# Caravela: Cloud@Edge

A fully distributed and decentralized orchestrator for Docker containers

André Pires
Instituto Superior Técnico
Lisbon, Portugal
pardal.pires@tecnico.ulisboa.pt

## 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. Due to this, new computing paradigms like Edge Computing and Fog Computing are emerging.

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

## CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; **Cloud computing**; **Peer-to-peer architectures**;

## KEYWORDS

Cloud Computing, Edge Computing, Fog Computing, Volunteer Computing, P2P, Edge Cloud, Resource Scheduling, Resource Discovery, Docker

## 1 INTRODUCTION

Cloud Computing is a mature platform that gained its momentum due to its incredible advantages such as: resource elasticity, no upfront investment for the consumers (pay what you use, utility style), global access and more [10]. It is implemented with a set of geo-distributed energy hungry data centers at the Internet's backbone, which causes high latencies from the network's edge to the cloud, and it amplifies the possibility of having expensive bandwidth to reach it.

With the increase of IoT applications (as stated by CISCO [7]), among others, the network's edge is producing a lot of data, that is pushed to the cloud for processing and/or storage. The problem is that it is expensive in terms of bandwidth to upload everything to the cloud and for latency sensitive applications that need fast replies the cloud is far away. The increase of community networks (e.g. the GUIFI.net [2] with ≈35K nodes and a steady growth of 2k nodes/year) conjugated with the nowadays very powerful desktops, laptops and even RPIs the network's edge is filled with a lot of resources that most of the time are under utilized. The Edge and Fog Computing intend to leverage these resources to provide services that are near the Internet's edge. These new terms are not well defined yet so we present, CISCO's and Vaquero et al. definitions that use these two new terms interchangeably:

*Definition 1.1.* CISCO [4]: 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.

*Definition 1.2.* Vaquero et al [14]: 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.

CISCO's definition (Def. 1.1) for Fog/Edge Computing clearly mentions the space between the end user's devices and the cloud where a platform (potentially developed and backed up by their solutions) could offer compute, storage and network capabilities much closer to the user's offering faster services and mediating the communication with the cloud. Vaquero et al. definition (Def. 1.2) is more specific than CISCO's, it describes a completely decentralized platform where the **user's own devices** cooperate to offer the compute, storage and network capabilities using sandboxed environments, which makes it feel like the past Volunteer Computing works like SETI@Home [1].

Our work follows the Vaquero's definition line environment. Our contribution consists in Caravela, a Docker container's orchestrator, similar to Docker Swarm[3], but enhanced in order to be used as an Edge Cloud platform. Our work targets the environment of fog/edge computing where the amount of devices available is large, heterogeneous, connected via wide area networks, churn is presented and there is a need to impose fairness rules to the users. This requires a distributed and decentralized architecture, discovery and scheduling algorithms to cope with the amount of nodes and users.

[1] Caravela (a.k.a *Portuguese man o'war*) is a colony of multi-cellular organisms that barely survive alone, so they need to work together in order to function like a single viable animal.
[2] Code available at https://github.com/Strabox/caravela

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

Due to the volunteer environment, a decentralized solution is recommended since there is no natural central administration. Current Docker Swarm implementation targets small clusters (maximum of 3k nodes as stated by its creators) of homogeneous machines in a controlled environment used by a limited set of users where a centralized solution is viable.

The rest of the paper is structured as follows. Section 2 describes briefly the fundamental and state of the art works in the Edge Clouds, Resources Management and usage Fairness. Section 3 presents the architecture and the resource discovery and scheduling algorithms that compose Caravela. Section 4 presents the evaluation to our Caravela's prototype comparing its performance with an adaptation of the Docker's Swarm and a naive random-based approach. Finally, Section 5 wraps up the paper with our main conclusions.

## 2 RELATED WORK

We present our related work in three different but complementary topics to build an Edge Cloud: **Edge Clouds** is a broader but recent topic, **Resource Management** that consist in discovering the resources and schedule the computations or data into the system's nodes, and finally **usage Fairness** that cover topics for maintaining a volunteer multi-user system fair for all the users without abuses and violations.

### 2.1 Edge Clouds

Cloud@Home [8] work tries to provide a framework for the development of an Edge Cloud, where the Volunteer Computing and the Edge Computing intersect. The authors provide a high level description of the system, where a centralized set of nodes would control the resources of the volunteer nodes, discovering resources and scheduling the user's requests (via VMs deployment) in the nodes. They soon state that when the amount of devices increased above a given threshold, distributed and decentralized schedulers would be needed to cope with the amount of devices, which is exactly what we aim for in our work.

Autonomic Cloud [9] is a preliminary work in a P2P Cloud that also target volunteer resources to build it. The authors were testing the platform development on top of the Pastry DHT in order to find resources and communicate in a scalable fashion for large networks such as an Edge Cloud. They used OSGI bundles as a deployable component in the nodes, which does not provide so good isolation and security as a VM or even a Container for multi-tenant platforms.

### 2.2 Resource Management

Resource Bundles [5] work presents a resource discovery algorithm for node's current available #CPUs and RAM using an hierarchical overlay of nodes. Some node's are responsible for set of regular nodes, designated by super nodes. These super nodes use a cluster algorithm called multinomial model-based expectation maximization in its regular nodes resource availability. This clustering algorithm allows to aggregate the regular's nodes available resources in a compacted form while maintain a good degree of node's individual available resources. It allows to reduce the traffic in the network when spreading the node's resource availability. It has SPoF and bottleneck in the super nodes that are responsible for regular nodes, although it is more scalable than centralized solutions.

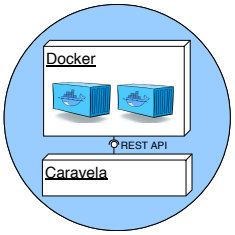



**Figure 1: Caravela's mandatory node's components.**

Selimi et al. [12] work contains resource discovery and scheduling algorithms that schedule services (sets of co-related containers, e.g. micro services) in nodes with higher bandwidth available, in the context of GUIFI.net community network. It only considers the node's available bandwidth for the discover and deployment. It uses a centralized solution where the knowledge of the network and all nodes is necessary. They use the K-means clustering algorithms to group nodes by its geo-graphical position, then they calculate the groups head nodes that maximize the bandwidth available with its group's nodes in order to discover the nodes with best links connecting them. Finally they recalculate the groups formed early but this time with the bandwidth information in order to group the nodes with higher bandwidth available between them.

Docker Swarm is a Docker Container orchestrator that use a centralized server/node (or set of replicated nodes) to control all the other nodes, taking the scheduling decisions in a centralized way. This approach is able to enforce global policies in the deployment like consolidating the containers to maximize resource utilization or spreading to offer better performance for the containers with a similar distribution of load in the nodes. The centralized server is a bottleneck to the system scalability and SPoF. It only consider the node's current availability of #CPUs and RAM.

### 2.3 Fairness

SocialCloud [6] is a centralized reputation system that looks for user's social network's friends, friends of friends and so on in order to provide a trustworthiness rank for the users. User's that have a higher trustworthy rank are preferable to realize trades, e.g. nodes from users with high ranks are preferable to deploy container(s) because they are more trustworthy. In the case of the SocialCloud the centralized solution is a bottleneck and the network of friends to rank the users is a bit restrictive in a system with tens of thousands of users as an Edge Cloud.

Karma [15] work consists in a distributed and decentralized reputation system that maintains user's reputation as users interact, e.g. in our case the user's node receive a container from other user to run, these interactions increase the users' reputation if it all happens smoothly or decreases it from the user that sabotage the interaction. Edge Clouds with volunteer resources need this kind of mechanisms to maintain the system usable without free-rides and other attacks that otherwise would make the system unusable.

## 3 ARCHITECTURE

Caravela is a Docker container's orchestrator that use the users' donated devices to provide computational, storage and network capabilities to build an Edge Cloud. So it is **mandatory** for Caravela's nodes to have (See Figure 1):

- Docker's Engine running;
- Caravela's middleware running as daemon;
- To simplify each node should have static public IP address.

The following components are optional (out of the scope for our work), because they are not mandatory to demonstrate Caravela's resource discovery and scheduling algorithms scalability, efficacy and efficiency, but they would be mandatory in a real life deployment:

- A client for a highly distributed and decentralized **file system** like IPFS [3] or BitTorrent [11]. It would be necessary to transfer and maintain the container's images in a scalable way. We used DockerHub, a centralized public repository of container's images to demonstrate the Caravela's prototype basic functionalities.
- A client for a distributed and decentralized **reputation system**, e.g Karma, to maintain user's reputation in order to control user's abuses in the system like promising a certain kind of resources but giving others.
- A client for a distributed and decentralized **virtual currency system**, e.g. Bitcoin, to maintain user's balance between its contributions with devices/resources and its consume of the other users resources.

Caravela provides a REST API for the users, allowing to interact with its node's daemons. We also developed an CLI tool to consume the REST API facilitating the use of Caravela. This tool provides the same syntax and a similar semantics to the Docker Swarm's CLI tool. The CLI allows the user to do the three fundamental operations specified as follows:

- **Deploy Container(s)**: Allow the user to deploy a container in Caravela specifying its resources needs CPUClass, #CPUs and RAM necessary. CPUClass is a binary value that we use to classify the node's performance (correlated with the CPU speed). It also allows to do a Stack Deployment (Swarm also allows it) which consist in deploying a set of correlated containers in the system in one request, e.g. micro services.
- **Stop Container(s)**: Stop containers releasing its resources from the node where it was deployed.
- **List Containers(s)**: List all the user's containers running and its respective details.

Note that in Docker Swarm there exist no notion of CPUClass or CPU capabilities because it targets homogeneous clusters of machines which is not the case of an Edge Cloud.

Now that we settled the prerequisites and operations of Caravela we will describe how we manage all the nodes/devices that are part of the Caravela in a distributed and fully decentralized way, helping us to build scalable and efficient resource discovery and scheduling algorithms.

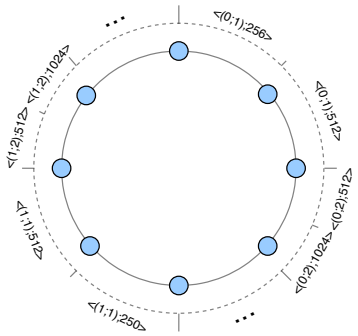



**Figure 2: Resources regions mapping in Chord's ring.**

### 3.1 Network Management

Caravela is built on top of a Chord [13] overlay, that consists in a ring of node each one with a unique ID in a space of *k-bits*. Chord provides one single operation: map a given key in a node, looking for the node in average $log_2(N)$ hops with N being the network's size. Caravela use Chord to leverage this lookup operation in order to find the resources necessary to deploy a container in a scalable and efficient way for large networks.

Chord's typical use consist in finding the node that contains some data (e.g. files or chunks of files) hashing the content's identifier/key with a consistent hashing algorithm (e.g. SHA-1) to obtain a Chord's *k-bits* key. With the key the Chord's client provide it to its lookup mechanism that will return the IP address of the node with the content. The consistent hashing provides a good dispersion of the keys over the nodes balancing the load in the system which is important in large networks.

When a user submit a container it specifies the resources that the container needs in a form of a pair $< (CPUClass; \#CPUs); RAM >$, if the system does not find any node with that minimum of resources available the user is notified of the error and can retry later. We use Chord to find out what are the nodes that have enough CPUs and RAM to run a user's container, e.g. if a user requests $< (0; 2CPUs); 512MB >$ we need to find at least a node with that amount of resources available in that moment. We look for the **node's current available resources**. So using the typical approach for Chord, hashing the resources needed with SHA-1, would result only in perfect matches, e.g. $< (0, 1CPU); 256MB >$ and $< (0, 1CPU); 300MB >$ would be mapped to completely different nodes while its needs are very similar. Basically the equal-based search of a typical Chord's use must be replaced by a kind of range query search.

To solve this problem we **encoded the resources availability of the nodes in its IDs**. We divided the Chord's ID/Key space (statically) in contiguous regions that represent different combinations of resources. Figure 2 pictures an example of the resources encoding in Chord's ring. One region with $< (0; 2CPUs); 512MB >$ label means that the nodes that have IDs in that region are responsible for node's with resources availability of at least the specified in the region's label. In Section 3.2 we detail how we leverage this mapping. Figure 2 pictures larger regions for weaker combinations of resources, this is by design, because in a real Edge Cloud we expect

that there are much more nodes offering weaker combinations. It is natural that are more users offering small resources than large resources.

## 3.2 Resource Discovery

Before introducing our resource discovery algorithm we introduce some terminology used in the rest of the paper. **Resources offer** (**offer** to simplify) consist in a data structure that contains:

- **IP address** of the node that have the specified resources available;
- **Node's resources available** < (*CPUClass*, *#CPUs*); *RAM* >;
- **Node's resources used** < (*CPUClass*, *#CPUs*); *RAM* > (helpful for global scheduling policies);
- **Offer's unique ID** (unique for its supplier only).

Each node has three roles depending on the action it is doing:

- **Supplier**: Node's role when it is supplying its resources (via offers);
- **Buyer**: Node's role when it is searching for resources (via offers) to deploy a container in behalf of the node's owner/user;
- **Trader**: Node's role when the it is mediating the supply/search for offers.

From here onwards if we describe actions made by a trader, supplier or trader is same as if it was node since each role as on-to-one mapping with the node. Each trader is responsible for offers that belong to the resource region where its ID belong. Suppliers publish offers into traders and buyers search for offers in traders.

**Algorithm 1:** Supplier's resource supplying.

```
   Data: suppOfferMap
1  Function SupplyResources(freeRes, usedRes):
2      regions ← SuitableResourcesRegions(freeRes)
3      foreach offer in suppOfferMap do
4          offerRegion ← Region(offer.TraderID)
5          if regions.Contains(offerRegion) then
6              upOffer ← Offer(offer.ID, freeRes, usedRes)
7              UpdateOffer(upOffer)
8              regions.Remove(offerRegion)
9          end
10         else
               /* When node free resources decrease.  */
               suppOfferMap.Remove(offer.ID)
11             RemoveOffer(offer.ID)
12         end
13     end
14     foreach region in regions do
           /* When node free resources increase.     */
           /* See Algorithm 2.                       */
15         CreateOffer(freeRess, usedRes, region)
16     end
```

The Algorithm 1 is called by the supplier every time its free resources change, e.g. due to container's launch/exit in the node consuming/releasing resources. A supplier provides its resources to the buyers by creating $N$ offers in the system, one for each of the configured resource regions that represent less or equal resources than the supplier's current free resources (line 2). After that the supplier iterates the offers it already has in the system and looks for the regions where they are registered (lines [3-4]). Than we cross reference the regions where we must create offers and the regions from the offers where we already have them (line 5). The rest of the algorithm is straightforward, if the supplier already have an offer in the region an `UpdateOffer` message is sent directly to the trader (the supplier saves the trader's IP address when creates an offer) in order to update the offer free/used resources (lines [6-8]). This update will be important in Section 3.3 when we try to enforce global policies in the container's scheduling. If the supplier have an offer in a region where its current free resources cannot handle the requests for it, the supplier removes the offer from the trader sending a `RemoveOffer` message (lines [11-13]). Finally if there exist regions where the supplier does not have any offers, it creates one offer in each region (lines [14-15]).

**Algorithm 2:** Supplier's create offer algorithm.

```
   Data: supplierIP
1  Function CreateOffer(freeRes, usedRes, destRegionRes):
2      newOffer ← Offer(freeRes, usedRes, supplierIP)
3      destTraderID ← RandomID(destRegionRes)
4      traderIP ← ChordLookup(destTraderID)
5      ok ← CreateOffer(traderIP, NodeInfo(), newOffer)
6      if ok = true then
7          newOffer.SetTraderIP(traderIP)
8          suppOffersMap[newOffer.ID] = newOffer
9          return
10     end
11     return Error("offerCouldNotBeCreatedError")
```

How a supplier creates/publishes a new offer? The supplier runs the Algorithm 2. It starts by creating an offer object with the supplier's resource availability (line 2). After that it obtains a **random** ID/key in the offer's target resource region (line 3), the random is used to distribute the load among the region's traders. With the ID it calls Chord to obtain the trader's IP responsible for that ID/Key (line 4). Finally it sends a `CreateOffer` message to the trader (line 5). The trader register the offer in its internal offer's table and acknowledges it. The supplier register the trader's IP responsible for the offer and also register the offer in its internal offer's table (lines [7-9]).

Now that we have the supplier providing the resources into the system as the node's available resources change, how can we discover offers for a container deployment request? The Algorithm 3 details how it happens, it receives the container's necessary resources. The algorithm have a maximum retry threshold defined in the system configuration file (parameter $MaxDiscovery_{retries}$) that is used to limit the times we try to search for the resources (lines [2-3]). In a retry we start by generating a random ID/key (again to distribute the load between region's traders) in the region responsible for the resources closest to the requested (line 4). Then we use Chord to get the trader's IP and, we send a `GetOffers` message to it in order to obtain the trader's registered offers. If the set

**Algorithm 3:** Resource discover algorithm.

```
   Data: configs
1  Function DiscoverResources(resourcesNeeded):
2      retry ← 0
3      while retry < configs.MaxDiscoverRetries() do
4          destTraderID ← RandomID(resourcesNeeded)
5          traderIP ← ChordLookup(destTraderID)
6          resultOffers ← GetOffers(traderIP)
7          if resultOffers != ∅ then
8              return resultOffers
9          end
10         retry ← retry + 1
11     end
12     return ∅
```

of offers received is not empty we return them otherwise we retry if the threshold was not reached.

Until now our algorithms rely on Chord's lookup only when we need to publish offers due to node's resource availability increase and when searching for offers. Chord's lookup is the most expensive network operation here with $log_2(N)$ ($N$ being the network size) so we avoid it at maximum. We will see in evaluation that we configured the maximum retries of the discover resources algorithm to only 1 obtaining a very interesting efficacy and efficiency in the discovery process.

Due to the node's crash (likely scenario in an Edge Cloud) a trader could be giving offers from dead suppliers, and consequently a supplier would think that its resources were available in the trader but the trader was dead. To minimize this problem each trader refresh an offer from time to time (sending a `RefreshOffer` message) defined by Caravela's parameter $Refresh_{interval}$. This way the trader acknowledges the presence of the supplier and vice versa. The parameter $MaxRefreshes_{failed}$ define how many refreshes a supplier can fail before the trader removes the supplier's offer, and complementary the parameter $MaxRefreshes_{missed}$ define how many refresh a trader can fail before the supplier publish the offer onto other trader (of the same region).

Why a supplier publishes its resource availability in several regions instead of publishing it only in the highest one? Because that way we would have little information per trader (one offer per node only) which would decrease our efficiency and efficacy when looking for resources due to the little information spread over too many nodes.

## 3.3 Container's Scheduling

Now that we have explained the resource discovery process, in this section we describe how we implemented the container's scheduling on top of it. But before detailing the scheduling algorithm we want to introduce the notions of **global scheduling policy** and **request-level scheduling policy**.

Docker Swarm offers **two** global scheduling policies: **binpack** and **spread**. When binpack is configured the system's scheduler tries to consolidate containers in few nodes, while providing the container's requested resources. Spread is the opposite of binpack, its distributes the containers thinly by all the system's nodes. These policies are applied globally to all the requests scheduled in the system and all the nodes respect the configured policy because every node that joins Caravela receives a copy of the system configurations. Caravela's configurations are used to configure the bootstrap nodes. All the other nodes that join it receive a copy of it.

Docker Swarm allows a stack deployment which consist in a composite deployment request where a user can specify a set of container to be scheduled together. This is a common case nowadays with micro services deployments. We also allow these stack deployments in Caravela. We extend the stack deployments to allow **request-level (or group level) scheduling policies**, which means a user can specify different scheduling policies for containers in the stack deployment. We developed the **co-location** and **spread** request-level scheduling policies. The co-location scheduling policy allows the user to specify that a set of nodes in the stack deployment must be scheduled in the same node. The spread policy might be used by the user specify container that must be deployed in different nodes. Co-location is useful for components/containers that communicate a lot or need low latency communications. Spread is useful for robustness properties. The global scheduling policies and the request-level ones are orthogonal, so the Caravela can be consolidating (with binpack) while the user's systematically request all the containers to be spread (request-level spread) and vice versa.

**Algorithm 4:** Algorithm to schedule containers in nodes.

```
   Data: globalSchedulingPolicy
1  Function Schedule(contConfigs, resourcesNeeded):
2      offers ← DiscoverResources(resourcesNeeded)
3      offers ← globalSchedulingPolicy.Rank(offers)
4      foreach offer in offers do
5          contsStatus ←
             Launch(offer.SuppIP, offer.ID, contConfigs)
6          if contsStatus != ∅ then
7              return contsStatus
8          end
9      end
10     return Error("CouldNotScheduleContainersError")
```

Before explaining the complete algorithm that runs when a user's deployment request is submitted, we describe the algorithm (Alg. 4) that given a set of container's configurations (one per container), and the sum of all the resources necessary by all the containers, it founds the suitable node for deploying them. The first thing the algorithm does is to call the Algorithm 3 with the resources needed. It receives a set of offers that can be used to deploy the container. If the set of offers is empty we return an error to the user (line 10). Otherwise we will rank the set of offers accordingly to the global policy configured in Caravela (binpack or spread). The algorithms used to rank[4] are a slightly adaptation of the used in Docker Swarm. The adaptation introduces the *CPUClass* attribute in the ranking. We do not present them here due to the space constraints. Lastly, the algorithm iterates the ordered offers and sends a `Launch` message

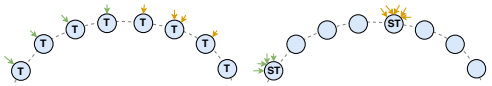
[4]Node ranking algorithms available at:
https://github.com/docker/swarm/tree/master/scheduler/strategy

to the supplier responsible for the offer. When the first supplier that acknowledges the launch of the containers the algorithm exits returning the information about the containers (container status) deployed and the IP of supplier that will run them.

**Algorithm 5:** Buyer's on request deployment algorithm.

```
1  Function OnDeploymentRequest(containersConfigs):
2      deploymentFailed ← false
3      colocatedResSum ← NewResources(0, 0)
4      colocatedConts, spreadConts, deployedConts ← ∅
5      foreach contConfig in containersConfigs do
6          if contConfig.GroupPolicy = "Co-location" then
7              colocatedResSum.Add(contConfig.Resources)
8              colocatedConts ← colocatedConts ∪ contConfig
9          end
10         else
11             spreadConts ← spreadConts ∪ contConfig
12         end
13     end
       /* Skipped if there are not co-located.        */
14     contStatus ←
        Schedule(colocatedConts, colocatedResSum)
15     if contStatus = nil then
16         return NewError("DeployFailedError")
17     end
18     deployedConts ← deployedConts ∪ contStatus
       /* Skipped if there are not spread.            */
19     foreach contConfig in spreadConts do
20         spreadContRes ← contConfig.Resources
21         contStatus ← Schedule(contConfig, spreadContRes)
22         if contStatus = nil then
23             deploymentFailed ← true
24             break
25         end
26         deployedConts ← deployedConts ∪ contStatus
27     end
       /* Rollback the deployment if necessary.       */
28     if deploymentFailed = true then
29         foreach cont in deployedConts do
30             StopContainer(cont.SuppIP, cont.ContID)
31         end
32         return Error("DeployFailedError")
33     return deployedConts
```

With all the scheduling properties specified and how a set of container are scheduled in a node we will now describe how it is performed the complete request deployment algorithm incorporating the request-level policies too. Algorithm 5 describes the function called by the node's buyer when the node's own/user inject a container(s) deployment request on it. The `OnDeploymentRequest` function receives as parameter a non empty set of container's configurations (one per container) specify all the details for the container: image's key, container name, container's arguments, port




(a) Without super traders. (b) With super traders.

**Figure 3: Super Traders usage.**

mappings (Host:Container), resources necessary and group policy. The algorithm starts by iterating all the configurations (line 7), if the container has the co-location group policy active it sums the container's resources in an accumulator, and adds the configuration structure in a list of the co-located containers (lines [8-10]). If the container has the spread group policy configured, it adds it to other list for the spread containers. The next part of the algorithm consists in scheduling the co-located containers in one node. To do this, it calls the `Schedule` function (recall Alg. 4) with the container's configurations and with the sum of its resources needs (lines [16-20]). As we already showed when it returns with success means that the containers are already being deployed in a node. After the co-located containers are deployed, it needs to deploy the spread containers. This is straightforward, by calling the `Schedule` function one time per each spread container (lines [21-29]). Due to the `Schedule` (Alg. 4) and `DiscoverResources` (Alg. 3) functions that select random traders to obtain offers, with high probability we obtain different nodes to deploy the spread containers. If any of the container deployment fail we rollback all the others that were already deployed by sending `StopMessages` to respective suppliers and returning an error to the user (lines [30-35]).

Note that if a user does not specify the container group-policy we assume it is spread. If it does not specify the resources necessary to the container we assume the value designated in lowest resource region. In the co-located containers we assume the CPUClass of the target node to be 1 (highest available) if exist at least one co-located container with it specified.

## 3.4 Optimization: Super Traders

Our discovery algorithm previously described (recall Alg. 3) selects a random trader in the resource region it targets. When the system is low on resources there are less offers in the traders so the chances of targeting an empty trader is higher. This lead to two problems: if the trader is empty our search for resources would fail and it was needed a retry (automatically or done by the user), which would affect the the resource discovery efficacy and efficiency; the second problem is that our global scheduling policy is enforced accurately if the buyer has many offers to rank and choose.

To mitigate these problems we devised a new way to chose a trader of a resource region. Instead of choosing a random key from the resource region, we choose a random key from a limited set of keys evenly distributed in the region. The size of this set of keys affect how we concentrate the `CreateOffer` and `GetOffers` messages in more or less traders. In the end we are creating a kind of super traders that would manage more offers than before, while the other nodes would not manage any offer. Figure 3 pictures the nodes receiving `CreateOffer` and `GetOffers` (colored arrows) without and with super traders. The amount of super traders can

be controlled by configuration parameter $SuperTrader_{factor}$, e.g. the value 7 for the parameter would mean that each super trader would manage the same offers as 7 nodes.

# 4 EVALUATION

To evaluate our Caravela's prototype we developed a cycle-based simulator called CaravelaSim, due to the necessity of re-utilizing ours complete Go's code base of Caravela, and at same time test the scalability of the prototype with thousands of nodes. Our simulations ran with **20s ticks** and a duration of **360 ticks (2h of simulation)**. Our simulator tested the real components of Caravela. We implemented the Chord's protocol as it is in its paper, except the background stabilization protocol, due to the simulation overhead.

We developed two resource discovery and scheduling benchmarks to compare with ours. A Docker Swarm centralized solution adapted to work over Chord and a naive random approach also over Chord, from now on designated by **Swarm** and **Random** respectively. The **Swarm** uses a master node that receives the offers and the deployment requests from all the nodes. This master node takes into account all the offers/nodes when deploying containers so it is a near "oracle" approach that allow us to obtain a near perfect request satisfaction and the near perfect global policy enforcement. The master node is the Chord's node responsible for the key 0. The **Random** approach is very simple, when a node receives a deployment request it looks for a random key/node in Chord. If the node has enough resources to accept the request, it sends the Launch message immediately with the container(s)'s information, otherwise the request is retried automatically by the system until the maximum retries are achieved. When the maximum retries are achieved it is considered a failed request and the user is informed. This approach has minimal overhead and is used to verify that our problem did not have a trivial solution. Our approach is designated as **Multi-Offer** from here onwards.

We submitted the same request stream (deploy containers requests and stop containers requests) to all the approaches in order to maintain the evaluation fair. We submitted the system to a load where at least 50% of its total resources would be used. After that the request stream had more or less the same deploys and stops in order to maintain the system in a near constant state of resource utilization. We did this because because a real and well designed system is always at least 50% utilized, otherwise it would be overdimensioned or poorly used system.

The requests profiles (in terms of resources needs) contained 50% of light requests (e.g. micro services deployments) the other 50% were heavier requests (e.g. heavy applications or heavy background tasks).

In the remainder of this section we present the results of our evaluation. We gathered 4 main metrics to verify the scalability (without nodes being bottlenecks) of the solutions and its discovery and scheduling algorithms efficacy and efficiency. The metrics are: **bandwidth consumed per node**, **RAM used per node**, amount of **requests fulfilled with success** (user satisfaction and resource discovery algorithm efficacy), and the **efficiency of the deployment requests** (assess discovery algorithm efficiency). We tested the approaches with two network sizes **65,536** (a.k.a 64K) and **1,048,576** (a.k.a $2^{20}$) in order to test the approaches scalability when the network grows **16 times**. Note that we do not show higher network sizes or the Swarm approach for the $2^{20}$ network because the simulations would take too many hours. Swarm simulation time hints that its not scalable as we will show next.

## 4.1 Bandwidth consumed per node

Figures 4 and 5 show the distribution of the bandwidth consumed per node (on receiving) over time. Note that the amount of outliers represented in the quartile plots (these and the ones that follows) are [7%-9.5%] of system's total node (super traders). Figures depicts that Swarm master node, the highest outlier (in 64K network), consume **500 times** more bandwidth than the highest outlier in Multi-Offer. With smaller networks we noted that the bandwidth consumed by the master node double when the network size also doubled. So, doing the extrapolation we checked that with $2^{20}$ network, the master node would consume ≈**2.8TB of bandwidth** in a month span, which would be unbearable for a user due to ISPs fair usage. Multi-Offer consumes a little bit more than Random, but would only consume ≈**150MB in a month span**. When the network scales **16 times** the node's bandwidth consumption only increased by a factor of ≈1.2 per node (on average) which gives a great scalability factor to Multi-Offer.

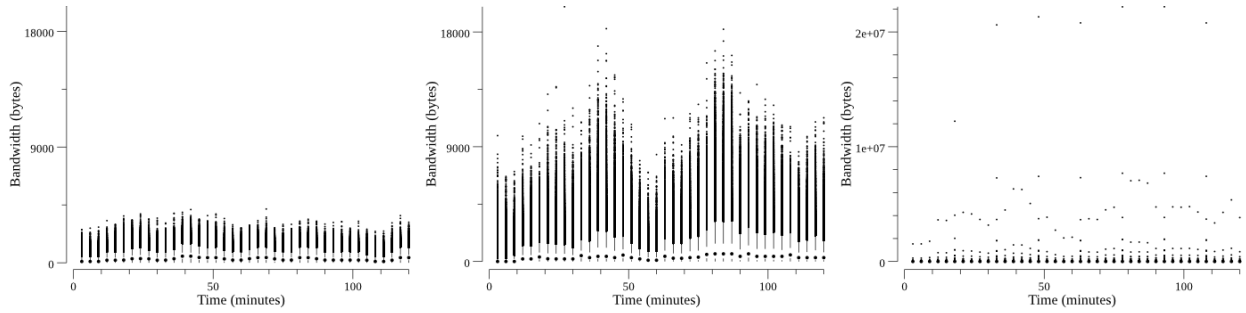
## 4.2 RAM used per node

Figures 6 and 7 pictures the distribution of the RAM used per node over time. We only account for the Caravela's data structures (without Chord and WebServer structures) in order to check the overhead of our solution. We can extract similar conclusions as the ones presented in the bandwidth consumption per node. Swarm's master node needs to save information about all the node's participating in order to schedule the container, it also needs to save the information about each request scheduled in the system.

## 4.3 Deployment Request Efficacy

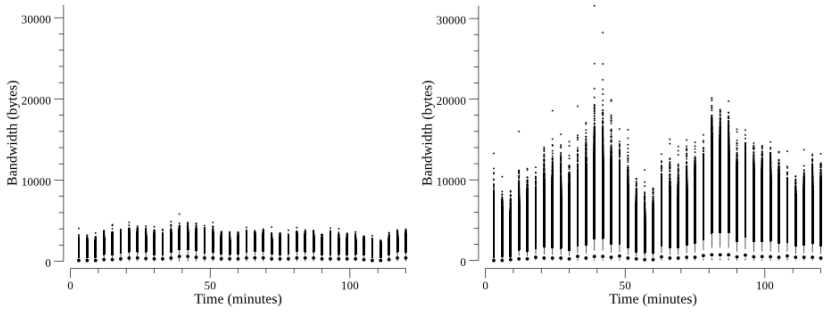
Figure 8 pictures the user's deployment request fulfilled (cumulatively) over time. Swarm can fulfill 4% more requests than Multi-Offer. Multi-Offer can maintain a good deployment efficacy with a fully decentralized architecture. Multi-Offer in the end of the simulation deployed ≈**24%** more requests than Random. It is worth to mention that the simulations ran with an automatic retry mechanism for Random and Multi-Offer. The maximum retries for Random was set to 3 and the Multi-Offer for 1. As we will show later the 3 retries of Random gives it a very inefficient resource discovery in terms of network hops.

We also started to notice that random failed to deploy the requests with higher resource needs, e.g. $< (0; 3CPUs); 2GB >$ so we devised a deployment request allocation efficacy metric pictured in the Figure 9. The metric consists in the cumulative ratio of $totalResourcesAllocated/totalResourcesRequested$ over time. It shows that Multi-Offer is near swarm which is good because due to the centralized solution of Swarm it manifests a near-optimal allocation efficacy because it knows all the nodes' state. As expected Random allocates much less resources than Multi-Offer. Random is only useful for light requests.



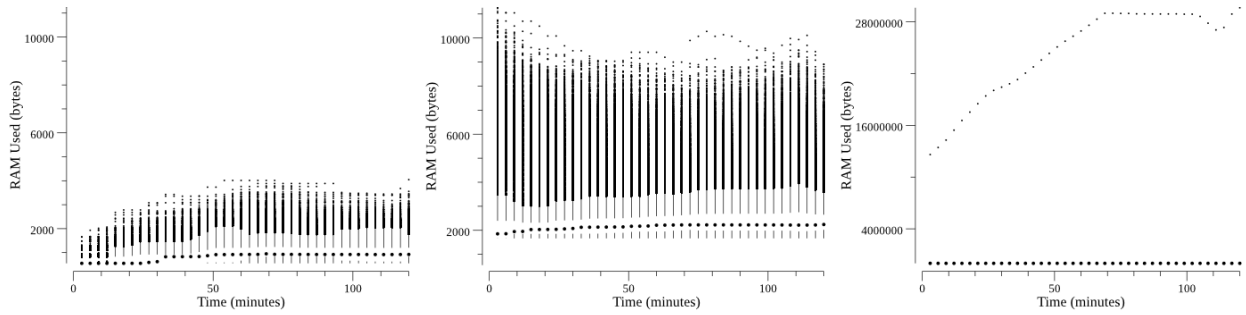

(a) Random (b) Multi-Offer (c) Swarm

**Figure 4: Bandwidth used per node over time, in the 65K node's network (Quartile Plots).**




(a) Random (b) Multi-Offer

**Figure 5: Bandwidth used per node over time, in the 1M node's network (Quartile Plots).**



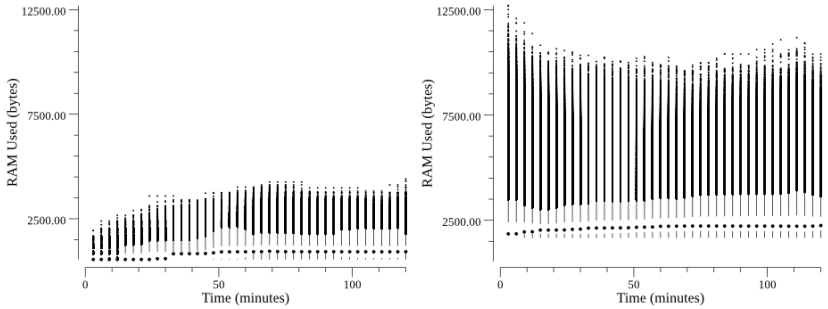

(a) Random (b) Multi-Offer (c) Swarm

**Figure 6: RAM used per node, in the 65k node's network (Quartile Plots).**

## 4.4 Deployment Requests Efficiency

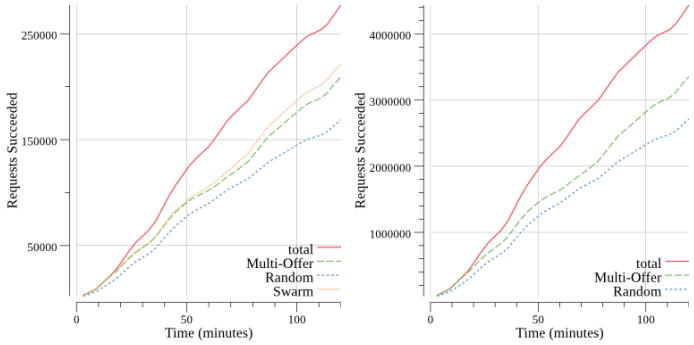
After we assess the deployment request efficacy, here we assess how much it costs for each approach. Figures 10 and 11 picture the distribution of sequential messages (hops), taken until the deployment request succeeded or failed. Swarm as a constant cost of 3 messages because the master nodes saves the node's IP making the subsequent contacts direct. **Multi-Offer highest outlier costs less than Random's median cost**. It is also notable that Multi-Offer cost has a small variance compared with Random. When there are few free resources in the system, Random uses its retries to achieve the deployment request efficacy that we showed before, making its efficiency worst. This metric is important in Edge Clouds due to the WAN networks that connects the nodes. More hops/messages means higher latency between the user's request submission and the reply from the system telling the success or failure of the request.

Due to space constraints we did not provide the plots with the simulation of the Random approach with only 1 retry. With 1 retry the deployment request efficiency became the same as Multi-Offer



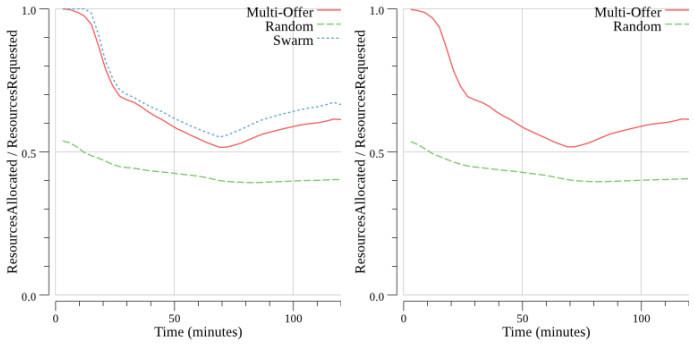

(a) Random (b) Multi-Offer

**Figure 7: RAM used per node, in the 1M node's network (Quartile Plots).**




(a) 65k nodes network (b) 1M nodes network

**Figure 8: Deployment requests successfully attended.**



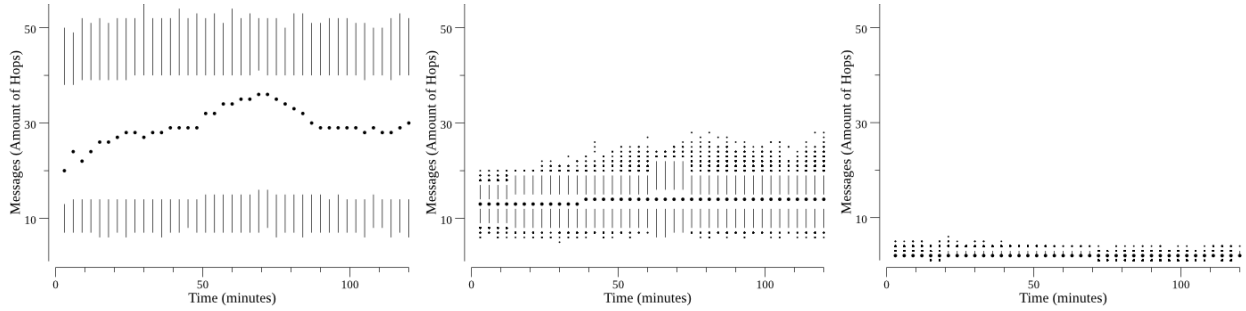

(a) 65k nodes network (b) 1M nodes network

**Figure 9: Deployment requests resource allocation efficacy (Cumulative).**

because it only uses one Chord lookup too. The problem was that the deployment request efficacy we showed in Section 4.3 decreased. Multi-Offer fulfilled ≈**70%** more requests than Random.

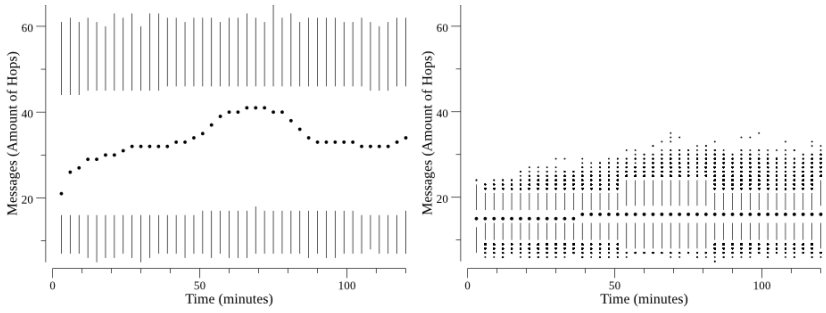
## 5 CONCLUSION

Caravela was devised to serve as a fully decentralized Docker Cloud to be deployed in an Edge Computing environment, where there are tens of thousands of nodes participating, high latencies between the nodes and no central administration. Its architecture and algorithms verified to be close in terms of efficiency and efficacy to a centralized "oracle" solution as our adaptation of Docker Swarm to Edge Cloud environment, while maintaining its scalability even with $2^{20}$ nodes. A typical centralized solution is defeated by the scale. The Random approach was scalable with a low overhead per node, but it had




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

**Figure 10: Distribution of the number of messages exchanged per deploy request submitted, in the 65k node's network.**




(a) Random (b) Multi-Offer

**Figure 11: Distribution of the number of messages exchanged per deploy request submitted, in the 1M node's network.**

a low deployment request efficiency and efficacy. It also cannot enforce the binpack global scheduling policy that is interesting to leverage the maximum of the system/nodes.

## REFERENCES


[1] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. 2002. SETI@home: an experiment in public-resource computing. *Commun. ACM* 45, 11 (2002), 56–61. https://doi.org/10.1145/581571.581573

[2] Roger Baig, Ramon Roca, Felix Freitag, and Leandro Navarro. 2015. Guifi.net, a crowdsourced network infrastructure held in common. *Computer Networks* (2015). https://doi.org/10.1016/j.comnet.2015.07.009

[3] Juan Benet. 2014. IPFS-Content Addressed, Versioned, P2P File System. *IPFS-Content Addressed, Versioned, P2P File System* Draft 3 (2014). https://doi.org/10.1109/ICPADS.2007.4447808

[4] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. 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* (2012), 13. https://doi.org/10.1145/2342509.2342513

[5] Michael Cardosa and Abhishek Chandra. 2010. Resource bundles: Using aggregation for statistical large-scale resource discovery and management. *IEEE Transactions on Parallel and Distributed Systems* 21, 8 (2010), 1089–1102. https://doi.org/10.1109/TPDS.2009.143

[6] Kyle Chard, Simon Caton, Omer Rana, and Kris Bubendorfer. 2010. Social Cloud: Cloud computing in social networks. In *Proceedings - 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD 2010*. 99–106. https://doi.org/10.1109/CLOUD.2010.28

[7] Cisco Systems. 2016. Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are. *White Paper* (2016), 6. https://doi.org/10.1109/HotWeb.2015.22

[8] Vincenzo D. Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. 2009. Cloud@Home: Bridging the gap between volunteer and cloud computing. In *Lecture Notes in Computer Science*, Vol. 5754 LNCS. 423–432. https://doi.org/10.1007/978-3-642-04070-2{_}48

[9] Philip Mayer, Annabelle Klarl, Rolf Hennicker, Mariachiara Puviani, Francesco Tiezzi, Rosario Pugliese, Jaroslav Keznikl, and Tomáš Bure. 2014. 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*. 89–94. https://doi.org/10.1109/SASOW.2013.16

[10] Peter Mell and Timothy Grance. 2011. The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology. *Nist Special Publication* 145 (2011), 7. https://doi.org/10.1136/emj.2010.096966

[11] Johan Pouwelse, Pawel Garbacki, Dick Epema, and Henk Sips. 2005. 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. 205–216. https://doi.org/10.1007/11558989{_}19

[12] Mennan Selimi, Llorenc Cerda-Alabern, Marc Sanchez-Artigas, Felix Freitag, and Luis Veiga. 2017. Practical Service Placement Approach for Microservices Architecture. In *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017*. 401–410. https://doi.org/10.1109/CCGRID.2017.28

[13] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking* 11, 1 (2003), 17–32. https://doi.org/10.1109/TNET.2002.808407

[14] Luis M. Vaquero and Luis Rodero-Merino. 2014. Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *ACM SIGCOMM Computer Communication Review* 44, 5 (2014), 27–32. https://doi.org/10.1145/2677046.2677052

[15] Vivek Vishnumurthy, Sangeeth Chandrakumar, and G Emin. 2003. Karma: A secure economic framework for peer-to-peer resource sharing. *In Workshop on Economics of Peer-to-peer Systems* 34 (2003).