CARAVELA: A Cloud @ Edge

André Pires, n. 76046
pardal.pires@tecnico.ulisboa.pt

INESC-ID / Instituto Superior Técnico - Universidade de Lisboa
1049-001 Lisbon, Portugal

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 filled with a large amount of still under utilized resources. From these requirements and conditions, new computing paradigms like Edge Computing and Fog Computing are emerging.
This work presents CARAVELA[1] a cloud platform 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 tied to a centralized cluster environment deployment. So CARAVELA extends the Swarm platform employing a decentralized architecture and scheduling algorithm to cope with the volunteer-based, and hence volatile environment of the edge devices plus the wide area networks that connects them.

Keywords: Cloud Computing, Edge Computing, Fog Computing, Volunteer Computing, Edge Cloud, Resource Scheduling, Resource Discovery, Docker, Fairness

[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.

Table of Contents

1 Introduction

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.

Definition 1. *CISCO: Fog 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]*

Definition 2. *Vaquero et al: Fog 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]*

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 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 appearing in some literature [8], [9] that normally consists in cloud solutions to support mobile computing but not focused in RANs.

A graphical view of the computing paradigms is pictured in Figure 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 3.1.

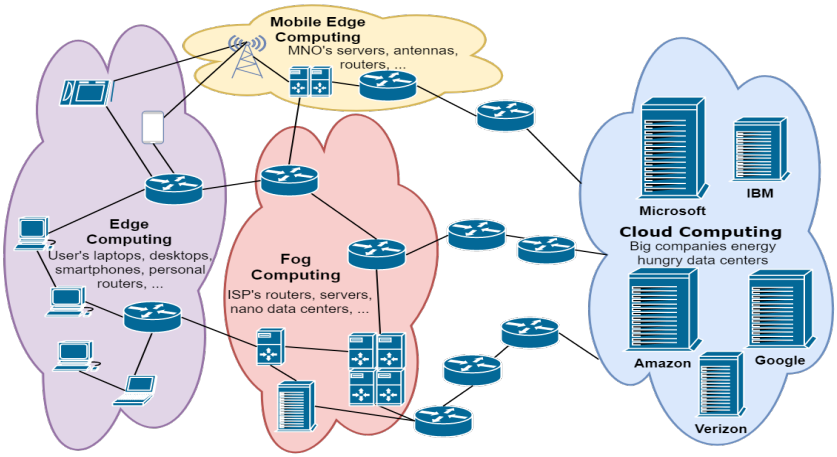


Fig. 1: Today's relevant computing paradigms and its resource owners

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 2. EC can be regarded in part as a sucessor 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.

Definition 3. *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, resource 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 widely spread suffering from big 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.
- **Fairness**: It should enforce fairness mechanisms to usage because the resources are contributed by multiple users.
- **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)[2].
- **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[3]) 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 provider deployment mechanisms.
- **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.
  - **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.

There are other possibilities, e.g. using OSGI framework[4] 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 scenario of the Edge Clouds compared to SVMs.

[2] Since offering the typical performance SLAs is difficult in a volunteer environment we consider SLAs to be only resources necessary to run the application.

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

[4] https://www.osgi.org/developer/architecture/

1.5 Current Shortcomings

Most of current Open Source solutions to cloud platforms: OpenStack[5], OpenNebula[6], Swarm[7] 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 Roadmap

The rest of the document is organized in the following way: Section 2 describes the main goals of our work. In Section 3, we present an analysis to the related work. Section 4 presents a solution architecture and the main protocols proposed. In Section 5, we describe how evaluate our solution in terms of system metrics, and what workloads will be used to exercise it. Lastly, Section 6 concludes the document and wraps up with the important marks.

2 Goals

Our fundamental contribution is the development of CARAVELA: a distributed and decentralized Edge Cloud that leverages volunteer resources from multiple users (e.g. laptops, workstations, spare RPIs, ...), allowing them to deploy their applications in 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/discovery resource algorithms in CC and fairness mechanisms in CC and Distributed Systems.
- The implementation of CARAVELA should consist in extending a middleware over the SWARM cluster management platform, offering:
  - **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 amount of CPU and RAM they need to deploy a container. It should be possible to deploy a set of containers that form an application stack, specifying if the user want the containers in the same node, promoting co-location, or spread them over different nodes.
- Experimental evaluation of the work in order to assess the feasibility and efficiency of our design (as described in Section 5).

3 Related Work

In this section we present the fundamental and state of art, academical and commercial, work in the development of Edge Clouds in Section 3.1, Cloud Resource Management in Section 3.2 and System Usage Fairness in Section 3.3. Lastly we have a presentation of Relevant Related Systems in Section 3.4.

[5] https://www.openstack.org/
[6] https://opennebula.org/
[7] https://docs.docker.com/engine/swarm/

3.1 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.

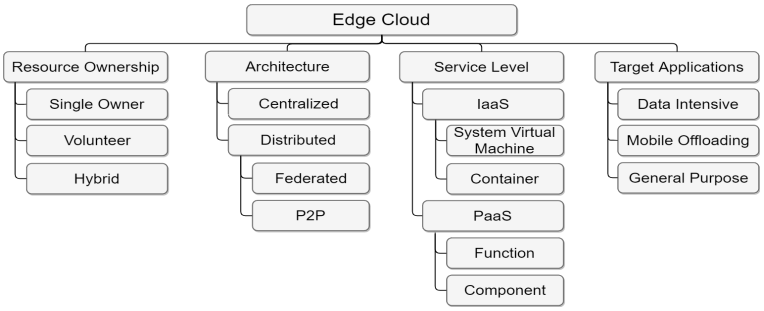


Fig. 2: Edge Clouds Taxonomy

Resource Ownership: It 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]).

Architecture: This characteristic 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]).

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[8] 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 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 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.

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

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

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.

Table 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.

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 1: Edge Cloud Works Classification

3.2 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. From here onwards in this document SVM, Virtual Machine (VM) and Container are used interchangeably to represent an IaaS deployable image.

3.2.1 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 RAM and CPU 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 3 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 centralized architecture.

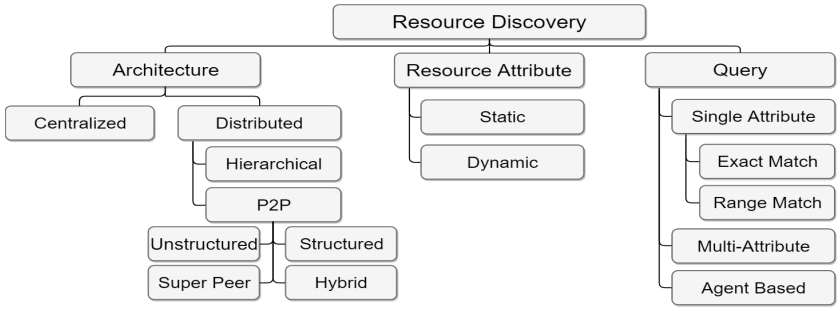


Fig. 3: Resource Discovery Taxonomy

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 + 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 drawbacks of bottleneck and 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, ...). 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.

Resource Attribute: 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 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.

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: *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 three CPU cores 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

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: Resource Discovery Works Classification

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

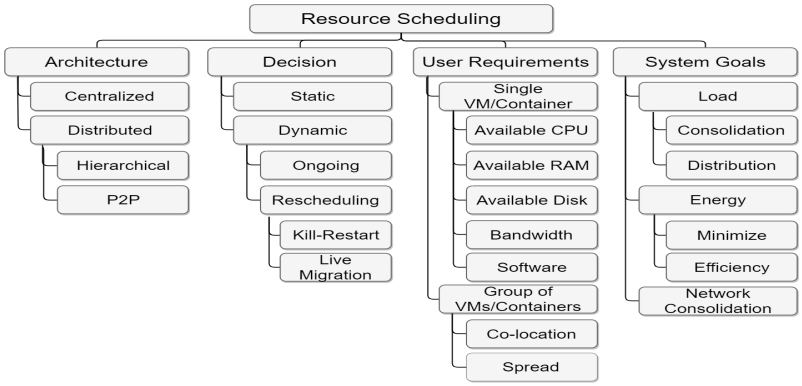


Fig. 4: Resource Scheduling Taxonomy

3.2.2 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 4.

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 underneath. There are two main types of architectures: Centralized and Distributed.

Centralized resource scheduling uses only 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: *Hierar-*

chical 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.

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 usage.

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.

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[9] 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

[9] 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 3: Resource Scheduling Works Classification

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.

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

3.3 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 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**. The Figure 5 pictures the taxonomy.

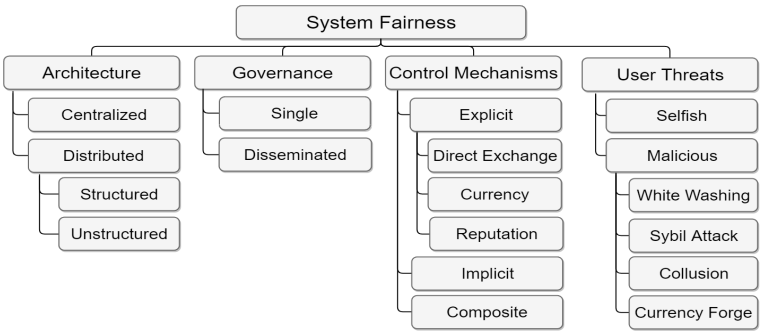


Fig. 5: System Fairness Taxonomy

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].

Governance: This 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[10]). Users tend easily accept systems that have a trustworthy entity controlling it. This entity

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

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.

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 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.

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. An 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.

System/Work	Architecture	Governance	Control Mechanisms	User Threats
SocialCloud [57]	Centralized	Central	Implicit (friends connections) & Currency	Currency Forge & Whitewashing
Karma [58]	Structured	Disseminated	Currency	Selfish, Currency Forge, Whitewashing & Sybil Attack
Gridlet Economics [59]	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 4: Fairness Works Classification

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 [60]) 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 4 lists the researched systems and its classification considering the taxonomy presented above. The last column represents the attacks that we know the system can mitigate.

3.4 Relevant Related Systems

Swarm[11] is an open source cluster manager where it is possible deploy Docker containers. It has a centralized architecture where a set of actively replicated nodes (called managers), using the Raft 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 central and consistent view of the cluster status. Due to this architecture, its scalability and fault tolerance are limited. The use of Raft

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

algorithm to maintain the replicated managers is not adequate for a wide area deployment of the system nodes. Moreover, none of these solutions are concerned with the system fairness since they assume a trustworthy environment. There are other similar open source solutions like OpenStack and OpenNebula that suffer from the same, or similar limitations due to a cluster like environment target.

Babaoglu et al. [19] present a prototype for a full P2P IaaS Cloud system. Their prototype uses a gossip protocol to maintain all the nodes connected without requiring any structure. 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.

4 Solution Proposal

In this Section we present CARAVELA, a distributed and decentralized Edge Cloud where it is possible to deploy standard Docker containers. Section 4.1 describes the distributed architecture. Section 4.2 presents the discovery and scheduling protocols that guide the containers deployment. Section 4.3 describes the data structures used. Section 4.4 describes the software that run in each node. Lastly, Section 4.5 describes some additional functionalities that we can consider after the base work is implemented. Our solution expects user's devices to have:

- An unmodified Docker engine and our extended Swarm middleware running.
- A client for a highly distributed file system (e.g. InterPlanetary File System (IPFS)[12] [62] or Bittorrent[13] [63]) 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 [60], in order to manage a virtual currency avoiding attacks like double spending.
- A client for a distributed and decentralized reputation system, like Karma [58], in order to maintain users' reputation. It is used to avoid accepting resources from users that after receiving the currency do not provide the promised resources.

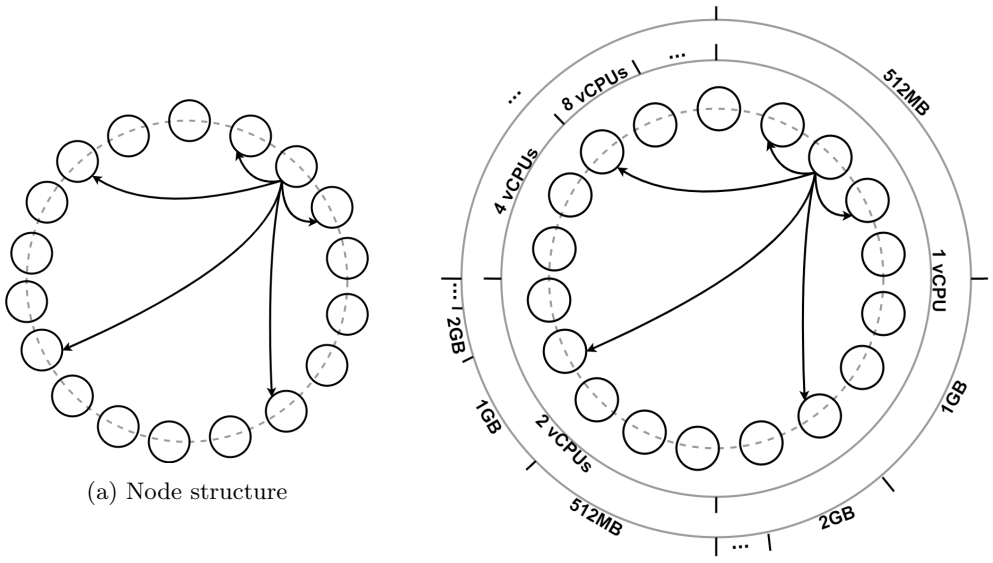
The last two clients are not really essential to demonstrate CARAVELA functionality. They are only, expected in a real life deployment to avoid abuse of the system (as Swarm also does not have these issues in mind, running within a dedicated cluster), and are outside the scope of our work.

4.1 Distributed Architecture

The distributed architecture consists in a ring structure that supports a DHT, similar to Chord [64] and Pastry [65] systems. Each user's own device corresponds to a node in the ring. In order to join the system, and start contributing to the Edge Cloud, a user only needs to know the IP address of a node that is present in the ring and pass it to our middleware in a join request.

Each node has a 256-bit Global Unique Identifier (GUID) that represents its position in the ring. In the joining process (similar to Chord and Pastry) a random GUID is generated to the new node. Each node has a neighborhood table (a.k.a finger table) that contains multiple pairs of GUID and IP from multiple nodes spread over the ring. Figure 6a pictures the architecture and the arrows represent the finger table entries. All the messages routed

[12] https://ipfs.io/
[13] http://www.bittorrent.com



(a) Node structure

(b) Resources distribution over the node structure

Fig. 6: CARAVELA's distributed architecture

through the ring have as destination a GUID, and they are routed in a similar way to Chord and Pastry using the finger table.

A user container's deployment request can be expressed in terms of resources as a pair, with the number of Virtual Central Processing Units (vCPUs) (similar to what EC2 does) and the amount of RAM necessary, e.g. $< 1vCPU, 2GB >$. The mapping between the real physical CPU power and the vCPUs is necessary to handle CPU hardware heterogeneity. In terms of disk space we assume that all nodes have a minimum space to hold the containers' images and some information from the applications.

The 256-bit GUID encodes a specific combination of vCPUs and RAM that a node can offer to deploy a single container. The distribution of the combinations over the GUID space is static for simplicity and scalability but not uniform, as represented in Figure 6b. The distribution is not uniform because we want to have more nodes handling resource combinations that have a greater demand. Another reason is that the combination of weaker resources like 1vCPU and 512MB of RAM can be offered by all the nodes while 4vCPU and 4GB of RAM can only be offered by a smaller set of more powerful nodes. Figure 7 pictures the GUID space.

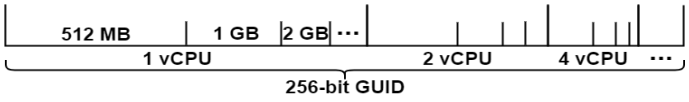


Fig. 7: GUID space

A node with a maximum capacity of 4vCPUs and 4GB of RAM can accommodate different combinations of requests, e.g. 4x $< 1vCPU, 1GB >$ or 2x $< 1vCPU, 4GB > +$ $< 2vCPUs, 2GB >$. We want to avoid that a node with 4vCPUs and 4GB of RAM of maximum capacity only offers a deployment of $< 4vCPUs, 4GB >$ for a container, because we want maximize the utilization of system resources. On the other hand, we want to have

still enough powerful nodes with capability to accommodate heavy requests when needed. This is a typical fragmentation problem, we handle it in Section 4.2.1.

4.2 Protocols

4.2.1 Resource Discovery With the resource combinations mapped in the ring structure, the discovery process is straightforward, which is important so that it has reduce cost and complexity for each node. Due to the DHT support it is decentralized hence scalable and churn resilient. The discovery mechanism is handled by six types of messages exchanged between the nodes: `offer`, `ackOffer`, `refresh`, `remove`, `search` and `noResources`. All these messages are generated by inter-node API calls with similar names. The full reference of CARAVELA's APIs is present in Appendix B.

When a node (a.k.a **Supplier**) has a resource offer, e.g. $< 1vCPU, 2GB >$, it creates an `offer` message that contains its IP, the amount of slots of the resource combination it offers, and it has as destination a randomly GUID generated inside the identifier zone that represents that combination of resources. The message is efficiently routed using the fingers table. The destination node (a.k.a **Trader**) saves the amount of slots it is offering, and the supplier's IP, thus acknowledging the offer. The trader uses the IP of the supplier to send a direct `refresh` message, with a T_r periodicity, in order to check whether the supplier is still alive and offering what it advertised, otherwise the supplier could be dead. If the refresh fails R_f times the trader removes the offer. If the trader fails to send R_m refresh messages the supplier re-advertises the offers into the ring again. This ensures liveness, garbage collects outdated information, while still ensuring completeness of accessibility to the resources in the long run. These are the most important aspects, as scalability, is defended by employing long enough periodicity and limited number of retries. If the supplier does not receive the acknowledge of the offer in a T_a period, it advertises it again into the ring. If by some reason a supplier advertises the same offer twice it will not acknowledge the offer for the second trader avoiding the existence of two traders managing the same offer. There is only one trader responsible for an offer in order to facilitate its management. Figure 8 pictures the nodes interactions of a single resource offering sequence.

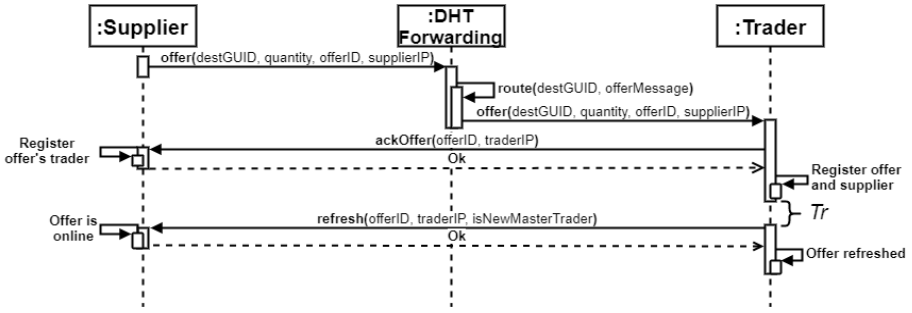


Fig. 8: Resources offering for a single container deployment

With a T_x periodicity the supplier nodes run an offering exchange algorithm that tries to mitigate the fragmentation problem. The idea is that with a given probability the supplier will maintain, coalesce or unfold the current offers, e.g. a node with a current offer of $< 8vCPUs, 16GB >$ will have 25% of probability to unfold that big offer into several smaller ones like 4x $< 2vCPUs, 4GB >$. These probabilities are defined in Table 5. In the specific case of unfolding the offer into smaller ones, the smaller combination is chosen probabilistically too, based on the size of the ring zones. This tries to balance the maximum utilization of the resources with the goal of ensuring that there are nodes that can handle

heavier requests, all once again in a fully decentralized, albeit probabilistic manner. The `remove` message is used by the supplier to remove an offering that it does not want to handle anymore. These messages are sent directly to the traders because the supplier has saved all the traders' IPs responsible for its offerings.

Offers State	Probability
Coalesce	25%
Maintain	50%
Unfold	25%

Table 5: Probabilities for a node to change its offers, a more stable conservative deploy would employ a higher maintain probability

Parameter	Description
T_r	Refresh periodicity (from trader to supplier) to check if a offer is still valid
R_f	Maximum failed refreshes to a supplier before removing the offer
R_m	Maximum refreshes missed by trader before the supplier re-advertises
T_a	Timeout for an offer being acknowledged by a trader
T_x	Periodicity of the offer exchange algorithm execution
K	Number of trader replicas
TTL	Maximum number of hops that a search message can travel
$GlobalPolicy$	Consolidate or Distribute

Table 6: System configuration parameters

In order to find a specific combination of CPU and RAM to deploy a container, the node (a.k.a **Buyer**) only needs to generate a `search` message with a randomly generated, hence decentralized, GUID as destination that belongs to the range section zone of the resources needed. It sends the image file key of each container's image and the respective resources necessary for the execution. This is done to save an additional message in the resource scheduling phase presented in the following section. The search message contains a TTL that is decremented in each hop in order to cancel the request if there are not any offers for the request. When the TTL expires the trader will send a `noResources` message to the buyer. Due to the random distribution of the offers in the ring, we use a retrying mechanism in order to try obtain the necessary resources. We anticipate that for the most dominant requests we will have no retries or a small number of them. All the nodes in the system have the three roles: **Supplier**, **Trader** and **Buyer** making it full P2P.

As we presented the discovery mechanism, if a trader fails, several offers will become unreachable, while the suppliers do not detect the trader failure, creating a window of unavailability for them. To compensate this, each trader (a.k.a Master Trader) will replicate the information of the offers it is responsible for in the K successor nodes (a.k.a **Replica Trader**), similar to what Chord does. The master is responsible for updating the replicas when there are changes in its offerings. If the master fails, the routing mechanism transparently sends the request to the successor that contains a replica. The replica trader knows that, if it received the request the master has died. This means that when a trader receives a request it must check if it is for a node that it is its master. It will become the master of those offers. The replica trader will contact all the respective suppliers (with a special `refresh` message) that were managed by the master, checking if all the offers are still valid and informing that it is the master now. With, this strategy we can have longer values for the refresh message periodicity T_r, avoid the need for multiple suppliers to re-advertise the

offers into the ring, decreasing the network load and increasing the availability of the nodes in order to contribute.

4.2.2 Resource Scheduling A container scheduling request that is submitted by a user into our middleware is a pair composed by: the pair $< vCPUs, RAM >$ presented above and a IPFS/BitTorrent file key of the container's image, e.g. $<< 2vCPUs, 4GB >, imageKey >$. The resource scheduling mechanism uses two types of messages exchanged between the nodes: `launch` and `started`.

The `search` message (from the resource discovery mechanisms) is built and sent when a user requests a container deployment, making the node that sends the request the **buyer** node for this request. When the message arrives to the respective trader node, if there are offers available it sends a `launch` message to the supplier, containing all the necessary information to launch the container (container image key, $< CPU, RAM >$, buyer's IP). The trader node removes the offer from the table when the supplier replies to the launch message with a confirmation. If the trader does not have any offer it sends the `search` message to a successor node that handles the same combination of resources, until the TTL expires. Asynchronously, the supplier will send a `started` message to the buyer node with the local container identifier and its IP address. The buyer node is responsible for saving this information in order to contact directly the supplier with `stop` containers messages for example, and to later access the container itself. Figure 9 describes the interactions to search and deploy a single container.

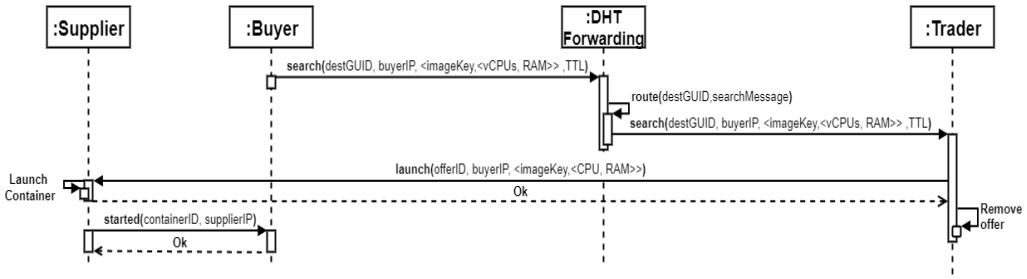


Fig. 9: Resource search and container launch

A user can also submit a composite request that contains a group of containers to deploy. This is useful in order to deploy a distributed application. The scheduling process is similar to a single container scheduling and it is transparent to the user. The only difference is that the `search` message contains a list of containers to deploy instead of a single one. When a trader launches a container, it will forward the message into the ring again in order to find a node that corresponds to the resources needed of the next container. This goes on until all the containers are launched.

A user can choose for the scheduling of a group of containers one of the two following policies: *co-location* and *spread*. A user can request that a set of containers must be placed in the same node (*co-location*) or spread over different nodes (*spread*). These policies make a scheduling request in reality being a pair, $< groupPolicy, List < imageKey, < vCPUs, RAM >>>$, that contains the group policy for the deployment and a list of pairs that contain the key for obtaining the container's image, and the respective resources combination necessary for the execution of the container.

We treat a *co-location* request for a group of containers as a larger request for a single container, e.g: 2x $< 2vCPUs, 4GB >$ with *co-location* enabled is the same as 1x $< 4vCPUs, 8GB >$. So, all the resource combinations that can satisfy the summing of

all the containers necessary resources are suitable for the deployment, for the previous example, the nodes with offers of $< 4vCPUs, 8GB >$ or $< 16vCPUs, 64GB >$ are suitable for the deployment. The choice for which combination we should send the request is a global system policy that we can configure. This global policy has two possible values: *Consolidate* and *Distribute*. The *Consolidate* policy favors the use of the resource combinations that left the small resource combination left in the node, while the *Distribute* policy favors nodes with a resource offer that have much more capability than the request needs. This is a decentralized approach inspired by the Swarm scheduling policies, further enhanced by being request-specific and not global to the complete network.

To achieve a *spread* deployment of a group of containers, we simple treat each pair on the list as a single deployment.

All the system's configuration parameters are listed in Table 6.

4.3 Data Structures

The following data structures are maintained by each node of the system:

- *Finger Table*: Hash table that contains the key-value pairs $<$ `GUID`, `IP` $>$ of O(log N) ring neighbors, with N being the network size with a maximum of 2^{256}.
- *Trader Table*: Hash table that contains the key-value pairs $<$ `quantitity`, `List` $<$ `offerID`,
  `supplierIP` $>>$ to save the offers that the node is responsible for.
- *Offers Table*: Hash table that contains the key-value pairs $<$ `traderIP`, `List` $<$ `offerID` $>>$ of traders that have the node's own offers.
- *Containers Table*: Hash table with the key-value pairs $<$ `containerID`, $<$ `buyerIP`, $<$ `vCPUs, RAM` $>>>$ for each container launched from CARAVELA's middleware. The field `containerID` unequivocally identifies the container in the respective node. The value is a pair with the IP of the node that launched the container and the resources that were allocated to the container execution.

4.4 Software Architecture

We now present the software components of CARAVELA that run in each node, what are their responsibilities and interactions. A components diagram is pictured in Figure 10.

Membership Manager: component that deals with the construction/maintenance of the ring overlay. It maintains the *finger table*.

Resource Discovery: component responsible for advertising the node's offers into the ring, manage the offers from other nodes and find resources for a request. It manages the *offers table* and the *trader table*.

Resource Scheduling: component responsible for deploying the user's request into the system, using the resource discovery module.

User Management: component that offers authentication of users in the node. It interfaces with external currency and reputation clients in order to manage user's currency and reputation. It provides an internal interface that verifies if the user has enough credits to submit a given request and how trustworthy is the user.

Containers Manager: manages only the Docker containers that are launched from the CARAVELA middleware. It offers an interface to know how much resources from the node are being used by each container. It manages the *containers table*.

Node Manager: this component is a façade component that transforms the calls to CARAVELA's API into calls of each module described above.

The Resource Scheduling API, Resource Discovery API and Membership API represented in Figure 10 are exposed to the other nodes via REST APIs.

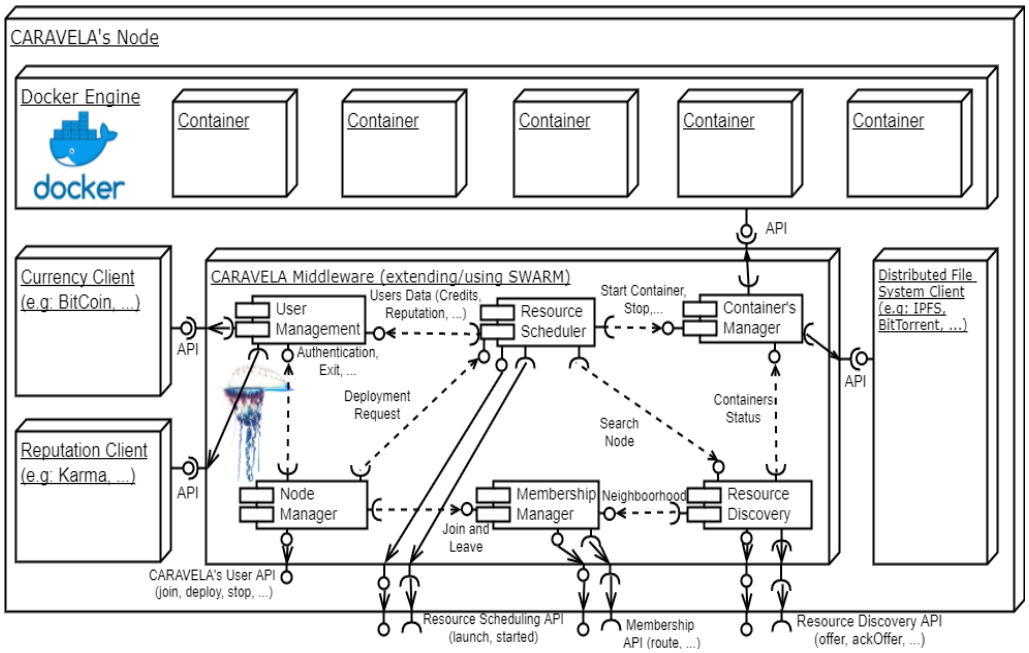


Fig. 10: CARAVELA's Components

4.5 Additional Work

The current Swarm implementation offers a way to specify a factor of replication for container deployment, e.g. a user can specify in a container deployment request that there must always be 5 replicas of that container present and running. Since this is something usual in the Swarm world, after our base work (presented above) is implemented, we will consider this feature in our distributed and decentralized implementation.

5 Evaluation Methodology

Here we present the system's metrics, in Section 5.1, and the workloads, in Section 5.2, that we intend to use during the evaluation phase of our prototype.

5.1 Metrics

The following list represents the main performance metrics that we will consider during the evaluation phase of our prototype development:

- *Scheduling Delay:* This metric assesses how much time will pass between the submission of a deployment request and the container starting running in the system at the destination node.
- *Allocation Success Rate:* This metric is very important in order to assess how effectively and useful our resource discovery mechanism is. It is fundamental because a low success rate here implies a huge overhead in the system (due to the retries) affecting system's scalability. This intends to assess how the fact of adopting a fully decentralized approach (out of a cluster like environment) may affect the ability of effectively discovering, allocating, and otherwise manage the resources.
- *Effect on Workloads:* This metric assesses if our system provides the suitable resources for the containers execution, its correctness to some extent. In the end, this will assess how good is our scheduling strategy providing the best nodes to run the workloads.

- *Resource Utilization-Allocation:* This metric is a ratio that assesses how the system handles the fact that containers may not consume exactly the resources asked in the request. The use of overbooking strategies, while managing the containers, is crucial to obtain the maximum potential of a given node. In essence, this assesses how efficient our system is in making use of the resources provided by the users.
- *Space Overhead of Data Structures:* Essential to know if the space used by the systems data structures is sustainable by each node. This will be used to find out if the information necessary to maintain the system is truly distributed by all the nodes, in order to assess its scalability, regarding the amount of information necessary when the system increases in number of nodes.

5.2 Workloads

The following approach will be used to exercise the system in the presence of different types of workloads needs, in order to know how the system management influences applications with different needs. The metrics presented in the previous section will be measured during these workloads executions. The workloads are:

- *PlanetLab's [66] Traces*: These are real traces (that come in the CloudSim [67] distribution) from a worldwide scientific/community network called PlanetLab that is used to test network applications, like P2P systems.
- *Containers running different kinds of workloads inside them, that have different kinds of of application profiles and hence resource requirements: e.g. CPU intensive, and different types of execution timespan, e.g. jobs with limited durations or services that run indefinitely.*
  - *CPU Intensive (Jobs):* FFMPEG[14] workloads like converting/encoding media files.
  - *Memory Intensive (Services):* Redis[15] an open source database manager.
  - *CPU and Memory Intensive (Job):* Deep Learning workloads[16]
  - *Non Intensive (Service)*: Timeservers[17]

6 Conclusion

Our work presented CARAVELA, a proposal for a fully distributed and decentralized Edge Cloud that allows the deployment of standard Docker containers. We started by describing the current Fog Computing and Edge Computing paradigms that instigate the appearance of Edge Clouds. Next, we have settled the goals of our work. After our research throughout the current open source cloud systems and scientific literature, we presented taxonomies for the Edge Clouds works, resource management in Cloud Computing and System Fairness in Volunteer Computing systems. The following section, introduced CARAVELA with the respective architecture and algorithms. The document ended with the evaluation methodology to assess the future implementation of CARAVELA.

[14] https://www.ffmpeg.org/
[15] https://redis.io/
[16] https://hub.docker.com/r/alexjc/neural-enhance/
[17] https://hub.docker.com/r/sergiomendes/timeserver/

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.

[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 (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 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*, 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, 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] 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.

[58] V. Vishnumurthy, S. Chandrakumar, and G. Emin, "Karma: A secure economic framework for peer-to-peer resource sharing," *... Economics of Peer-to ...*, 2003.

[59] 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.

[60] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," *Www.Bitcoin.Org*, p. 9, 2008.

[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] J. Benet, "IPFS-Content Addressed, Versioned, P2P File System," *IPFS-Content Addressed, Versioned, P2P File System*, no. Draft 3, 2014.

[63] 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.

[64] 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.

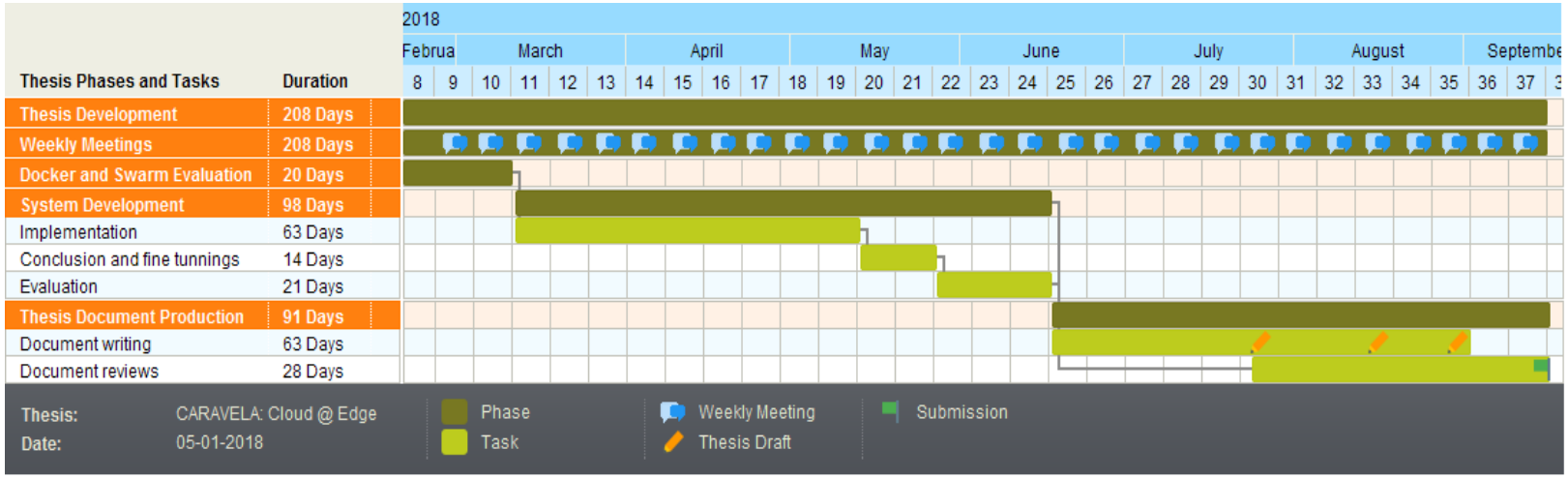
[65] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," *IFIPACM International Conference on Distributed Systems Platforms Middleware*, vol. 11, no. November 2001, pp. 329–350, 2001.

[66] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: an Overlay Testbed for Broad-Coverage Services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[67] 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.

A Planned Gantt Chart

The Gantt Chart presents the schedule of the main tasks and the respective deliverables, it will guide the work progress during the thesis development.



B CARAVELA's APIs Reference

All the APIs presented here will be implemented with REST services, except the user level API that probably, will have a Command Line Interface (CLI) and eventually a small Software Development Kits (SDKs) for programming languages.

B.1 User API

- `join(IPaddress)` This API call should be the first one in order to join CARAVELA. The `IPaddress` should be a IP address of node that as already succeeded in joining the system.
- `deploy(List<containerLocalImageFilePath,<vCPU, RAM>>, groupPolicy)` This API call is the basic one for a user to deploy containers in the system. The list contains all the containers images and the respective resources necessary for the execution. The `groupPolicy` corresponds to the user requirement for a group of containers. By default it is used the `spread` policy.
- `stop(containerID)` API call used to stop a container execution. `containerID` is the identifier for the container.
- `status(containerID)` API call used to know the status of deployed container: running, stopped and more. `containerID` is the identifier for the container.

B.2 Resource scheduling API

- `launch(offerID, buyerIP, <imageKey, <vCPUs, RAM>>)` This API call is used by traders to launch a container in a supplier. `offerID` argument is the local identifier of the offer. It is used by the supplier to know what offer is being used. `buyerIP` is the IP address of the buyer node. The last argument is the image key of the container image, in order to download it from the distributed file system, and the resources necessary for its execution.
- `started(containerID, supplierIP)` This API call is used by a supplier to inform a buyer about where its request is being executed. `containerID` is the local identifier of the container running. `supplierIP` is the IP address of the supplier where the container is running.

B.3 Resource discovery API

- `offer(destGUID, quantity, offerID, supplierIP)` This API call is used when a supplier wants to offer resources. `quantity` represents the number of offers, for that combination of resources, that the supplier is offering. `destGUID` is a random GUID that belongs to the zone of resource combination the supplier wants to offer. `offerID` is a local identifier used by the supplier to distinguish its own offers. `supplierIP` is the IP address of the supplier.
- `ackOffer(offerID, traderIP)` This API call is used by a trader to acknowledge a resource offer. `offerID` is the offer's supplier local identifier. `traderIP` is the IP address of the trader that is responsible for managing the offer.
- `refresh(offerID, traderIP, isNewTrader)` API call is used to verify if a supplier is alive and still offering what he advertised. `offerID` is the supplier local identifier for the offer. `traderIP` is the IP address of the trader that is sending the message. `isNewTrader` is `True` if this is a replica trader that assumed the place of the master trader responsible for the offer, it is `False` if it is still the master trader since the last refresh.
- `remove(supplierIP,offerID)` API call is used by a supplier to remove one of its offers from the respective trader. `supplierIP` is the IP address from the supplier that is removing the offer. `offerID` is the local identifier of the offer that is being removed.

- `search(destGUID, buyerIP, List<imageKey, <vCPUs, RAM>>, groupPolicy, TTL)` This API call is used by a buyer to find resources and at same time sending all the information to start the containers. `destGUID` argument is a random GUID that belongs to the region of resources necessary for the first container of the `List` argument. `buyerIP` is the IP address of the buyer, it is used to receive the deployment information later. The `List` of key-value pairs contains the keys for downloading the containers' images from the distributed file system, and the respective resources necessary for the execution of each container. The `groupPolicy` corresponds to the deployment requirement for a group of containers. TTL is the number of hops left that the search message can do.
- `noResources(searchMessage)` This API call is used by a trader to inform a buyer that its deployment request exceeded the TTL, because no suitable resources were found.

B.4 Membership API

- `route(GUID, message)` API call from the membership management component that is used to route the offer and search messages. `GUID` is the destination system GUID for the `message`.

Acronyms

CC Cloud Computing. 1–6, 8, 9, 12
CLI Command Line Interface. 31

DHT Distributed Hash Table. 9, 18, 20

EC Edge Computing. 1, 2

FC Fog Computing. 1, 2, 5, 6

GUID Global Unique Identifier. 18–21, 23, 31, 32

IaaS Infrastructure-as-a-Service. 6, 7, 18
IoT Internet of Things. 1
IPFS InterPlanetary File System. 18, 22
ISP Internet Service Provider. 5

JVM Java Virtual Machine. 3, 6

MEC Mobile Edge Computing. 1
MNO Mobile Network Operator. 1

OS Operating System. 3, 6

PaaS Platform-as-a-Service. 6

QoS Quality of Service. 8

RAN Radio Access Network. 1
RPI Raspberry PI. 1, 4

SaaS Software-as-a-Service. 6
SDK Software Development Kit. 31
SLA Service Level Agreement. 3, 11, 12, 14
SPoF Single Point of Failure. 4, 6, 8, 9, 11, 12, 15
SVM System-level Virtual Machine. 3, 6, 7, 12

TTL Time To Live. 8, 21, 22, 32

VC Volunteer Computing. 2, 5, 15, 16
vCPU Virtual Central Processing Unit. 19, 20, 22, 23, 31, 32
VM Virtual Machine. 7, 11, 12, 14

WAN Wide Area Network. 1