_{overcommit}$	100	100	100
$RAM_{overcommit}$	100	100	100
CPU_{slices}	1	1	1
$ResourcesRegions_{mapping}$	-	See Fig D.1	-
$Refresh_{interval}$	-	15min	15min
$MaxRetries_{discovery}$	3	1	-
$SuperTraders_{factor}$	-	7	-
$Chord_{key}$	128-bit	128-bit	128-bit
$Discovery_{strategy}$	Random	Multi-Offer	Swarm

Table 5.1: Caravela's configurations values for each discovery approach.

5.5 Test Configurations

Before we present the results we want to present the configurations parameters that we used in our prototype for each of the discovery approaches we test. We remind that the different tested approaches are all implemented in our Caravela's prototype only changing the resources discovery component. Table 5.1 present the configurations for each one of the approaches. Some of them do not have some parameters because they do not apply them in anyway, e.g. Random's approach does not implement any type of global policy because it simply cannot.

5.6 Results

All the results presented here come from simulations of CaravelaSim running on Amazon Web Services (AWS) Elastic Compute Cloud (EC2)[1] platform, using an c4.8xlarge instance with 36vCPUs and 60GB of RAM backed up by an Intel Xeon E5-2666 v3 (Haswell) with 25M Cache and 2.60 GHz.

We used **two network sizes** to present the results: (1) **65,536** nodes which is already near the double of GUIFI.net community network current size (≈36k); (2) **1,048,576** which is **16 times** more than the 65,536. For simplicity we abbreviate the nomenclature of the networks size, 65,536 for 64k and 1,048,575 for 2^{20}. We do not present higher network sizes because the simulations with 2^{20} already took ≈1h in the AWS instance. But from the results we will present, we envision that the results would scale in similar ways for 2^{23} and 2^{27} nodes which would be in the same scale of the BitTorrent network size. We **do not** present the results of the Swarm approach for the 2^{20} network size because the simulations would take too many hours. So, not even the simulation for Swarm could not scale well, because the master node look to all nodes for each deployment request. As we will see, in the simulation with 64k nodes the results hint that at 2^{20} scale, Swarm would be

[1] https://aws.amazon.com/ec2/

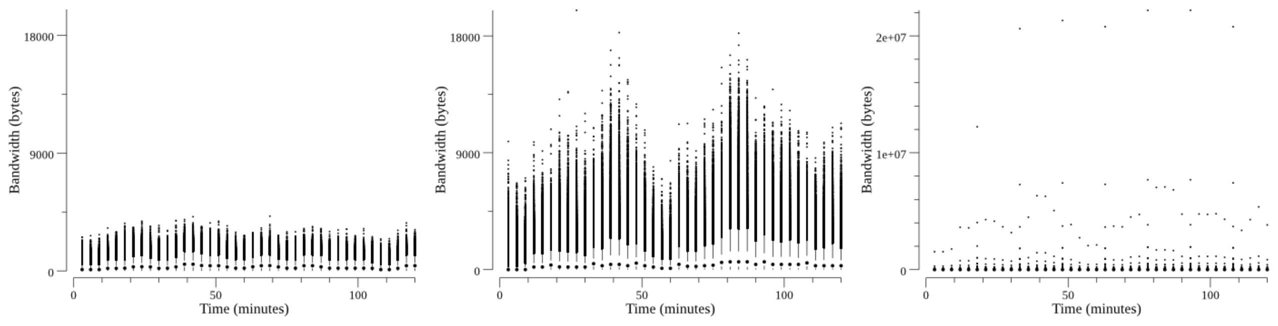

(a) Random (b) Multi-Offer (c) Swarm

Figure 5.2: Bandwidth used per node on receiving over time, in the 65K node's network (Quartile Plots).

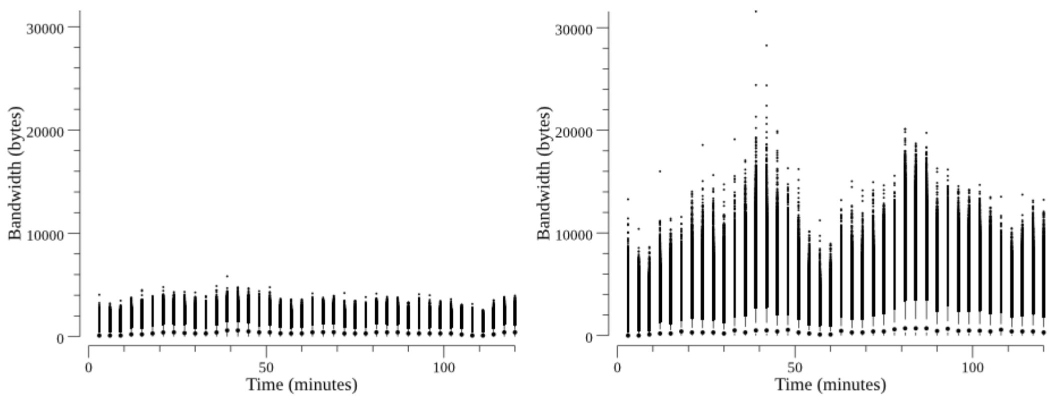

(a) Random (b) Multi-Offer

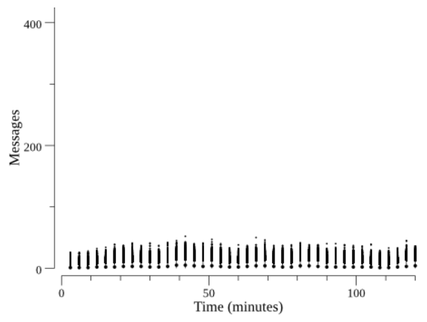
Figure 5.3: Bandwidth used per node on receiving over time, in the 2^{20} node's network (Quartile Plots).

unfeasible.

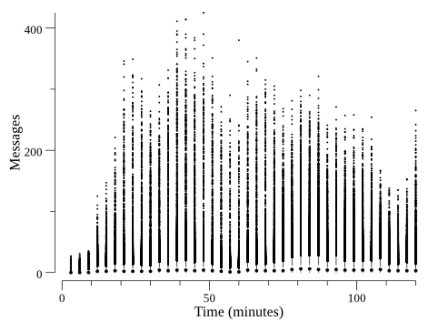
5.6.1 Node's Network Usage

In terms of the bandwidth used by each node we present Figure 5.2 for the 64k nodes network and Figure 5.3 for the 2^{20} nodes network, showing the distribution of the bandwidth consumed by each node, in 3 minutes intervals, overtime. Notice that the scale in the Random and Multi-Offer plots are equal and the one in Swarm is necessarily different **(much higher)**; this will be the norm in the rest of the results because Swarm does not scale at the same level as the the former ones. Our Multi-Offer approach has many outliers represented but they only account for [7%-9.5%] of the total system's nodes. These outliers are mostly the super traders which are in charge for the offers management (still mostly below 15KB over 3 minutes, ≈5KB per node). Even our (Multi-Offer) highest outliers consume **1000 times less bandwidth** than the Swarm master node which is the biggest outlier in the Swarm's plots.

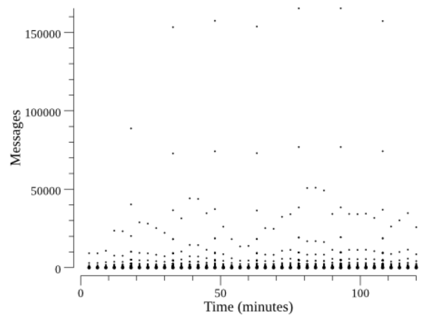
With the increase of 16 times in the network size our biggest outlier only increased by a factor of 1.5 the



(a) Random

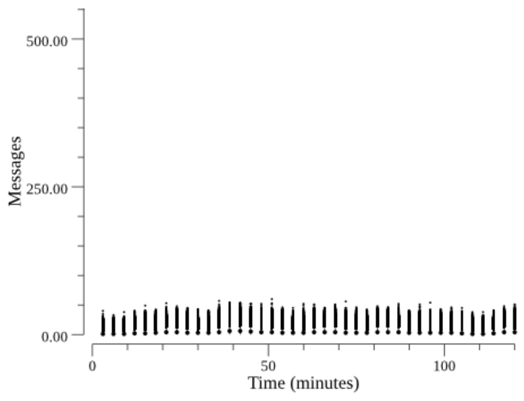


(b) Multi-Offer

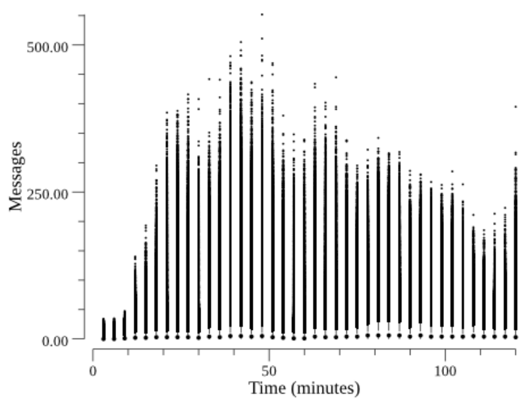


(c) Swarm

Figure 5.4: Number of messages received per node over time, in the 64k node's network (Quartile Plots).



(a) Random



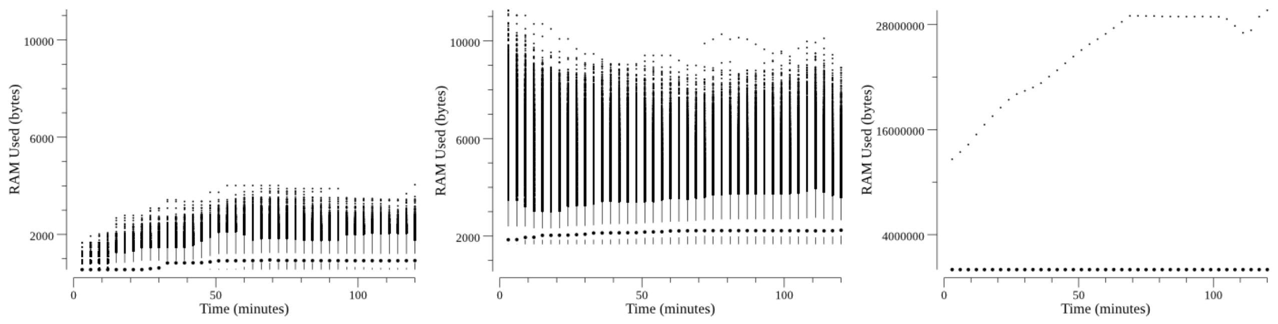
(b) Multi-Offer

Figure 5.5: Number of messages received per node over time, in the 2^{20} node's network (Quartile Plots).

bandwidth consumed while the overall distribution only increased by a factor of ≈1.2 (on average), which is a very good scalability factor for a **16 times** increase in the network size. As we said, we do not have the results for the Swarm for the 2^{20} network but, based on other tests we did with networks size varying from 8k to 64k the bandwidth consumed by the master node doubles when we double the network size.

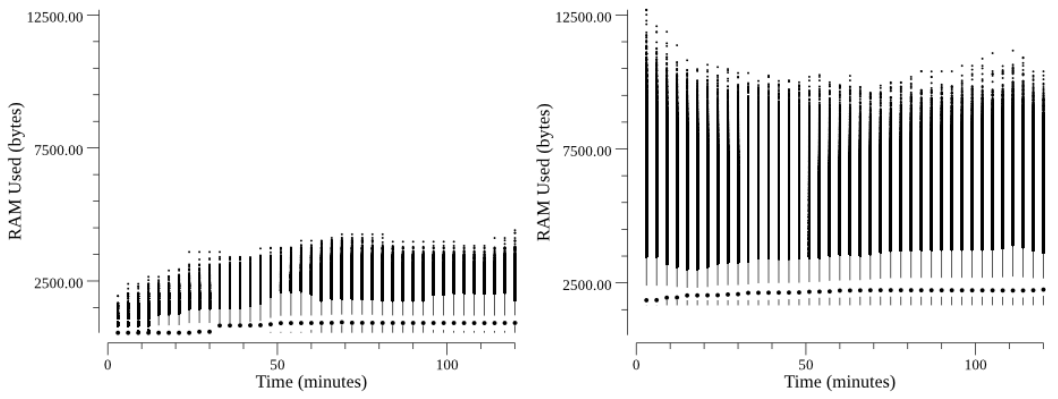
So, if we do the simple math, in the 2^{20} network size the master node would consume ≈200MB over 3 minutes, which does not seem too much but this means that it would consume **≈2.8TB of bandwidth** over a month span which, for an Edge Cloud user, it would not be possible to have the master node due to ISP's fair usage. When we do the math for Multi-Offer considering 10KB, per node in the 2^{20} network, each node would only consume **≈144MB of bandwidth** in a monthly span. Nodes near the Swarm's master in Chord's ring also consume a lot of bandwidth because Chord's queries that go to key/ID 0 pass through them, they are the outliers below the highest outlier.

Just by looking to the bandwidth, we can see that Swarm does not scale well. Our Multi-Offer approach uses ≈2 times more bandwidth than Random (on average), due to our offer's system management (creation and updating), but it still maintains a low bandwidth consumption overall anyway.



(a) Random **(b)** Multi-Offer **(c)** Swarm

Figure 5.6: RAM used per node, in the 64k node's network (Quartile Plots).



(a) Random **(b)** Multi-Offer

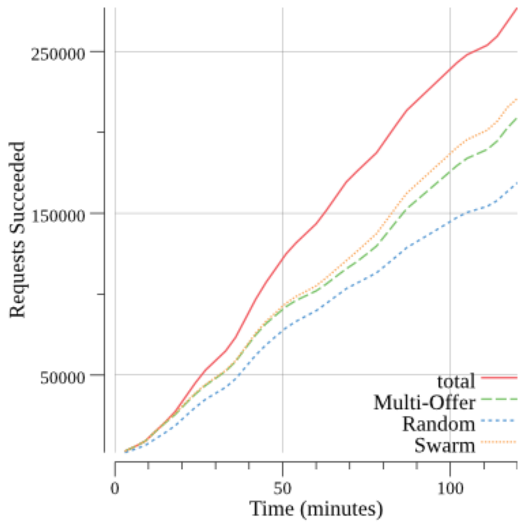
Figure 5.7: RAM used per node, in the 2^{20} node's network (Quartile Plots).

The bandwidth was measured by summing up the size of the JSON data sent in all the messages exchanged during the simulation (because we used the HTTP to build our protocols as the real Docker Swarm does). With a binary encoding of the messages we could reduce the message's size in ≈65%, but this would only delay the Swarm's scalability problem.

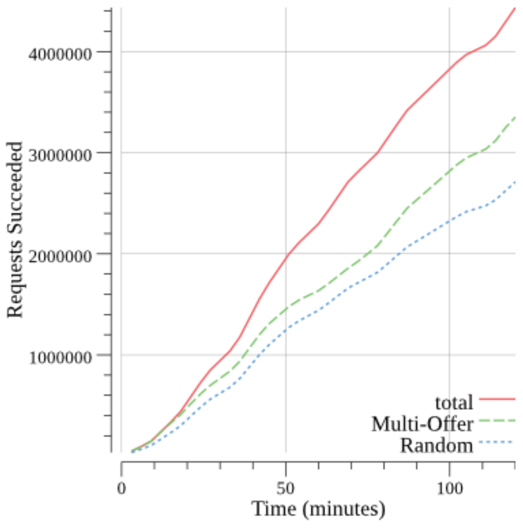
Figure 5.4 and 5.5 represent the distribution of the number of messages received per each node in the 64k and 2^{20} network size respectively. As we can see, the results are similar to the bandwidth ones.

5.6.2 RAM Used per Node

In this section we assess the RAM used per node in order to maintain all the information about Caravela's algorithms and structures, we exclude Chord's and HTTP WebServer structures, as they would probably be present in any Docker HTTP based solution on top of an Edge Cloud. This way we assess specific overheads of our solution. Figures 5.6 and 5.7 represent the distribution of RAM usage per node over time for both network sizes. Multi-Offer has [8%-14%] outliers represented, again due to the Super Traders that consolidate



(a) 64k nodes network.



(b) 2^{20} nodes network.

Figure 5.8: Deployment requests successfully attended over time (Cumulative).

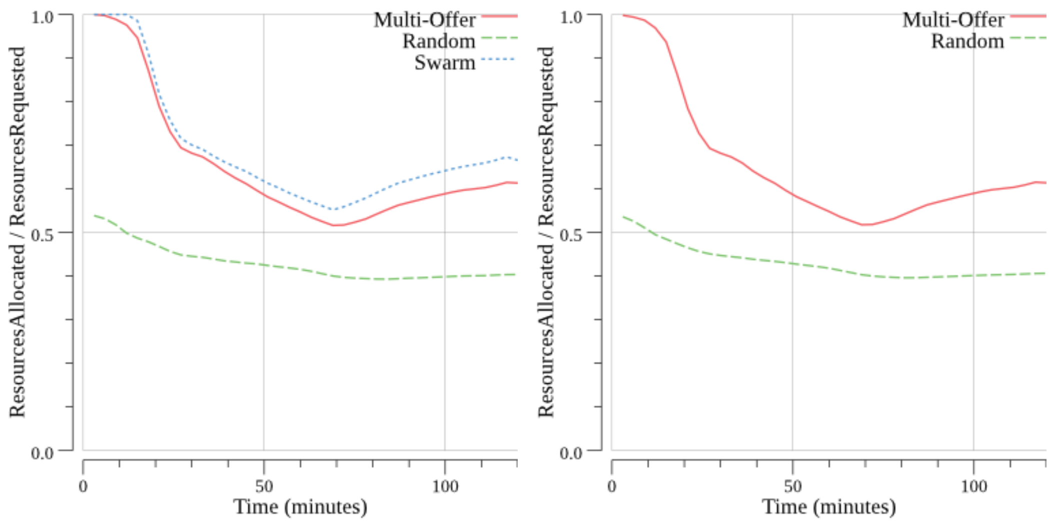
more offers in them, using more RAM. We can check the same scalability issues for Swarm and an identical difference between Multi-Offer and Random as the ones presented in the previous section.

Our Multi-offer approach consumes ≈2.5 times more RAM than Random, due to the offers management. Our highest outlier only consumes ≈11KB of RAM for the 64k network which is a low consumption. When we increased the network by **16 times** Multi-Offer nodes RAM usage only increased by a factor of ≈**1.25**. Swarm's master node consumes a lot of RAM (**2200 times** more than Multi-Offer highest outlier in the 64k network) because it needs to save where all the requests are running and their respective details to be consulted by the users later. It also must save information about every node in the system in order to be able to deploy the containers on the right node. In our Multi-Offer approach the data is spread across all the system's nodes, making it scalable. Applying the same extrapolation we did in Section 5.6.1, for the 2^{20} network, Swarm master node would consume more RAM than any typical user node could offer.

5.6.3 Deployment Requests Efficacy

Here we assess how many user's deployment requests the system could satisfy for each approach, which would translate in the **user's satisfaction and the discovery and scheduling algorithms efficacy**. Figure 5.8 represents the accumulation of successful deployed requests for the approaches in both of the network's sizes. As we can see in Figure 5.9(a), our Multi-Offer approach is close to Swarm, which has a completely global view of the system (checks all nodes) in order to find a node to deploy the container, increasing its chances to find a node to deploy the container(s).

Our Multi-Offer approach in the end of the simulation deployed ≈**24% more requests with success than Random**, but we should not forget that Random is configured to retry 3 times when trying to deploy a request.

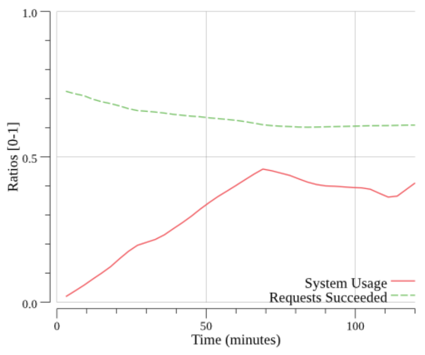


(a) 64k nodes network. **(b)** 2^{20} nodes network.

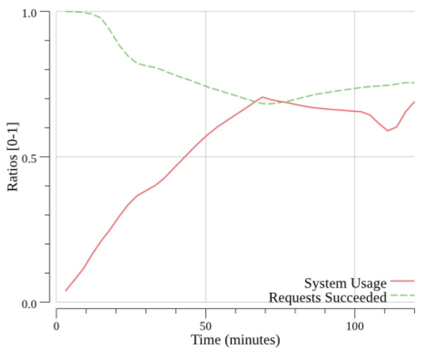
Figure 5.9: Deployment requests resource allocation efficacy (Cumulative). Ratio [0-1].

So, as we will see in Section 5.6.4, this deployment efficacy comes at the cost of many messages exchanged between the request submission and the container launch (low deployment request efficiency). Why we choose 3 retries?, because it was the best compromise for Random between the deployment efficacy vs the efficiency of finding the resources, maintaining it at its best in both. Less than that, the deployment efficacy is to low to be acceptable for a regular use. Due to space constraints we did not post the simulations results with only 1 retry for Random, but we measured ≈**70%** more requests/containers deployed with success in Multi-offer compared with Random. Consequentially, this reduction in the Random's retries also: lowered its overhead in terms of bandwidth consumed, and put its deployment request efficiency at the same level as Multi-Offer, as we will see in Section 5.6.4.

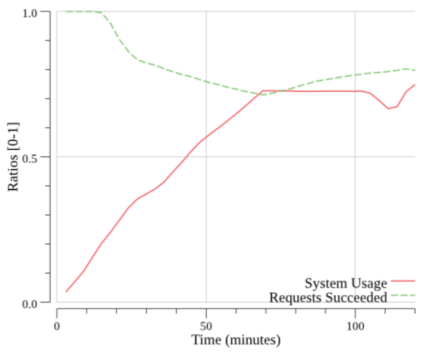
Figure 5.9 shows the resource allocation efficacy metric over time for both of the networks. The allocation efficacy metric consists on the following ratio: $totalResourcesAllocated/totalResourcesRequested$ cumulative over time. This metric was collected because we started to notice that the Random's deployment efficacy in terms of requests attained with success also came from the fact that it rejected much more requests with larger resources needs (hence, making it little useful for any other than small simple requests). Due to this fact, subsequent small requests would have much higher probability do find a node compared to Swarm and our Multi-Offer that could satisfy much many requests with larger resources needs sooner. This metric corroborates our hypothesis: Figure 5.9 pictures that when the simulation starts (with all the node's resources free) **our Multi-Offer allocation efficacy starts can allocate ≈2 times more requested resources than Random**, which means that even with all the nodes free Random cannot allocate requests with large resources needs. Our Multi-Offer approach has a very similar resource allocation efficacy as Swarm, which is very good because Swarm is an “oracle” approach that can take near-optimal decisions. All the approaches scale in the same way



(a) Random

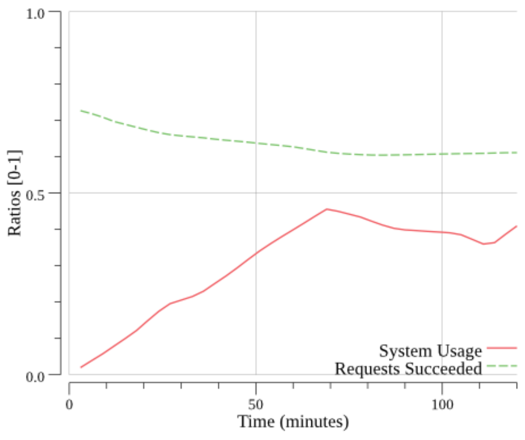


(b) Multi-Offer

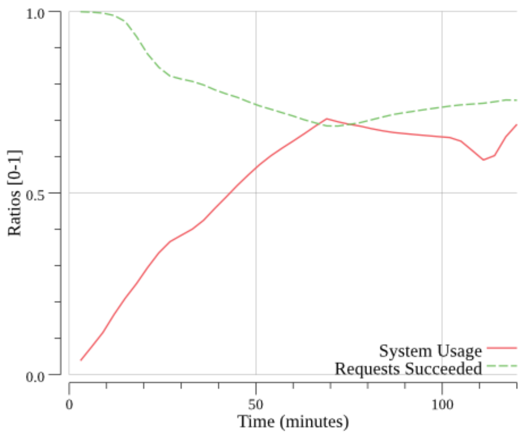


(c) Swarm

Figure 5.10: System's resources usage vs deployment requests succeeded (cumulative), in the 64k node's network. Ratios [0-1].



(a) Random

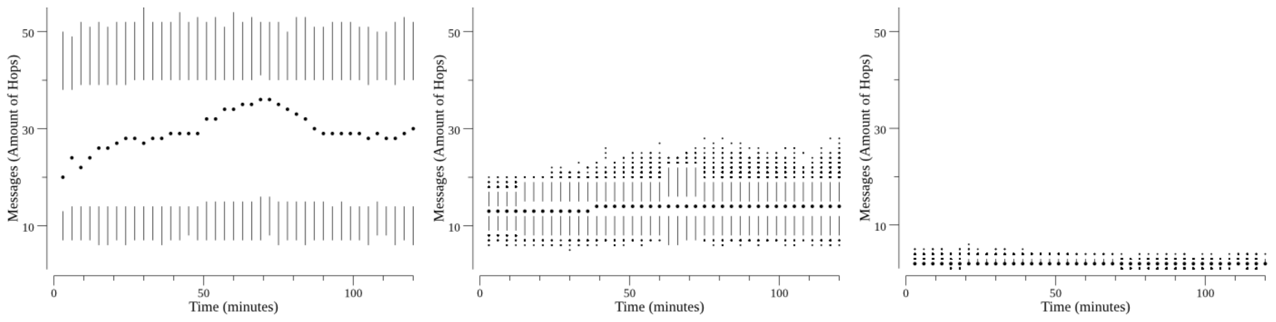


(b) Multi-Offer

Figure 5.11: System's resources usage vs deployment requests succeeded (cumulative), in the 2^{20} node's network. Ratios [0-1].

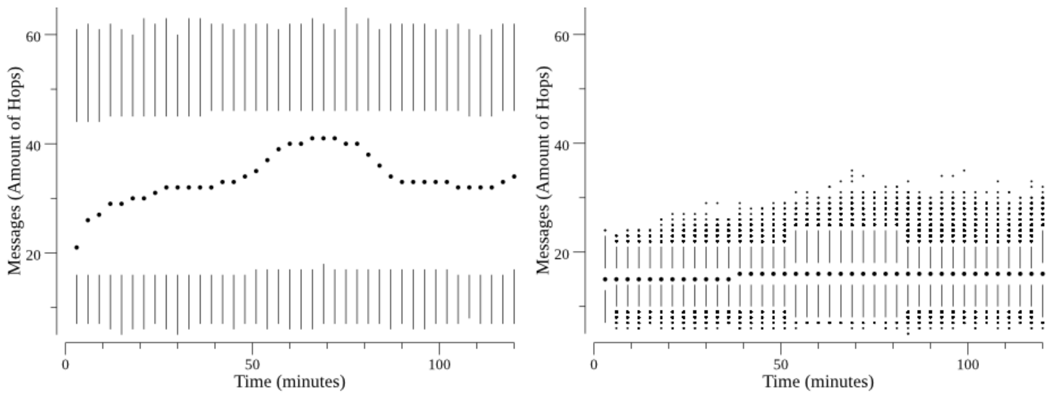
from the 64k node's network to the 2^{20} node's network. In this metric we would expect Swarm to scale in similar way from the 64k network to the 2^{20} because it consults all the nodes. If exists a node with capacity for a request, it is almost guaranteed that the Swarm can find it making the results for this metric near-optimal for Swarm.

To complement the previous metrics Figure 5.10 and 5.11 show the system's resources utilized ratio over time (red line), and the ratio of $succededDeploymentRequests/totalDeploymentRequests$ cumulative and over time (green line). As the figures depict, Multi-Offer is close to Swarm which we already saw regarding the requests deployment efficacy. We can also see that Multi-Offer and Swarm fulfill many more requests (light and heavy) because they can utilize 25% more system resources (total) than Random. We can also see that Random requests succeeded ratio is lower than Multi-Offer, even with Multi-Offer having the system with 25% more resources utilized (i.e. 25% more resources free).



(a) Random **(b)** Multi-Offer **(c)** Swarm

Figure 5.12: Distribution of the number of messages exchanged (critical path) per deploy request submitted, in the 64k node's network.



(a) Random **(b)** Multi-Offer

Figure 5.13: Distribution of the number of messages exchanged (critical path) per deploy request submitted, in the 2^{20} node's network.

5.6.4 Deployment Requests Efficiency

In this section we assess the requests deployment efficiency, which translates to resource discovery algorithm efficiency. In Figures 5.12 and 5.13, we present the distribution of the number of sequential messages exchanged (amount of hops) in the system (critical path), per request, between the time of the request submission in a node and a node accepting the container to launch. As we can see in Figure 5.12, for the 64k node's network, Swarm has a very small constant cost ($\approx$3 messages), because after the first time a node contacts the master, it saves the node IP and all the following communications (e.g. deploy a container) are direct via IP. For any network size, Swarm would always have this cost which would be near perfect but as we already saw the Swarm's master node collapses with the load.

The difference that exists between our Multi-Offer and Random is the fact that the Random uses 3 retries (= 3 Chord's lookups) while Multi-Offer only needs one. Due to this, we can see that from the 64k network size to the 2^{20} network size the difference between Multi-Offer and Random increases substantially due to the retries. Since the Edge Cloud's nodes are connected via a WAN, the number of messages traded until we find the resources (or not) has an impact int the time the user needs to wait before knowing that the deployment succeeded (or failed). Multi-Offer median requests cost (≈12 messages/hops per request), which is **half** of Random median cost (≈24 messages/hops per request). **Multi-Offer's highest outliers are smaller than Random's median value**. Mutli-Offer's median value of hops per request maintains constant even when the system has more than 50% of the system resources used, while Random median value increases due to the retries needed.

As we mention in the deployment request efficacy section (see Section 5.6.3), if we run these simulations only changing the Random retries to 1, the request deployment efficacy reduces **70%** compared with the Multi-offer. But the deployment request efficiency presented in this section would be similar to the Multi-offer because both would only use one Chord lookup.

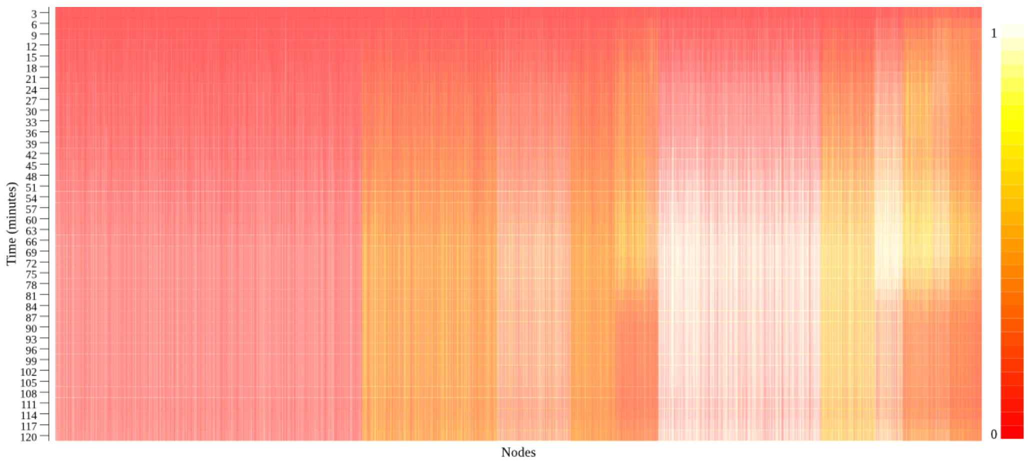
5.6.5 Global Scheduling Policy Efficacy

In this section we want to assess the efficacy degree of the approaches enforcing the global scheduling policies (binpack and spread). Our simulations were conducted with the binpack policy because it is the most useful for Edge Clouds, in order to maximize the node's resources used (consolidation). Appendix D contains the same results we presented in the previous sections but changing the scheduling policy to spread. Figure 5.14 shows heatmaps with the time-line in the Y axis (top -> bottom) and all the node's of the system in the X axis. Each vertical colored line represents a node in the system. The line color encodes the % of resources utilized in the node, the scale red -> yellow -> white represents the % of resources going from totally unused to completely utilized. The vertical lines (nodes) are ordered in ascending order by the maximum resources that a node could offer (small nodes -> larger nodes). So, white/yellowish areas represent sets of nodes that have all of their resources utilized. As we can see, Swarm is very effective in packing the requests in the nodes, it can take the best decision sooner. Multi-Offer is worse than Swarm because it takes the scheduling decision based on a sub-set of the global knowledge (hence its scalability in the other metrics), but it is better than Random as we can see in Figure 5.14, by inspecting the white areas and comparing with Swarm's white/yellowish areas. We do not present the plots for the 2^{20} network because the results were exactly the same.

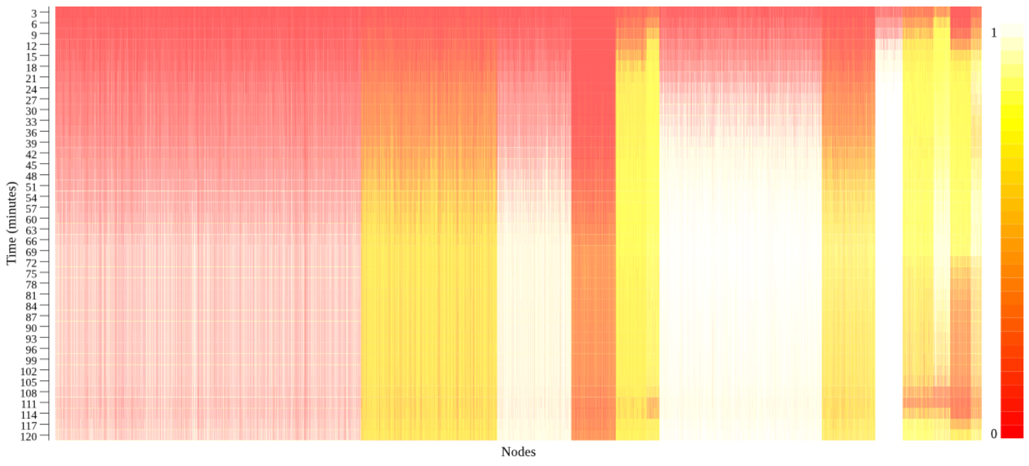
Even with our completely decentralized architecture and algorithms, **Multi-Offer can enforce the binpack global scheduling policy with a great degree**. Random simply cannot enforce the policy because it tries nodes randomly, launching the container(s) in the first node with enough resources available.

5.6.6 Trader's Active Offers - Multi-Offer only

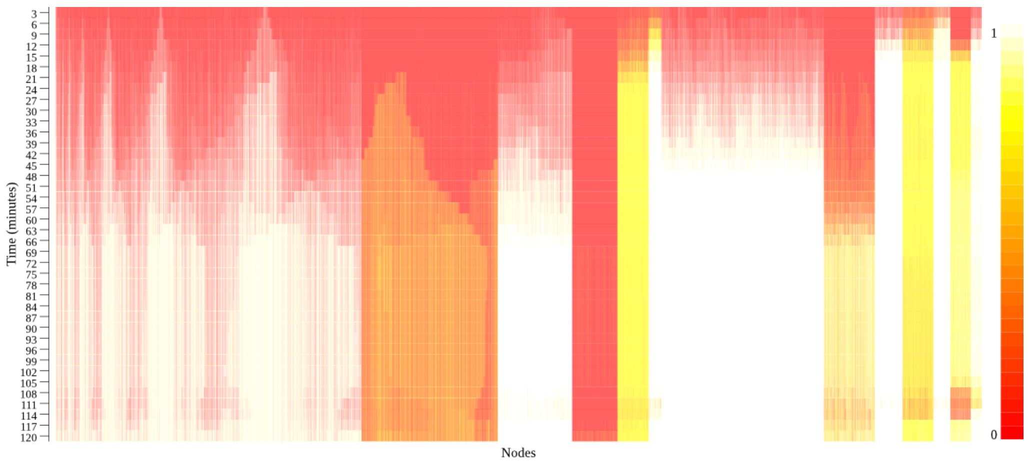
This metric is used to assess that the super traders in the Multi-Offer do not suffer from any kind of significant bottleneck due to the fact that they manage a lot more offers than if there were no super traders. Figure 5.15 shows the distribution of trader's offers active (per super trader). The number of nodes (super traders)



(a) Random

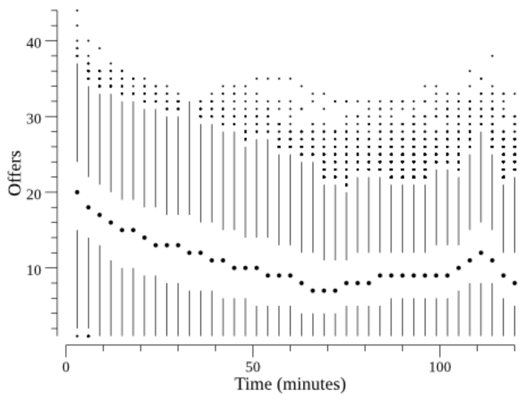


(b) Multi-Offer

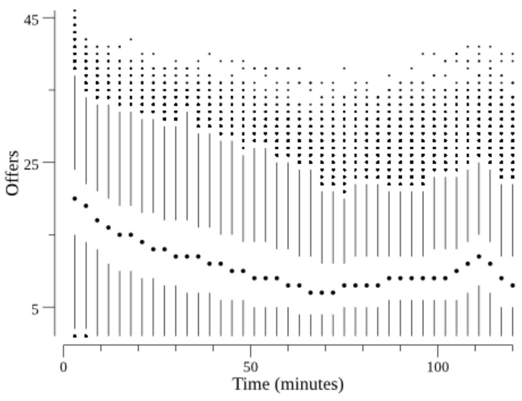


(c) Swarm

Figure 5.14: Node's used resources per node over time, in the 64k node's network.



(a) 64k node's network



(b) 2^{20} node's network

Figure 5.15: Distribution of trader's active offers maintained per Super Trader across time.

presented in the plots are **14% of the system's total nodes**. All the remaining nodes manage 0 offers. As we can see, as we scale the network **16 times**, the concentration of offers barely increases, meaning that the offers distribution remains similar. Each offer occupies $\approx$90 bytes in a node, so if we take the highest outlier from the plot the **memory overhead that a super trader sustains, compared with other nodes is, about just 3.5KB more RAM used, which is a very small overhead**. Another thing to notice is that the amount of offers in the traders is inversely proportional to the system's utilized resources.

5.7 Summary

After the analysis of the previous results we can conclude:

1. The Swarm's master node concentrates too much load in terms of network's usage and computational power used to decide where to place the requests, therefore it is unfeasible for networks with tens of thousands of nodes. On the opposite side, for smaller networks it can schedule containers using $\approx$3 messages, and it can enforce a global policy "perfectly", choosing the best nodes which is better than the other two approaches.

2. Random is the one that scales better in terms of bandwidth and RAM consumed but, in contrast, its amount of requests attended with success, the deployment request efficiency and the efficiency of global resources consolidation are the worst of the three.

3. Multi-Offer has the strengths of the other two, while minimizing the effects of their weaknesses. So it is an approach that would fit in a deployment of an Edge Cloud with 1M of nodes or even more, while maintaining a very interesting performance in discovering the resources and scheduling the containers.

6 Conclusion

Contents

We started this work by noticing the movement of Cloud Computing from the internet's backbone into the end user's proximity (network's edge). This movement conjugated with the volunteer computing offers a lot of underutilized resources that could be leveraged in a cloud-like platform to be used by all the volunteers. These new movements are designed as Edge and Fog Computing. The works in these areas lacked decentralized and scalable prototypes that could cope with the amount of devices that could be leveraged.

Our proposed objectives were to: formulate taxonomies for the Edge Cloud, Resource Management and Fairness works. Then, we proposed to implement and evaluate a prototype that had decentralized and scalable features to handle the amount of nodes and deployment requests, maintaining good deployment speeds and success rates, at the cost of optimal resource discovery and scheduling algorithms.

We started the Caravela's architecture by specifying the desirable properties, like scalability, handling devices capabilities heterogeneity, and peer churn resilience. After that we detailed the prerequisites for our prototype. Then we described the developed architecture and algorithms. The resource discovery algorithm (based on our resource's offer system) runs on top of the Chord overlay. Chord lookup mechanism efficiency and scalability is responsible for Caravela's scalability and efficiency in deploying containers. We also incorporated the possibility of having global scheduling policies (binpack and spread) in our completely decentralized architecture. As a bonus feature, we introduced the possibility of having stack deployments with container's group-level scheduling policies (co-location and spread) which are not present in Docker Swarm.

We also developed a cycle-based simulator called CaravelaSim in order to test our prototype. This way we could run our Caravela's deployable prototype, but at the same time, test its scalability with thousands of nodes, all of this with the same Go's code base. CaravelaSim was developed to be efficient and realistic when testing with hundreds of thousands of nodes. It is also highly parameterizable, where we can change the number of nodes in the system, the rate of requests, the profile of requests in terms of resources needs, and the duration of the simulation.

Then we implemented in our Caravela's prototype other two other approaches, Random and Swarm, to compare with our Multi-Offer approach. The Random approach is a naive and low overhead approach and Swarm is a centralized "oracle" approach. This way we could see where our approach situated between the two of them, and how well combined their strengths and avoided their weaknesses.

Finally, we provided Caravela's to our CaravelaSim using the three resource discovery approaches. We set a realistically test-bed for an Edge Cloud. In the simulator, we tested the three approaches under the exact same conditions. We concluded that the Swarm approach is perfect for networks with few hundreds of nodes, with near perfect scheduling enforcement, and constant number of hops for a container deployment. After that, the master node starts to become strained and the performance degrades or, in the limit, the system became unusable. The Random approach scales very well in terms of the loads per node (better than Multi-Offer), it distributes it very well, but at the cost of low success deployment rate and too many hops to discover the resources. Multi-Offer is the best of the two worlds as it maintains a very good scalability factor, at the cost of a bit more load in the nodes, distributed in a scalable way. It also has a deployment success rate near

Swarm and a low overhead (fewer hops) to find resources (compared with Random). The global scheduling policies enforcement is not so good as Swarm, but much better than Random that only provides by design a load balancing one.

6.1 Future Work

Here we provide a list of suggestions for improvements and extensions to Caravela's prototype and simulator. The list is ordered top-bottom from the suggestions/improvements we think are more interesting in our perspective:

1. Develop a mechanism to allow a user to specify a replication factor for a specific container, mimicking a feature that Docker Swarm already has, but in our decentralized architecture. This replication factor means that the system should maintain the number of containers running matching it.
2. Enhance Multi-Offer algorithm's fault resilience and availability by forcing the super traders to replicate their offers' tables into their successors. This way when a super trader crashes, the lookups to it, automatically goes to the successor (by Chord lookup design) that has a copy of its offers' table.
3. Bring Caravela to a real life deployment would be interesting in order to measure the same metrics we have in the simulator but now in a real deployment. We could even assess the deployed applications performance in order to test our deployment and scheduling decision.
4. Introduce fairness mechanisms like virtual currency and reputation system in the user manager component. These mechanism should also be scalable to not to result in a system's bottleneck.
5. Integrate Caravela's algorithms in the Docker Swarm code base in order to test with the real industrial set of tools.
6. Develop the container's storage backend using IPFS or BitTorrent and assess the scheduling performance.
7. Develop/search for an enhanced Chord's implementation that allowed the increase of the fingers density in order to decrease the number of messages in a Chord lookup. At the same time update the Chord's simulation to redo all the tests using the dense finger's table and check the memory cost, to compare it with the regular $log_2(N)$ entry size of the finger's table used in our basic implementation.
8. Look at the research work in Chord's (latency-aware) optimizations like node's joining placement depending on the underlying latencies between the nodes, in order to assess its gains in terms of real life deployment. This would be particular interesting since the Edge Cloud environment is susceptible to WAN latencies.

Bibliography

[1] P. Mell and T. Grance, “The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology,” *Nist Special Publication*, vol. 145, p. 7, 2011.

[2] Cisco Systems, “Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are,” *White Paper*, p. 6, 2016. [Online]. Available: http://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf

[3] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, “A container-based edge cloud PaaS architecture based on raspberry Pi clusters,” in *Proceedings - 2016 4th International Conference on Future Internet of Things and Cloud Workshops, W-FiCloud 2016*, 2016, pp. 117–124.

[4] 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 - ICDCN ’17*, 2017, pp. 1–10.

[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC ’12*, p. 13, 2012.

[6] 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.

[7] Cisco Systems, “Cisco Fog Computing Solutions : Unleash the Power of the Internet of Things,” *White paper*, pp. 1–6, 2015.

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

[9] M. Satyanarayanan, P. Bahl, R. Cáceres, and N. Davies, “The case for VM-based cloudlets in mobile computing,” *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[10] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, “SETI@home: an experiment in public-resource computing,” *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[11] V. D. Cunsolo, S. Distefano, A. Puliafito, and M. Scarpa, "Cloud@Home: Bridging the gap between volunteer and cloud computing," in *Lecture Notes in Computer Science*, vol. 5754 LNCS, 2009, pp. 423–432.

[12] A. M. Khan, F. Freitag, and L. Rodrigues, "Current trends and future directions in community edge clouds," in *2015 IEEE 4th International Conference on Cloud Networking, CloudNet 2015*, 2015, pp. 239–241.

[13] G. Briscoe and A. Marinos, "Digital ecosystems in the clouds: Towards community cloud computing," in *2009 3rd IEEE International Conference on Digital Ecosystems and Technologies, DEST '09*, 2009, pp. 103–108.

[14] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172, 2015.

[15] K. Kumar and M. Kurhekar, "Economically Efficient Virtualization over Cloud Using Docker Containers," *Proceedings - 2016 IEEE International Conference on Cloud Computing in Emerging Markets, CCEM 2016*, pp. 95–100, 2017.

[16] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[17] C. Lages and G. Sim, "Economics-inspired Adaptive Resource Allocation and Scheduling for Cloud Environments," 2015.

[18] W. Zhu, C. L. Wang, and F. C. M. Lau, "JESSICA2: A distributed Java Virtual Machine with transparent thread migration support," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, vol. 2002-Janua, 2002, pp. 381–388.

[19] O. Babaoglu, M. Marzolla, and M. Tamburini, "Design and implementation of a P2P Cloud system," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, 2012, p. 412.

[20] P. Mayer, A. Klarl, R. Hennicker, M. Puviani, F. Tiezzi, R. Pugliese, J. Keznikl, and T. Bure, "The autonomic cloud: A vision of voluntary, Peer-2-Peer cloud computing," in *Proceedings - IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops, SASOW 2013*, 2014, pp. 89–94.

[21] M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed edge cloud for data intensive computing," in *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, 2014, pp. 57–66.

[22] H. Chang, A. Hari, S. Mukherjee, and T. V. Lakshman, "Bringing the cloud to the edge," in *Proceedings - IEEE INFOCOM*, 2014, pp. 346–351.

[23] N. Mohan and J. Kangasharju, "Edge-Fog cloud: A distributed cloud for Internet of Things computations," *2016 Cloudification of the Internet of Things, CIoT 2016*, 2017.

[24] A. Chandra and J. Weissman, “Nebulas : Using Distributed Voluntary Resources to Build Clouds,” *Proceedings of the 2009 conference on Hot topics in cloud computing*, vol. San Diego„ no. San Diego, CA, June 2009., p. 2, 2009.

[25] S. Choy, B. Wong, G. Simon, and C. Rosenberg, “A hybrid edge-cloud architecture for reducing on-demand gaming latency,” *Multimedia Systems*, vol. 20, no. 5, pp. 503–519, 2014.

[26] F. Costa, L. Veiga, and P. Ferreira, “Internet-scale support for map-reduce processing,” *Journal of Internet Services and Applications*, vol. 4, no. 1, pp. 1–17, 2013.

[27] U. Drolia, R. Martins, J. Tan, A. Chheda, M. Sanghavi, R. Gandhi, and P. Narasimhan, “The case for mobile edge-clouds,” in *Proceedings - IEEE 10th International Conference on Ubiquitous Intelligence and Computing, UIC 2013 and IEEE 10th International Conference on Autonomic and Trusted Computing, ATC 2013*, 2013, pp. 209–215.

[28] S. Singh and I. Chana, “A Survey on Resource Scheduling in Cloud Computing: Issues and Challenges,” *Journal of Grid Computing*, vol. 14, no. 2, pp. 217–264, 2016.

[29] N. Jafari Navimipour and F. Sharifi Milani, “A comprehensive study of the resource discovery techniques in Peer-to-Peer networks,” *Peer-to-Peer Networking and Applications*, vol. 8, no. 3, pp. 474–492, 2015.

[30] N. Jafari Navimipour, A. Masoud Rahmani, A. Habibizad Navin, and M. Hosseinzadeh, “Resource discovery mechanisms in grid systems: A survey,” *Journal of Network and Computer Applications*, vol. 41, no. 1, pp. 389–410, 2014.

[31] M. Cardosa and A. Chandra, “Resource bundles: Using aggregation for statistical large-scale resource discovery and management,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 8, pp. 1089–1102, 2010.

[32] A. Iamnitchi and I. Foster, “On Fully Decentralized Resource Discovery in Grid Environments,” *Grid Computing, GRID 2001*, vol. 2242, no. 1, pp. 51–62, 2001.

[33] T. Ghafarian, H. Deldari, B. Javadi, M. H. Yaghmaee, and R. Buyya, “CycloidGrid: A proximity-aware P2P-based resource discovery architecture in volunteer computing systems,” *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1583–1595, 2013.

[34] M. Hasanzadeh and M. R. Meybodi, “Grid resource discovery based on distributed learning automata,” *Computing*, vol. 96, no. 9, pp. 909–922, 2014.

[35] J. S. Kim, B. Nam, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman, “Resource discovery techniques in distributed desktop grid environments,” in *Proceedings - IEEE/ACM International Workshop on Grid Computing*, 2006, pp. 9–16.

[36] S. Kargar and L. Mohammad-Khanli, "Fractal: An advanced multidimensional range query lookup protocol on nested rings for distributed systems," *Journal of Network and Computer Applications*, vol. 87, pp. 147–168, 2017.

[37] A. S. Cheema, M. Muhammad, and I. Gupta, "Peer-to-peer discovery of computational resources for grid applications," in *Proceedings - IEEE/ACM International Workshop on Grid Computing*, 2005, pp. 179–185.

[38] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat, "Distributed Resource Discovery on PlanetLab with SWORD," in *WORDLS'04: Proceedings of First Workshop on Real, Large Distributed Systems*, no. 1, 2004.

[39] F. Messina, G. Pappalardo, and C. Santoro, "HYGRA: A decentralized protocol for resource discovery and job allocation in large computational grids," in *Proceedings - IEEE Symposium on Computers and Communications*, 2010, pp. 817–823.

[40] S. Basu, S. Banerjee, P. Sharma, and S. J. Lee, "NodeWiz: Peer-to-peer resource discovery for grids," in *2005 IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2005*, vol. 1, 2005, pp. 213–220.

[41] H. Papadakis, P. Trunfio, D. Talia, and P. Fragopoulou, "Design and implementation of a hybrid p2p-based Grid resource discovery system," in *Making Grids Work - Proceedings of the CoreGRID Workshop on Programming Models Grid and P2P System Architecture Grid Systems, Tools and Environments*, 2008.

[42] G. Kakarontzas and I. K. Savvas, "Agent-based resource discovery and selection for dynamic grids," in *Proceedings of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE*, 2006, pp. 195–200.

[43] T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[44] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID 2009*, 2009, pp. 124–131.

[45] F. Farahnakian, P. Liljeberg, T. Pahikkala, J. Plosila, and H. Tenhunen, "Hierarchical VM management architecture for cloud data centers," in *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, vol. 2015-Febru, no. February, 2015, pp. 306–311.

[46] C. C. Lin, P. Liu, and J. J. Wu, "Energy-aware virtual machine dynamic provision and scheduling for cloud computing," in *Proceedings - 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011*, 2011, pp. 736–737.

[47] F. Lucrezia, G. Marchetto, F. Risso, and V. Vercellone, "Introducing network-Aware scheduling capabilities in OpenStack," in *1st IEEE Conference on Network Softwarization: Software-Defined Infrastructures for Networks, Clouds, IoT and Services, NETSOFT 2015*, 2015.

[48] M. Selimi, L. Cerda-Alabern, M. Sanchez-Artigas, F. Freitag, and L. Veiga, "Practical Service Placement Approach for Microservices Architecture," in *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017*, 2017, pp. 401–410.

[49] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whalley, and E. Snible, "Improving performance and availability of services hosted on IaaS clouds with structural constraint-aware virtual machine placement," in *Proceedings - 2011 IEEE International Conference on Services Computing, SCC 2011*, 2011, pp. 72–79.

[50] A. M. Sampaio and J. G. Barbosa, "Dynamic power- and failure-aware cloud resources allocation for sets of independent tasks," in *Proceedings of the IEEE International Conference on Cloud Engineering, IC2E 2013*, 2013, pp. 1–10.

[51] E. Feller, L. Rilling, and C. Morin, "Snooze: A scalable and autonomic virtual machine management framework for private clouds," in *Proceedings - 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012*, 2012, pp. 482–489.

[52] F. Messina, G. Pappalardo, and C. Santoro, "Decentralised resource finding and allocation in cloud federations," in *Proceedings - 2014 International Conference on Intelligent Networking and Collaborative Systems, IEEE INCoS 2014*, 2014, pp. 26–33.

[53] E. Feller, C. Morin, and A. Esnault, "A case for fully decentralized dynamic VM consolidation in clouds," in *CloudCom 2012 - Proceedings: 2012 4th IEEE International Conference on Cloud Computing Technology and Science*, 2012, pp. 26–33.

[54] F. Hendrikx, K. Bubendorfer, and R. Chard, "Reputation systems: A survey and taxonomy," *Journal of Parallel and Distributed Computing*, vol. 75, pp. 184–197, 2015.

[55] P. D. Rodrigues, C. Ribeiro, and L. Veiga, "Incentive mechanisms in peer-to-peer networks," in *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW 2010*, 2010.

[56] H. Zhao and X. Li, "VectorTrust: Trust vector aggregation scheme for trust management in peer-to-peer networks," *Journal of Supercomputing*, vol. 64, no. 3, pp. 805–829, 2013.

[57] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," *www.bitcoin.org*, p. 9, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[58] K. Chard, S. Caton, O. Rana, and K. Bubendorfer, "Social Cloud: Cloud computing in social networks," in *Proceedings - 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD 2010*, 2010, pp. 99–106.

[59] V. Vishnumurthy, S. Chandrakumar, and G. Emin, “Karma: A secure economic framework for peer-to-peer resource sharing,” *In Workshop on Economics of Peer-to-peer Systems*, vol. 34, 2003.

[60] P. Oliveira, P. Ferreira, and L. Veiga, “Gridlet economics: Resource management models and policies for cycle-sharing systems,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6646 LNCS, pp. 72–83, 2011.

[61] D. Ongaro and J. K. Ousterhout, “In Search of an Understandable Consensus Algorithm,” *Atc ’14*, vol. 22, no. 2, pp. 305–320, 2014.

[62] R. Baig, R. Roca, F. Freitag, and L. Navarro, “Guifi.net, a crowdsourced network infrastructure held in common,” *Computer Networks*, 2015.

[63] J. Benet, “IPFS-Content Addressed, Versioned, P2P File System,” *IPFS-Content Addressed, Versioned, P2P File System*, no. Draft 3, 2014. [Online]. Available: http://arxiv.org/abs/1407.3561

[64] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, “The Bittorrent P2P file-sharing system: Measurements and analysis,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3640 LNCS, 2005, pp. 205–216.

[65] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup protocol for Internet applications,” *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.

[66] D. Karger, T. Leightonl, D. Lewinl, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web,” *STOC ’97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997.

[67] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala, “Locality-preserving hashing in multidimensional spaces,” *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing - STOC ’97*, 1997.

[68] J. J. Dongarra, P. Luszczek, and A. Petite, “The LINPACK benchmark: Past, present and future,” *Concurrency Computation Practice and Experience*, 2003.

[69] A. Rowstron and P. Druschel, “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems,” *IFIPACM International Conference on Distributed Systems Platforms Middleware*, vol. 11, no. November 2001, pp. 329–350, 2001.

[70] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, “Tapestry: A resilient global-scale overlay for service deployment,” *IEEE Journal on Selected Areas in Communications*, 2004.

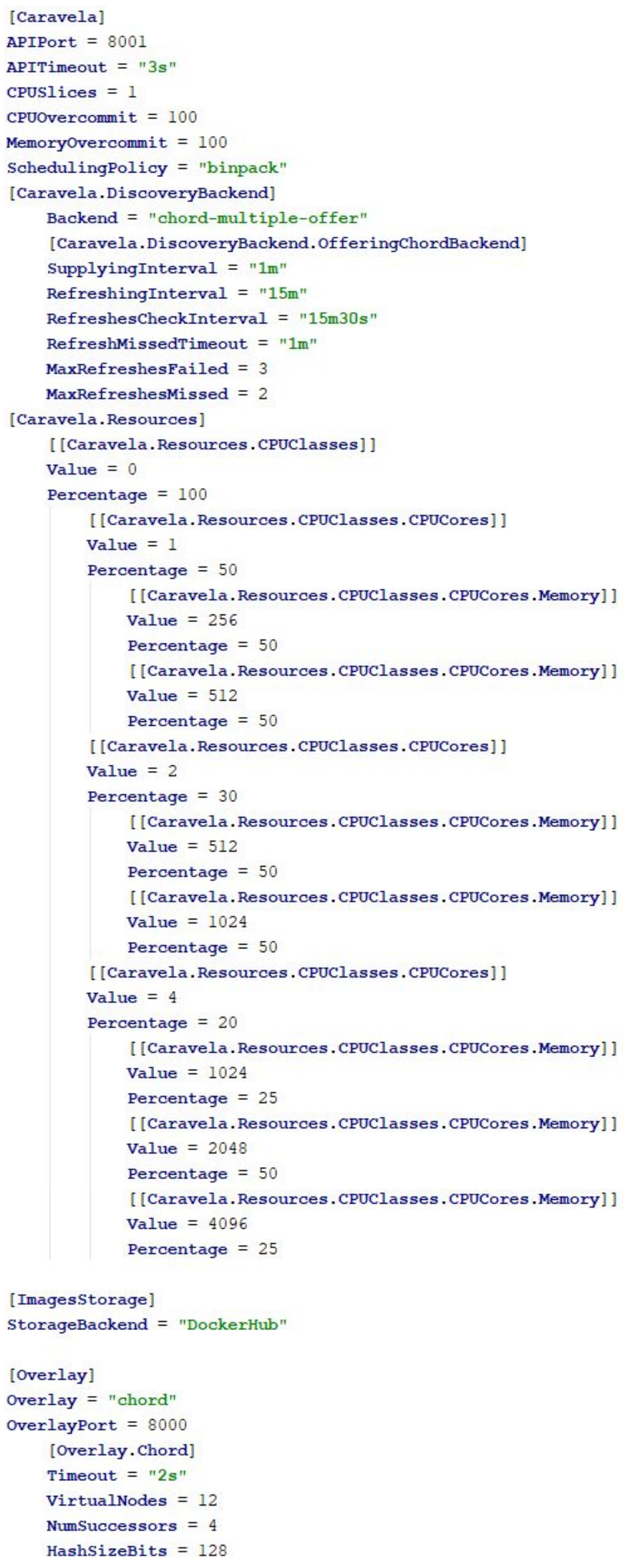
[71] P. Maymounkov and D. Mazieres, “Kademlia: A peerto -peer information system based on the xor metric.” *Iptps02*, 2002.

[72] 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.

[73] A. Montresor and M. Jelasity, “PeerSim: A Scalable P2P Simulator,” in *IEEE P2P’09 - 9th International Conference on Peer-to-Peer Computing*, 2009.

A

Appendix A - Node’s Configuration File



```
[Caravela]
APIPort = 8001
APITimeout = "3s"
CPUSlices = 1
CPUOvercommit = 100
MemoryOvercommit = 100
SchedulingPolicy = "binpack"
[Caravela.DiscoveryBackend]
    Backend = "chord-multiple-offer"
    [Caravela.DiscoveryBackend.OfferingChordBackend]
    SupplyingInterval = "1m"
    RefreshingInterval = "15m"
    RefreshesCheckInterval = "15m30s"
    RefreshMissedTimeout = "1m"
    MaxRefreshesFailed = 3
    MaxRefreshesMissed = 2
[Caravela.Resources]
    [[Caravela.Resources.CPUClasses]]
    Value = 0
    Percentage = 100
        [[Caravela.Resources.CPUClasses.CPUCores]]
        Value = 1
        Percentage = 50
            [[Caravela.Resources.CPUClasses.CPUCores.Memory]]
            Value = 256
            Percentage = 50
            [[Caravela.Resources.CPUClasses.CPUCores.Memory]]
            Value = 512
            Percentage = 50
        [[Caravela.Resources.CPUClasses.CPUCores]]
        Value = 2
        Percentage = 30
            [[Caravela.Resources.CPUClasses.CPUCores.Memory]]
            Value = 512
            Percentage = 50
            [[Caravela.Resources.CPUClasses.CPUCores.Memory]]
            Value = 1024
            Percentage = 50
        [[Caravela.Resources.CPUClasses.CPUCores]]
        Value = 4
        Percentage = 20
            [[Caravela.Resources.CPUClasses.CPUCores.Memory]]
            Value = 1024
            Percentage = 25
            [[Caravela.Resources.CPUClasses.CPUCores.Memory]]
            Value = 2048
            Percentage = 50
            [[Caravela.Resources.CPUClasses.CPUCores.Memory]]
            Value = 4096
            Percentage = 25

[ImagesStorage]
StorageBackend = "DockerHub"

[Overlay]
Overlay = "chord"
OverlayPort = 8000
    [Overlay.Chord]
    Timeout = "2s"
    VirtualNodes = 12
    NumSuccessors = 4
    HashSizeBits = 128
```

Figure A.1: Caravela's TOML configuration file

B

Appendix B - Single Offer Discovery Strategy

The second and last approach we developed for the resources discovery is called **single offer** because each node will only **advertise (via offer) its resources for one trader only**. Summing up, at a single moment in the Caravela operation the number of offers in all the traders are equal to the number of nodes participating in the system. As we describe in the following sections this changes the supply and find algorithm at some extent impacting the performance in terms of discovery efficiency and efficacy.

B.1 Resource Supplying

In this discovery strategy the supplier when publishing its available resources to the regions, it will only send its offer to one resource region. This region will be the fittest region (region with less resources but that can handle the requested) for its available resources. Each node is only registered in the highest resource region where its free resources can handle the requests. This strategy compared with the multiple offer creates a lot less offers in the system, meaning there is less overhead managing them.

Algorithm B.1 describes the resource supplying algorithm for the single offer approach. When the supplying algorithm runs due to a local container(s) start/stops (consuming/releasing node's resources) the supplier verifies if the current available resources are still the fittest for the resource region where it has the current offer (line 2). If the new resource availability are still enough for the resource region where it has the current offer, the supplier sends an `UpdateOffer` message to the offer's responsible trader in order to update the offer's free and used resources (lines [3-5]). After that the algorithm returns. If the resource availability of the supplier changed so much that it can now offer its resources in a higher region, or that it can only offer in a lower region, than it removes the offer from the trader (line 7). Lastly, with the previous offer removed, it creates a new offer (with the current resource availability) in the highest/fittest region that it can in that moment.

Algorithm B.1: Single Offer - Supplier's resources supplying algorithm.

```
   Data: $suppliersOffersMap$, $resourcesRegionMapping$, $remoteClient$
1  Function SupplyResources($freeResources$, $usedResources$):
2      $currentOffer \leftarrow suppliersOffersMap.CurrentOffer()$
3      if $resourcesRegionMapping.SameFittestRegion(freeResources, currentOffer.FreeResources)$ then
4          $newOffer \leftarrow NewOffer(currentOffer.ID, freeResources, usedResources)$
5          $remoteClient.UpdateOffer(currentOffer.TraderIP, newOffer)$
6          return
7      else
8          $suppliersOffersMap.Remove(currentOffer.ID)$
9          $remoteClient.RemoveOffer(currentOffer.TraderIP, LocalNodeInfo(), currentOffer)$

       /* Algorithm 3.2 describes how an offer is created in the system.    */
       $fittestRegion \leftarrow resourcesRegionMapping.FittestRegion(freeResources)$
10     $CreateOffer(freeResources, usedResources, fittestRegion)$
11     return
```

B.2 Resource Discovery

Algorithm B.2: Single Offer - Resource discovery algorithm.

```
Data: $resourcesRegionMapping$, $chord$, $remoteClient$
1  Function DiscoverResources($resourcesNeeded$):
2      $suitableOffers \leftarrow \varnothing$
3      $destTraderGUID \leftarrow resourcesRegionMapping.RandomGUIDDiscover(resourcesNeeded)$
4      $traderIP \leftarrow chord.Lookup(destTraderGUID)$
5      $suitableOffers \leftarrow remoteClient.GetOffers(traderIP)$
6      while $suitableOffers = \varnothing$ do
7          $destTraderGUID \leftarrow$
             $resourcesRegionMapping.RandomGUIDNextRegion(destTraderGUID, resourcesNeeded)$
8          if $destTraderGUID$ = NIL then
               /* This means that we exhausted all the regions where to look for the
               resources needed.                                                    */
9              return $\varnothing$
10         $traderIP \leftarrow chord.Lookup(destTraderGUID)$
11         $suitableOffers \leftarrow remoteClient.GetOffers(traderIP)$
12     return $suitableOffers$
```

Algorithm B.2 describes the algorithm to find resources using the single offer approach. The algorithm starts by looking for trader in the fittest region for the resources needed (lines [3-5]). If it found at least one offer, it returns (line 12), otherwise it will search in every regions that are higher than the fittest region (lines [6-11]). The rationale is simple, if it cannot find a node with resources free, similar to the resources needed, it tries to search for nodes with much more resources free, because it wants to satisfy the user's request. If it cannot find any offer until the highest region is reached the algorithm exits returning a empty set of offer (lines [8-9]), which eventually will bubble up to the user in the form of a deployment error.

B.3 Optimizations

In this section we introduce several optimizations that can be used in the single/multiple offer discovery approaches, but they are much more useful to the single offer discovery as we will see.

B.3.1 Advertise Offers Protocol

When we search in a nodes' region for offers, we select a random node as we explained above. This has the following problem: if the node that was selected does not have offers we wasted a Chord's lookup (which is costly, log_2N, with N being the network size) and we will do a retry that inject even more messages in the system. The success rate of finding traders that have offers is low, this is remarkably true in the single offer strategy due to the fact of only having one offer per supplier, which is equal to one offer per node. This optimization is fundamental when there are few resources in the system which makes the offers more scattered and harder to find efficiently while maintaining scalability and efficiency in the search for them.

In order to increase the chances of finding a node with offers we developed a protocol called **local offers advertisement**. This protocol runs in all nodes. Whenever an empty trader (trader without offers registered) registers an offer it automatically sends two asynchronous `AdvertiseOffer` messages to its two neighbors in the chord, the predecessor and successor. These messages contains its IP address and means that the Trader sending it has offers registered (at least one). The nodes that receive these messages continue to pass the message into the following nodes in the ring until a trader that receives them also has at least one offer registered. **All traders** have two "pointers" (IP addresses) for the closest traders in the ring that have offers. If one empty trades receive a `GetOffers` message it will send two `GetOffers` direct messages (with the IP addresses saved) at same time for the two "pointers" (traders that may have offers registered) aggregating the results of these messages, and returning them. This advertisement of offers through the ring is concealed inside each resource region. If the next trader, that must receive the `AdvertiseOffer` message, is from other resource region the `AdvertiseOffer` offer message is not sent. With this protocol we try to harvest the maximum number of offers in a region without using the Chord again, we only have two additional direct messages to the closest traders instead of a whole Chord lookup. Figure B.1 pictures a simple example of the protocol. Green nodes/traders have offers and the red ones do not. The arrows are the `AdvertiseOffer` messages and the number is the GUID from the node that is advertising the offers, the message also contains the IP address of the advertising node. Inside each node we have the node's GUID on top, the **N** is the pointer for the successor node with offers and **P** is the pointer to the predecessor node with offers.

When one Trader sends a `GetOffers` messages to one of the "pointers" trader, and it replies with an empty set of offers (because they could be already used) it invalidates the pointer in order to not use it again.

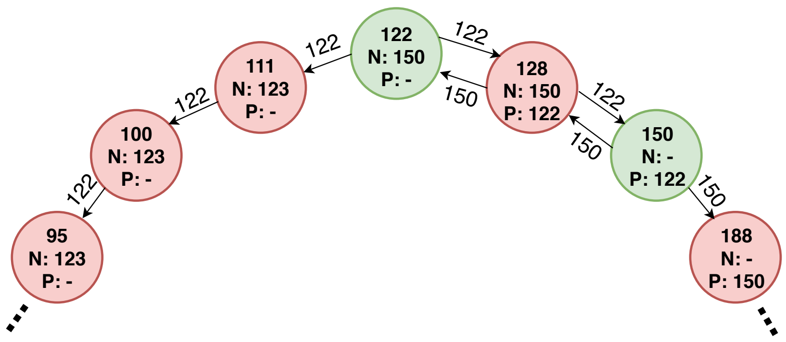


Figure B.1: Nodes running the local offers advertisement protocol.

B.3.2 Global Regions State, Gossip Algorithm

This protocol attacks the same problem as the local offers advertisement protocol B.3.1, targeting traders with no offers, but in a different way. When we have regions with the majority of traders empty, finding offers is very difficult, which makes the discovery process long and with many messages necessary. The problem is that each node does not have a global view of the regions state (if the region's traders have many offers or not), which is easy to implement in a centralized system. In order to try to make each node aware of the regions state in terms of amount of offers, the following protocol was developed: whenever a node/supplier sends a `GetOffers` message to a trader in a region it saves in a local map if it was a hit (trader had offers) or a fail (trader had no offers). In the node's local map each region has a integer that is incremented on hits and decremented on misses. This integer has a maximum value configured in the Caravela's bootstrap (so all the nodes have the same). The main part of this protocol consists in pig-backing the node's local map of the regions state in every messages (discovery messages, scheduling messages, etc) exchanged between nodes. When a node receives a message, before processing it, it merges the map in the message with its local map. This way at cost of slightly larger messages we can exchange the regions state for free in terms of number of messages exchanged in the system.

The information in the local regions state map is used by the suppliers when searching for offers. Algorithm B.3 describes the extension of the algorithm B.2 to support the "global" view of the resource regions, when searching for resources. The regions state map does not hold the perfect view of the state only the knowledge that the node has from using the system, and the knowledge from the interacting with it. When we want to search in a region we verify in the regions state map if the probability to find the resources is that region is high or not (line 17), depending on the misses and hits registered for it. With this being a probabilistic choice over imperfect data we will have false-positives and false-negatives, which will make us look in a region where there are few resources (false-positive), or not search in a region where we had a decent probability to find them (false-negative). Tuning the threshold parameter to decide when try or not the region is fundamental to minimize the errors.

After a good fine tuning, we could maximize the resource discovery efficiency by avoiding searches in resource regions with so few resources that we would target empty traders most of the times, while minimizing the probability of skipping resource regions where we had good probabilities to target a trader with offers.

Algorithm B.3: Single Offer - Resource discovering algorithm + regions state gossip optimization.

```
Data: $resourcesRegionMapping$, $regionsStateMap$, $chord$, $remoteClient$
1  Function DiscoverResources($resourcesNeeded$):
2      $suitableOffers \leftarrow \varnothing$
3      $destTraderGUID \leftarrow resourcesRegionMapping.RandomGUIDDiscover(resourcesNeeded)$
4      while $suitableOffers = \varnothing$ do
5          $traderIP \leftarrow chord.Lookup(destTraderGUID)$
6          $suitableOffers \leftarrow remoteClient.GetOffers(traderIP)$
7          if $suitableOffers$ != $\varnothing$ then
               /* Update the region state increasing its hits.                    */
8              $regionsStateMap.Hit(resourcesRegionMapping.Region(destTraderGUID))$
9              return $suitableOffers$
10         else
               /* Update the region state increasing its misses.                  */
11             $regionsStateMap.Miss(resourcesRegionMapping.Region(destTraderGUID))$
12         while $true$ do
13             $destTraderGUID \leftarrow$
                 $resourcesRegionMapping.RandomGUIDNextRegion(destTraderGUID, resourcesNeeded)$

14             if $destTraderGUID$ = NIL then
                   /* This means that we exhausted all the regions where to look for the
                   resources needed.                                               */
15                 return $\varnothing$
16             $targetRegion \leftarrow resourcesRegionMapping.Region(destTraderGUID)$
17             if $regionsStateMap.WorthTry(targetRegion) = true$ then
                   /* Region worth to try (probably it has many resources) so we break the
                   inner loop to the outside loop in order to do the search in the region.  */
18                 break
19     return $\varnothing$
```

C

Appendix C - Scheduling Policies Algorithms

Algorithm C.1: Weighting offers algorithm, adapted from Docker Swarm node weighting algorithm.

```
Function WeightOffers($availableOffers$, $resourcesNeeded$):
    $weightedOffers \leftarrow \varnothing$
    foreach $offer$ in $availableOffers$ do
        if $offer.freeResources.Lower(resourcesNeeded)$ then
            continue
        $supplierTotalCPUs \leftarrow offer.freeResources.CPUs + offer.usedResources.CPUs$
        $supplierTotalRAM \leftarrow offer.freeResources.RAM + offer.usedResources.RAM$
        $cpuScore \leftarrow 100$
        $ramScore \leftarrow 100$
        if $resourcesNeeded.CPUs > 0$ then
            $cpuScore \leftarrow$
              $(offer.usedResources.CPUs + resourcesNeeded.CPUs) * 100/supplierTotalCPUs$
        if $resourcesNeeded.RAM > 0$ then
            $ramScore \leftarrow$
              $(offer.usedResources.RAM + resourcesNeeded.RAM) * 100/supplierTotalRAM$
        if $cpuScore <= 100$ & $ramScore <= 100$ then
            $offer.weight \leftarrow cpusScore + ramScore$
            $weightedOffers \leftarrow Append(weightedOffers, offer)$
    return $weightedOffers$
```

Algorithm C.2: Binpack offer ranking and sort algorithm, adapted from Docker Swarm binpack algorithm.

```
Function Rank($availableOffers$, $resourcesNeeded$):
    $suitableOffers \leftarrow WeightOffers(availableOffers, resourcesNeeded)$
    $SortByWeightDesc(suitableOffers)$
    return $suitableOffers$
```

Algorithm C.3: Spread offer ranking and sort algorithm, adapted from Docker Swarm spread algorithm.

```
Function Rank($availableOffers$, $resourcesNeeded$):
    $suitableOffers \leftarrow WeightOffers(availableOffers, resourcesNeeded)$
    $SortByWeightAsc(suitableOffers)$
    return $suitableOffers$
```

D

Appendix D - Resources Combinations

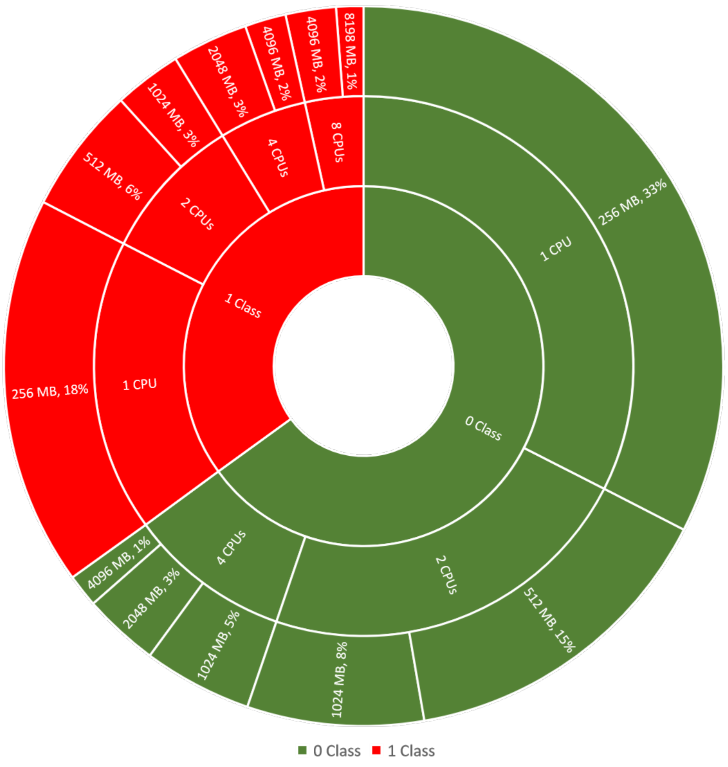


Figure D.1: Caravela's resource regions configurations used in evaluation.

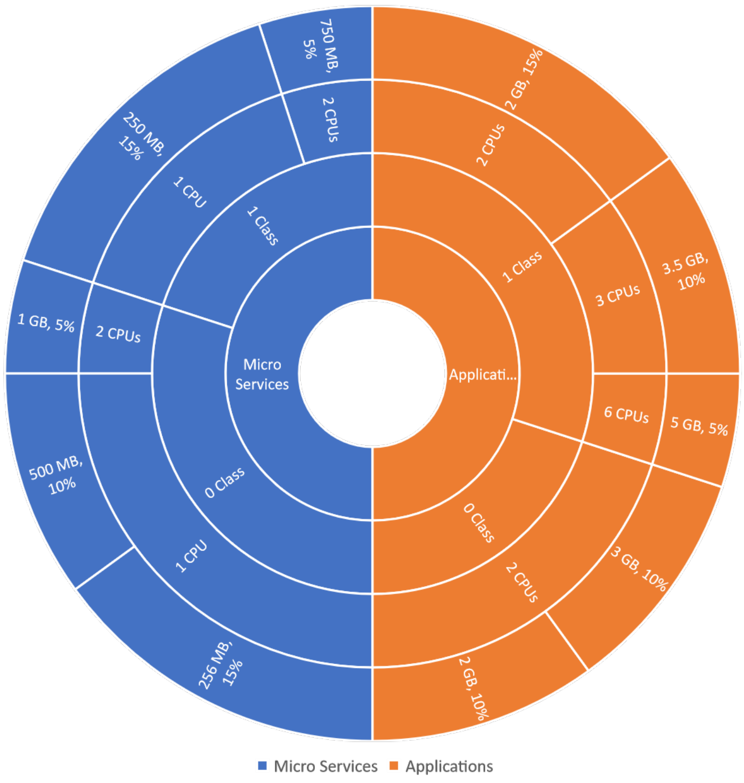


Figure D.2: Requests' profiles used in evaluation.