Journal of Grid Computing (2021) 19: 36
https://doi.org/10.1007/s10723-021-09575-x

Distributed and Decentralized Orchestration of Containers on Edge Clouds

André Pires · José Simão · Luís Veiga

Received: 4 November 2020 / Accepted: 21 June 2021 / Published online: 23 July 2021

Abstract Cloud Computing has been successful in providing substantial 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 underutilized resources. Due to this, new computing paradigms like Edge Computing and Fog Computing are emerging. This work presents Caravela 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 solutions are mostly tied to a centralized cluster environment deployment. Caravela employs a completely decentralized architecture, resource discovery and scheduling algorithms to cope with (1) the large amount of volunteer devices, volatile environment, (2) wide area networks that connects the devices and (3) nonexistent natural central administration.

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 to function like a single viable animal.

A. Pires
Instituto Superior Técnico, ULisboa, Lisbon, Portugal
e-mail: pardal.pires@tecnico.ulisboa.pt

J. Simão (✉)
INESC-ID Lisboa, Instituto Superior de Engenharia de Lisboa, IPL, Leiria, Portugal
e-mail: jose.simao@isel.pt

L. Veiga
INESC-ID Lisboa, Instituto Superior Técnico, ULisboa, Lisbon, Portugal
e-mail: luis.veiga@inesc-id.pt

Keywords Edge cloud · Cloud computing · Volunteer computing · 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 [20]. 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 [7, 12] the network 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 [5] with ≈35K nodes and a steady growth of 2k nodes/year),and with the nowadays very powerful desktops and laptops, the network's edge is filled with a lot of resources that most of the time are underutilized. The Edge and Fog Computing intend to leverage these resources to provide services that are near the internet's edge.

CISCO's definition [8] for Fog/Edge Computing represents the space between the end user's devices and the cloud as a platform (potentially developed and backed up by their solutions) to offer compute/storage/network capabilities mainly for mediating the communication with the cloud. Vaquero et al. definition [32], like Varghese and Buyya [33] see Fog Computing as a completely decentralized platform where the user's own devices cooperate to offer the compute, storage and network capabilities using sandboxed environments. This approach would inherit techniques used is Volunteer Computing works like SETI@Home [1] and Cloud@Home [13].

Edge Computing has the potential to reduce latency and bandwidth costs, improve security and privacy [29], with application in various fields, including consumer applications, industrial applications, e-health services, and smart mobility applications [14]. Community Clouds are another example of edge infrastructure gaining attention in the last decade [27]. In these networks, the infrastructure is managed as a common resource and established by the participants. Some real case experiments have been used in these environments, such as file sharing, game servers, video streaming and surveillance, remaining as a challenge the definition of service deployment models for the users of these networks.

Our contribution is Caravela, a Docker container's orchestrator, inspired by Docker Swarm[1], but enhanced to be used as an Edge Cloud platform. A Docker container is the sandbox environment that a user can configure and deploy in Caravela. Our work targets the environment of fog/edge computing where the number of devices available is large, heterogeneous, and connected via wide area networks. An Edge Cloud must also be churn resilient because edge devices have a high failure rates. Furthermore, users can add or remove their devices, in the cloud, whenever they want. All of this require a distributed and decentralized architecture, discovery, and scheduling algorithms to cope with the number of nodes and users. Due to the volunteer environment, a decentralized solution is recommended since there is no natural central administration.

Caravela has the following main properties: a) Decentralization: We propose a distributed and decentralized architecture, resource discovery and scheduler algorithms to avoid Single Point of Failure and bottlenecks to cope with the substantial number of volunteer resources and wide area of deployment; b) Workload placement heuristics: Users should be able to specify the class of CPU (reflects CPU speed), number of CPUs and RAM they need to deploy a container. It should be possible to deploy a set of containers that form an application stack, specifying if the user wants the containers in the same node, promoting co-location, or spread them over different nodes, e.g., promoting resilience.

The rest of the paper is structured as follows. Section 2 briefly describes the fundamental and state of the art works in Edge Clouds, resource management and usage fairness, highlighting the innovative aspects of the proposed approach. Section 3 presents the architecture, the resource discovery and scheduling algorithms that compose Caravela. Section 6 presents the evaluation of Caravela, comparing its performance with an adaptation of the Docker Swarm and a naive random-based approach. Finally, Section 7 wraps up the paper with our main conclusions.

2 Related Work

The related work is presented in two different but complementary topics to build Caravela: Edge Clouds, which is a broader and recent topic, and Resource Management, that consist in discovering the resources and schedule the computations or data into the system's nodes. In Caravela we focused the study and development of the latter one applied to the Edge Cloud scenarios.

Edge Clouds Volunteer Edge Clouds is where Cloud Computing and Volunteer Computing intersect. Resources are provided by the user personal devices (e.g. Cloud@Home [13], Satyanarayanan et al. [26], Cloudlets [34], Babaoglu et al. [4] and Mayer et al. [19]). The users have incentives to join, e.g., with the possibility to deploy their own applications. Volunteer

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

Edge Clouds have the potential to join many widespread resources resulting in virtually unlimited computational and storage power. GridCop [36] is a system designed to track the progress and correctness of a program executing on remote and potentially fraudulent host machines. In hybrid Edge Clouds a large slice of the infrastructure belongs to a single entity (usually a Fog Computing entity like Internet Service Providers (ISPs)). The volunteer resources are hooked to the management layer providing the computational and storage power to the cloud (e.g., Nebula [25], Chang et al. [11] and Mohan et al. [21]).

Cloud@Home [13] 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 [19] is a preliminary work in a P2P Cloud that also targets 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 [23].

Resource Management The management of distributed resources consists in two main stages: Resource Discovery and Resource Scheduling [9]. Resource discovery (a.k.a Resource Provisioning) focus in discovering 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. Resource scheduling is composed of three processes: Resource Mapping, Resource Allocation and Resource Monitoring [30].

Docker Swarm is a Docker Container orchestrator that uses 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 can 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 considers the node's current availability regarding the number of CPUs and RAM.

Resource Bundles [9] work presents a resource discovery algorithm for node's current available number of CPUs and RAM using an hierarchical overlay of nodes. Some nodes are responsible for a 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 still has SPoF and bottleneck in the super nodes that are responsible for regular nodes, although it is more scalable than centralized solutions.

Kargar et al. [16] describe how multiple rings are set up in a peer-to-peer network representing each individual type of resources, while caring to avoid wide area message exchanges across the network. This can be leveraged to the same purposes of our work but presently there is no integration with container technology and orchestration/scheduling.

Selimi et al. [28] propose 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 discovery and deployment. It uses a centralized solution where the knowledge of the network and all nodes is necessary. A K-means clustering algorithm is used to group nodes by their geo-graphical position. Head nodes for each group are then determined to maximize the bandwidth available within the group to discover the nodes with best links connecting them. Finally, groups are recalculated taking into account the bandwidth information to group the nodes with higher bandwidth available between them. Recent work [27] made the system more dynamic (able to incorporate continuously updated network state information) and more responsive, by reacting with a faster heuristic-driven approach to changes in resource availability across the network.

Resource scheduling in edge computing can also be designed to introduce preferences among users when they are related via some social network [2], or by the history of past interactions and social structure created among users [3]. Long-term sustainability of edge clouds can be promoted with economic incentives and compensations reliably recorded using permissioned blockchain technology, such as Hyperledger Fabric (HLF) or Ethereum [15, 18].

Caravela makes it feasible to construct large-scale decentralized clouds, harnessing synergies from network-edge small and medium clusters and peer machines owned by end-users. Users should be able to specify workloads and computing power in a platform-independent way (resembling the serverless computing paradigm [10]) but controlling some heuristics regarding workload placement. Among the different challenges of building Edge Cloud [17], Caravela has to address: a) Scalability: the architecture design should be very scalable to accommodate large number of devices that can participate to provide increasing power; b) Workload isolation: users' applications will run in other users' machines so it is necessary to isolate the cloud platform from the underlying private user resources; c) 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; d) Ease of Use: make it simple to contribute with resources and deploy applications because the success of its volunteer part depends on the user interest.

3 Architecture and Resource Discovery

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 (Fig. 1):

- Docker's engine running;
- Caravela's middleware running as daemon.

To simplify Caravela's development (and deployment) each node should have a stable and public IP address (in realistic deployments, e.g., IPFS, web technology allows to lower these requirement).[2] Caravela middleware is a broker of resources among nodes, here each node can supply, offer or trade resources. Because of this market-oriented approach, the following components are also relevant for the overall system, although they are out of scope for this paper:

- A client for a highly distributed and decentralized file system like IPFS [6] or BitTorrent [24]. It would be necessary to transfer and maintain the container's images in a scalable way. We used Docker Hub, 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 [35], 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 [22], to maintain user's balance between its contributions with resources and its consumption of the other users resources.

Caravela's middleware exposes a Node-to-Node REST interface, that is used by the discovery and scheduling algorithms we will show, in order to cooperate. The middleware also provides a REST API for the users, allowing to interact with its node's daemons, and a command line tool to consume the API facilitating the use of Caravela. This tool provides the same syntax and a similar semantics to the Docker Swarm's CLI tool. Our API 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: CPU class, number of CPUs and RAM. The CPU class identify the node's performance (correlated with the CPU speed). It also allows to do a Stack Deployment (Swarm also allows it) which consists in deploying a set of correlated

[2] In edge computing deployments, and peer-to-peer before, there may be issues with bidirectional IP addressing due to network address translation (NAT). Web-based technologies such as WebRTC have specifications employing techniques circumventing these issues (port forwarding, ICE, STUN, TURN).

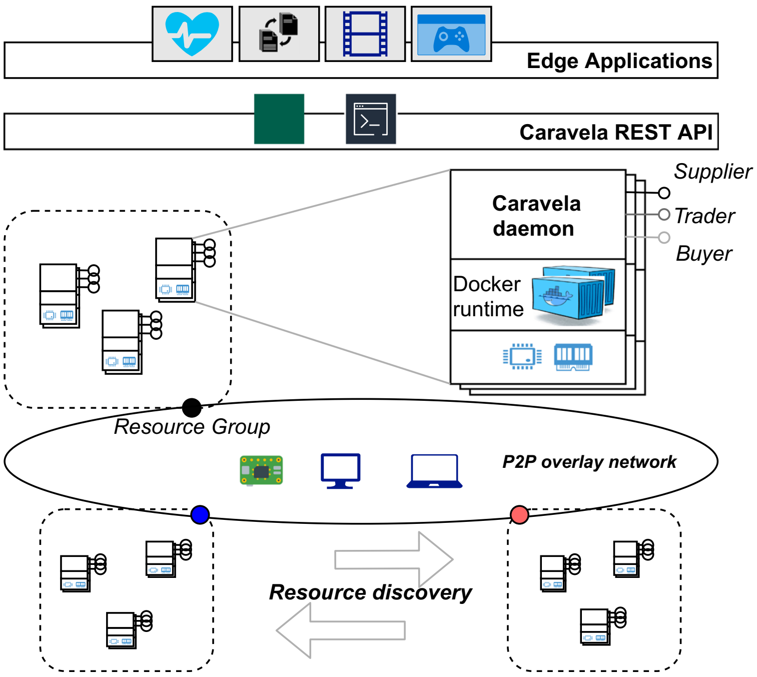


Fig. 1 Overview of Caravela's architecture

containers in the system in one request, e.g. micro services deployment;

– **Stop Container(s)**: Stop container(s) releasing the resources allocated for each one from the nodes where each 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 classes of CPUs because it targets homogeneous clusters of machines which is not the case of an Edge Cloud.

The following sections 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.

3.1 Network Management

Caravela is built on top of a Chord [31] P2P overlay, that consists of a ring of nodes. Each one has a unique ID in a key space of *k-bits*. Chord maps each key in a node, it can look up for for the key's responsible node in average $log_2(N)$ network hops with N being the network's size. Caravela uses 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 ID/key with a consistent hashing algorithm (e.g. SHA-1) to obtain a Chord's *k-bits* key. With the key Chord's client provide it to its lookup mechanism that will return the IP address of the node responsible for 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. In Caravela, the node's resources necessary for the container(s) (specified by the users) are the key that we provide to Chord. The next two paragraphs detail how we used Chord in Caravela.

When a user submits a container it specifies the resources that the container needs in 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. 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 nodes with resources availability of at least the specified in the region's label. In Section 3.2 we detail how we leverage this mapping to efficiently discover the resources. 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 of resources. 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) is a mechanism used by nodes to inform about their available

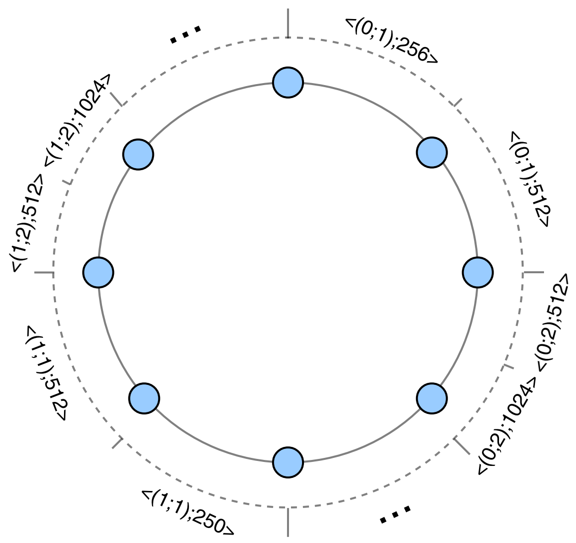


Fig. 2 Resources regions mapping in Chord's ring

and used resources, consisting in a data structure that contains the following node's information:

- **IP address** of the node that have the specified resources available;
- **Resources available** $\{(CPUClass; \#CPUs); RAM\}$;
- **Resources used** $\{(CPUClass; \#CPUs); RAM\}$;
- **Offer's unique ID**.

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 it is mediating the supply/search for offers.

Suppliers publish offers into traders and buyers search for offers in traders. Each trader is responsible for offers that belong to the resource region where its ID belong. From here onwards if we describe actions made by a trader, supplier or trader is same as if it was node since each node has the 3 roles, which makes Caravela completely P2P.

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]). Then 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 has an offer in the region an `UpdateOffer` message is sent directly to the trader to update the offer free/used resources (lines [6-8]). This update is important to enforce global policies in the container's scheduling (Section 4). If the supplier has 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 [11-12]). Finally, if there exist regions where the supplier does not have

Algorithm 1 Supplier's resource supplying.

```
Data: $suppOfferMap$
1  Function SupplyResources($freeRes, usedRes$):
2      $regions \leftarrow SuitableResourcesRegions(freeRes)$
3      foreach $offer$ in $suppOfferMap$ do
4          $offerRegion \leftarrow Region(offer.TraderID)$
5          if $regions.Contains(offerRegion)$ then
6              $upOffer \leftarrow Offer(offer.ID, freeRes, usedRes)$
7              $UpdateOffer(upOffer)$
8              $regions.Remove(offerRegion)$
9          end
10         else
               /* When node free resources decrease. */
11             $suppOfferMap.Remove(offer.ID)$

12             $RemoveOffer(offer.ID)$
13         end
14     end
15     foreach $region$ in $regions$ do
           /* When node free resources increase. See Algorithm 2. */
16         $PublishOffer(freeRess, usedRes, region)$
17     end
```

any offers, it publishes one offer in each region (lines [14-15]).

To publish an offer the supplier runs Algorithm 2. It starts by creating an offer object with the supplier's resource availability. After that it obtains a **random ID/key** in the offer's target resource region , the random is used to distribute the request's load among the region's traders. With the random node's ID it calls Chord to obtain the trader's IP responsible for that ID/Key.

The trader registers the offer in its internal offer's table and acknowledges it.

Making suppliers and traders save the IP addresses make subsequent contact directly via IP avoiding the more expensive operation of the Chord's lookup.

Once suppliers can provide their resources,

Algorithm 2 Supplier's publish offer algorithm.

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

Caravela discovers available offers for deploying containers running Algorithm 3. It receives the container's necessary resources. The algorithm has 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, a new random ID/key is generated (again to distribute the request's load between region's traders) in the region that represent the smaller resource combination but greater than the necessary resources. Then system then uses Chord to get the trader's IP (responsible for the random key generated), sending a `GetOffers` message to it to obtain the trader's registered offers. If the set of offers received is not empty it returns them, otherwise retries if the threshold was not reached.

Chord's lookup protocol is used 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 show in evaluation that we configured the maximum retries of the discover resources algorithm to only 1 obtaining a very interesting efficiency in the discovery process.

Algorithm 3 Resource discover algorithm.

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

Due to the nodes' crash (frequent 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 $MaxRefreshes_{failed}$ defines how many refreshes a supplier can fail before the trader removes the supplier's offer, and complementary the parameter $MaxRefreshes_{missed}$ define how many refreshes a trader can fail before the supplier publish the offer onto other trader (of the same region).

Suppliers publish resource availability in several regions. Doing otherwise, i.e., publishing it only in the highest one where the resources available are greater, 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.

4 Container's Scheduling

This section describes the container's scheduling on top of the resource discovery process. Before detailing the scheduling algorithm, the **global scheduling policy** and the **request-level scheduling policy** are defined.

Similar to Docker Swarm Caravela 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, it distributes the containers thinly by all the system's nodes. The globally policy configured is applied to all the requests scheduled in the system. All nodes respect the configured policy because every node that joins Caravela receives a copy of the system's configurations. Caravela's configurations are used to configure the bootstrap nodes. All the other nodes that join it receive a copy of it.

Also similar to Docker Swarm, Caravela allows a stack deployment which consist in a composite deployment request where a user can specify a set/group of containers to be scheduled together. This is a common case nowadays with micro services deployments. We extend the stack deployments to allow for 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 scatter request-level scheduling policies. The co-location scheduling policy allows the user to specify that a sub-set of containers in the stack deployment must be scheduled in the same node. The scatter policy can be used by the user to specify that a sub-set of containers must be deployed in different nodes. Co-location is useful for components/containers that communicate a lot or need low latency communications. Scatter is useful for robustness properties.

Note that the spread policy is like the scatter policy, the difference is that the spread policy has a global scope, if it is configured, then all deployment requests respect it. The scatter policy works among the containers of a stack deployment (request-level scope). This rationale also applies to the binpack and co-location policies. The global scheduling policies and the request-level ones are orthogonal e.g., Caravela (as a system) can be consolidating (global-level binpack), while users systematically request stack deployments with all the containers in them configured to be scattered (request-level scatter).

Algorithm 4 Algorithm to schedule containers in nodes.

```
   Data: $globalSchedulingPolicy$
1  Function Schedule($contConfigs$, $resourcesNeeded$):
2      $offers \leftarrow DiscoverResources(resourcesNeeded)$
3      $offers \leftarrow globalSchedulingPolicy.Rank(offers)$
4      foreach $offer$ in $offers$ do
5          $contsStatus \leftarrow Launch(offer.SuppIP, offer.ID, contConfigs)$
6          if $contsStatus$ != $\varnothing$ then
7              return $contsStatus$
8          end
9      end
10     return $Error(\text{“CouldNotScheduleContainersError''})$
```

Algorithm 4 finds the suitable node to deploy a given set of container's configurations (one per container) and the sum of all the resources necessary by all the containers. The first thing the algorithm does is to call the Algorithm 3 with the resources needed in order to obtain a set of resource offers that have free resources equal or greater than the sum of resources needed by all containers. If the set of offers is empty, we return an error to the user (line 10). Otherwise Caravela will rank the set of offers accordingly to the global-level policy configured (binpack or spread).

The adaptation introduces the *CPUClass* attribute in the ranking, because there is not a CPU class notion in Swarm.

Algorithm 5 describes the function called by the node's buyer when the node's owner/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) specifying all the details for each container: image's key, container name, container's arguments, port mappings $\{Host : Container\}$, resources necessary and the request-level policy.

4.1 Optimization: Super Traders

The discovery algorithm, previously described (recall Algorithm 3), selects a random trader in the resource region it targets. When the system is low on resources (few suppliers publishing offers) there are less offers in the traders so the chances of targeting an empty trader is higher. This led 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 resource discovery efficacy and efficiency; the second problem is that global scheduling policies are enforced accurately if the buyer has many offers to rank and choose because it led to better decisions.

To mitigate these problems, we devised a new way to choose 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 affects how we concentrate the `CreateOffer` and `GetOffers` messages in 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 number of super traders can be controlled by the Caravela's configuration parameter $SuperTrader_{factor}$, e.g. the value 7 for the parameter would mean that each super trader would manage the offers of 7 nodes.

5 Implementation

Caravela is a P2P system, so each node contains the same components and runs the same code. Here, we address the inside of Caravela to understand what components are responsible for the algorithms and protocols described in Sections 3 and 4. Caravela is composed by several components such as, Node, User Manager, Remote Client, Docker Client Wrapper, Scheduler, Discovery, Images Storage Client, Containers Manager, Overlay Client, Configuration's Manager and a HTTP WebServer. Figure 4 pictures

Algorithm 5 Buyer's on request deployment algorithm.

```
Function OnDeploymentRequest($containersConfigs$):
    $deploymentFailed \leftarrow false$
    $colocatedResSum \leftarrow NewResources(0, 0)$
    $colocatedConts, scatterConts, deployedConts \leftarrow \varnothing$
    foreach $contConfig$ in $containersConfigs$ do
        if $contConfig.GroupPolicy$ = "Co-location" then
            $colocatedResSum.Add(contConfig.Resource)$
            $colocatedConts \leftarrow colocatedConts \cup contConfig$
        end
        else
            $scatterConts \leftarrow scatterConts \cup contConfig$
        end
    end
    /* Skipped if there are not co-located.                                   */
    $contStatus \leftarrow Schedule(colocatedConts, colocatedResSum)$
    if $contStatus = nil$ then
        return $NewError(``DeployFailedError'')$
    end
    $deployedConts \leftarrow deployedConts \cup contStatus$
    /* Skipped if there are not scatter.                                      */
    foreach $contConfig$ in $scatterConts$ do
        $scatterContRes \leftarrow contConfig.Resources$
        $contStatus \leftarrow Schedule(contConfig, scatterContRes)$
        if $contStatus = nil$ then
            $deploymentFailed \leftarrow true$
            break
        end
        $deployedConts \leftarrow deployedConts \cup contStatus$
    end
    /* Rollback the deployment if necessary.                                  */
    if $deploymentFailed = true$ then
        foreach $cont$ in $deployedConts$ do
            $StopContainer(cont.SuppIP, cont.ContID)$
        end
        return $Error(``DeployFailedError'')$
    return $deployedConts$
```

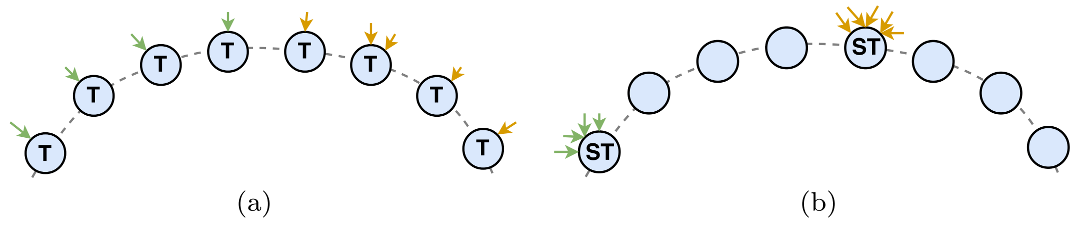


Fig. 3 Super Traders (STs) usage

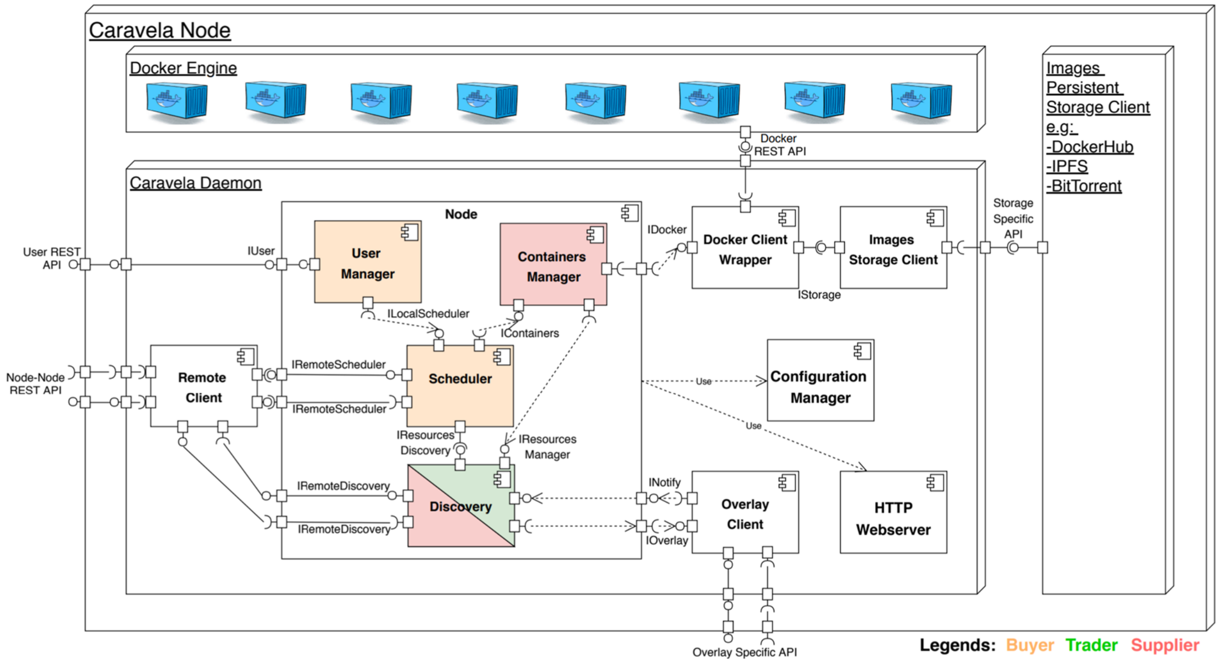


Fig. 4 Caravela's components and interfaces relationships

Caravela's node components, its interfaces and relationships. It also pictures the relations with the external components like the Docker Engine. This picture is used as reference. Caravela prototype is written in Go, Docker and Swarm native language.

The Node component is a component that contains the main logic implementation of algorithms. It exposes the public services of the User Manager (`IUser` interface), Scheduler (`IRemoteScheduler` interface) and Discovery (`IRemoteDiscovery` interface). The User Manager component manages the user's information and requests. It exposes to the outside an interface called `IUser` that is used as the front-end of Caravela for a user.

The Scheduler component drives the containers deployment. It exposes two interfaces: `ILocalScheduler` and `IRemoteScheduler`. The `ILocalScheduler` interface is used by the User Manager component to inject user requests in the system (as a local buyer). `IRemoteScheduler` is exposed to other nodes to allow them (as remote buyers) to send messages to launch containers in other nodes. The Scheduler is responsible for choosing the best node(s) for the container(s)'s deployment. To achieve that, it uses the Discovery component (via `IResourcesDiscovery` interface) to find suitable nodes for the requests that the User Manager component forwards to it. The set of suitable nodes (via offers) that it receives depends on the Discovery component's implementation.

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

The Containers Manager component manages the containers that run in the local's node Docker engine. It exposes the `IContainers` interface for the Scheduler component. This interface is used by the Scheduler component to launch a container in the node on behalf of other nodes. The Docker Client Wrapper, as the name indicates, wraps the Go's Software Development Kit (SDK) for the Docker Engine exposing a very simple interface to be used by the Containers Manager component. We did this to isolate the containers management at Caravela's level from the Docker's API details. The Docker Client Wrapper, as the name indicates, wraps the Go's Software Development Kit (SDK) for the Docker Engine exposing a very simple interface to be used by our Containers Manager component. We did this to isolate the containers management at Caravela's level from the Docker's API details.

6 Evaluation

To evaluate Caravela[3] we developed a cycle-based simulator called CaravelaSim[4], due to the necessity of re-utilizing our Go's code base of Caravela, and at same time test the scalability of the solution with thousands of nodes. We used Go to code Caravela because Docker and all tools around it are written in Go. The simulations were deployed on Amazon Web Services (AWS) Elastic Compute Cloud (EC2) platform, using a c4.8xlarge instance with 36 vCPUs and 60GB of RAM backed up by an Intel Xeon E5-2666 v3 (Haswell) with 25M Cache and 2.60 GHz, running with 20s ticks and a duration of 360 ticks (2h of simulation). The simulator was designed in a way that it was a harness for Caravela's components, which means the simulation ran the Caravela's code with a minor modification to support multi-threaded simulation. We implemented the Chord's protocol as it is in its paper [31], except the background stabilization protocol, due to the simulation overhead.

6.1 Methodology

Two resource discovery and scheduling alternatives were developed to set a baseline. 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 other nodes. This master mode considers all the offers/nodes when deploying containers, so it is a near "oracle" approach that allow it to obtain a near perfect request satisfaction and the near perfect global policy enforcement. The master node is the Chord's node responsible for 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 discovery 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. Finally, our approach detailed in Section 3, is designated as **Multi-Offer** from here forward.

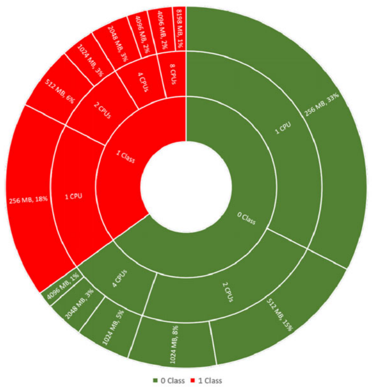
The same request stream (deploy containers requests and stop containers requests) was submitted to all the approaches to maintain the evaluation fair. We submitted the system to a load where at least 50% of its total resources would be used. After that, the request stream had the same deploys and stops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

The distribution of maximum resources available for each node when setting up the Caravela's nodes in the simulator is presented in Fig. 5a. This distribution is the same for Caravela's resources regions configuration parameter. This was determined by what usually happens in volunteer computing environments, where each person only donates the resources he/she has available; so, donating only 1CPU and 512MB of RAM is something normal because the user probably needs the rest for his/her own personal computations. We used that distribution because it is realistic to consider that in an Edge Cloud there are fewer highly capable nodes than weaker nodes. Even with the maximum resources being picked with a pseudo random generator from the presented distribution, we guarantee that all the simulations have exactly the same nodes with the same number of maximum resources because we provide the same seed for all of them. 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), as depicted in Fig. 5b.

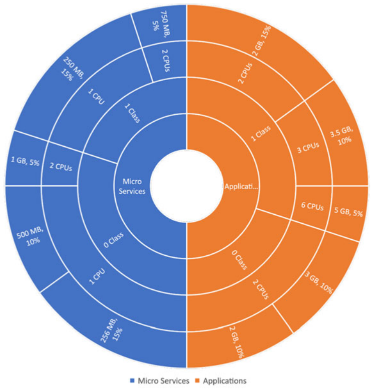
[3]Code available at https://github.com/Strabox/caravela

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

Fig. 5 Distribution of resources regions and request profiles used during the evaluation



(a)



(b)

In the rest of this section we present the results of our evaluation. We gathered 4 main metrics to verify the scalability (without any node being a bottleneck) of our solution and the 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** nodes (which we will refer loosely as 65K for simplicity) and **1,048,576** nodes (referred as 2^{20}) to test the scalability when the network grows 16 times. Note that we do not show higher network sizes because the simulations would take too many hours. We do not present the Swarm results for the 2^{20} network for the same reason. Swarm's simulation time hints that it is not scalable as we will show next.

6.2 Results

Bandwidth consumed per node Figures 6 and 7 show the distribution of the bandwidth consumed per node (on receiving) over time in time windows of 3 minutes. Note that the number of outliers represented in the quartile plots (these and the ones that follows) are [7%-9.5%] of system's total node (super traders). We did not take them out in order to show that not even outliers would be stressed in any way.

Results show that Swarm master node consumes 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. Multi-Offer consumes a bit more than Random, which results from the overhead introduced by more messages to maintain the offer system but would only consume$\approx$150MB in a month span. When the network scales **16 times** the node's bandwidth consumption only increases by a factor of $\approx$1.2 per node (on average), which gives a very good scalability factor to our Multi-Offer approach. It is worth noting that to compute this metric the size of the HTTP messages payload was measured to compare the overheads between the approaches.

For the network size of 2^{20} nodes, the master node would consume $\approx$2.8TB of bandwidth in a month span, which would be unbearable for a user due to ISPs fair usage.

RAM used per node Figures 8 and 9 show the distribution of the RAM used per node over time. Like the previous metric we have [7%-13%] of outliers, which in most cases, correspond to the super traders. We only account for the data structures (without Chord and WebServer structures) to check the overhead of the solutions. 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 nodes participating to schedule the container, it also needs to save the information about each request scheduled in the system.

Deployment Request Efficacy Figure 10 show user's deployment request fulfilled with success (cumulatively) over time. In the end of the simulation Swarm (centralized solution) can only fulfill 4% more requests than Multi-Offer, which mean it can maintain

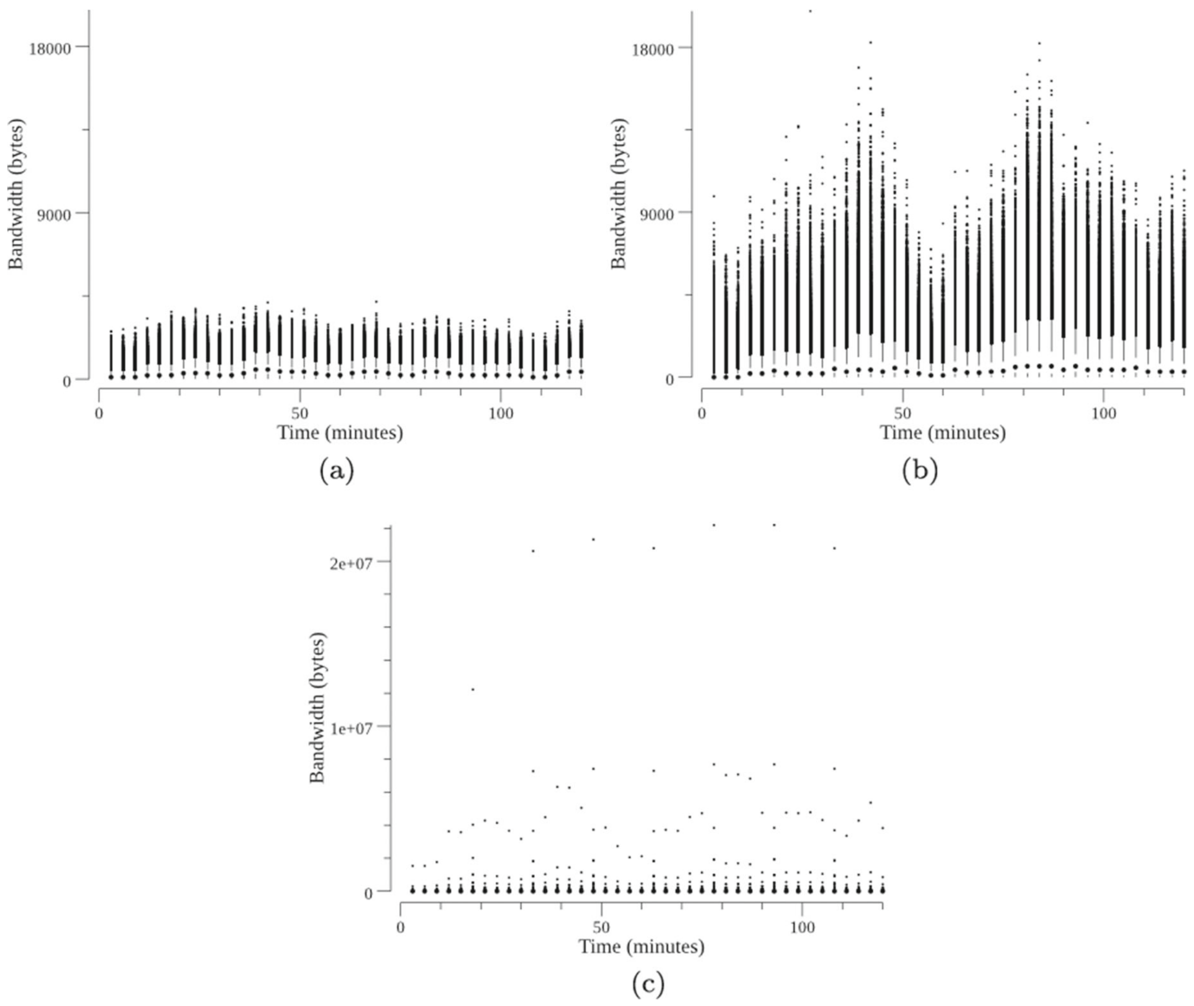


Fig. 6 Bandwidth used per node over time, in the 65K node's network (Quartile Plots)

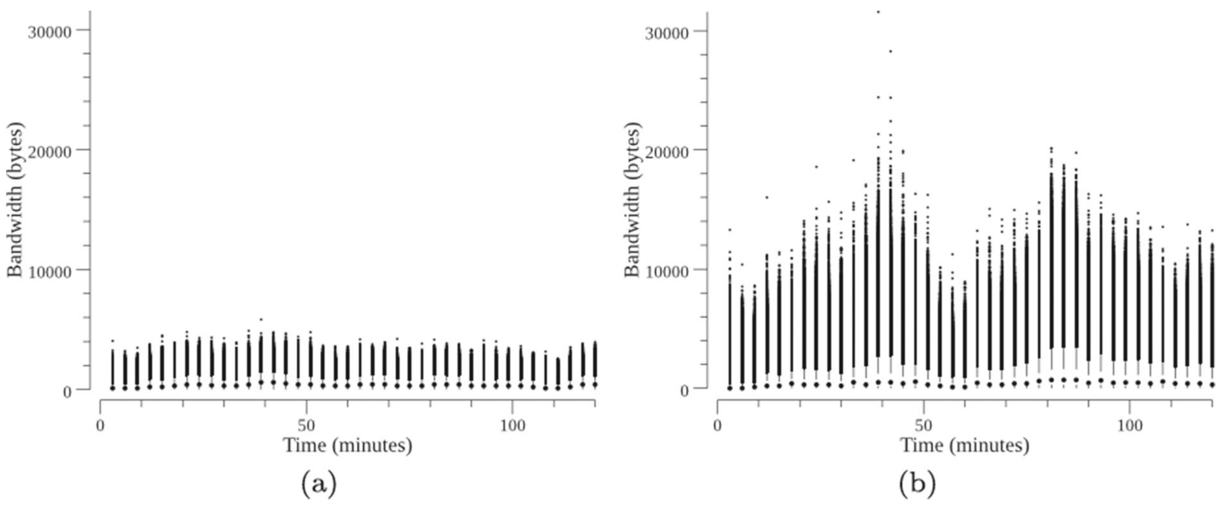


Fig. 7 Bandwidth used per node over time, in the 2^{20} node's network (Quartile Plots)

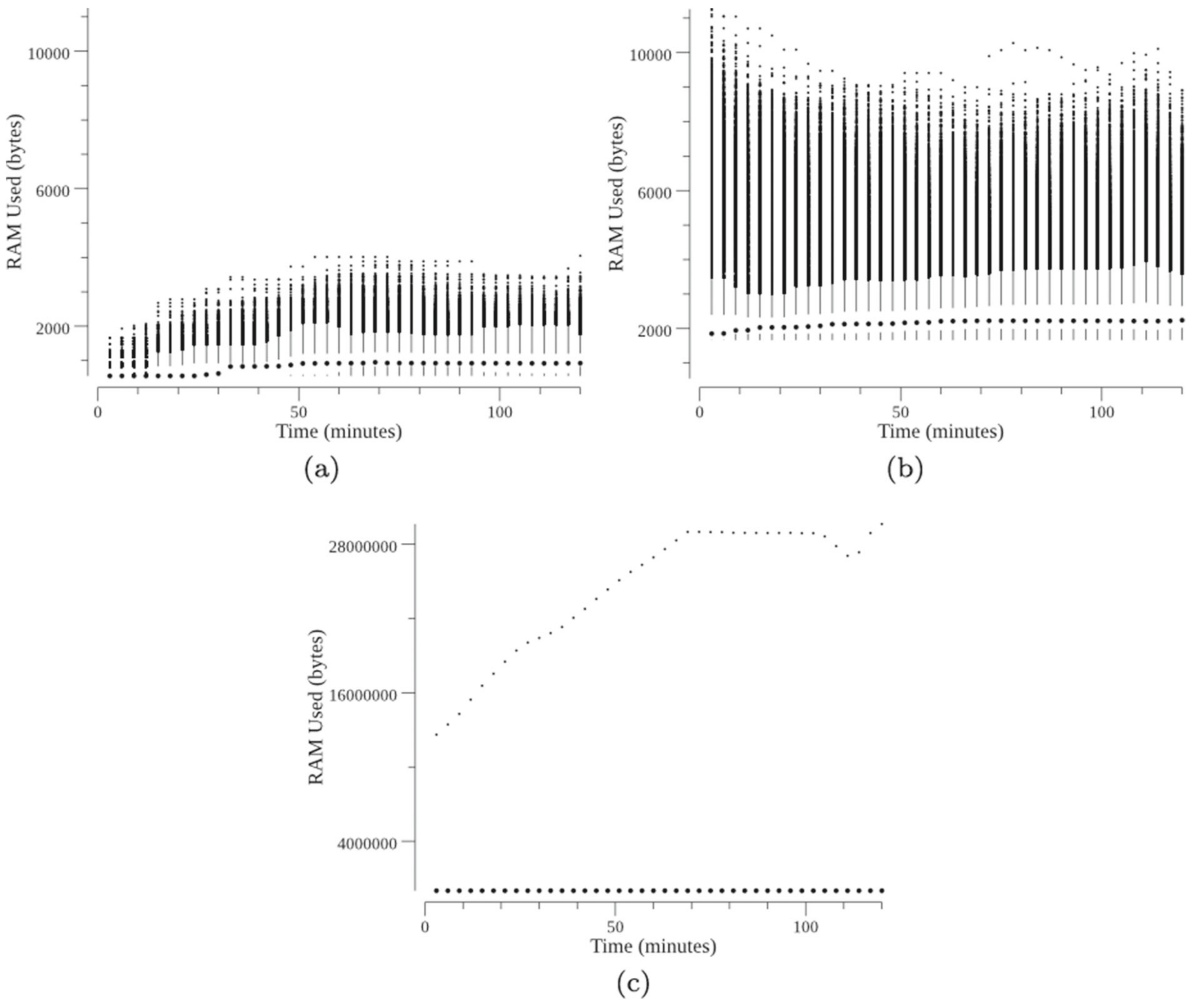


Fig. 8 RAM used per node, in the 65k node's network (Quartile Plots)

a very 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, because we wanted Random to have a decent efficacy to compare, although this gives it a very inefficient resource discovery in terms of network hops, as explained in the next section. The total (red line) represents the total amount of requests submitted into the system (also cumulatively over time).

Deployment Requests Efficiency Figures 11 and 12 show the distribution of sequential messages (network hops), taken until the deployment request succeeded or failed. Swarm has a constant cost of 3 messages because the master nodes save 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 retries to achieve the deployment request efficacy that was discussed before (recall Section 6.2), making its efficiency worst. This metric is important in Edge Clouds

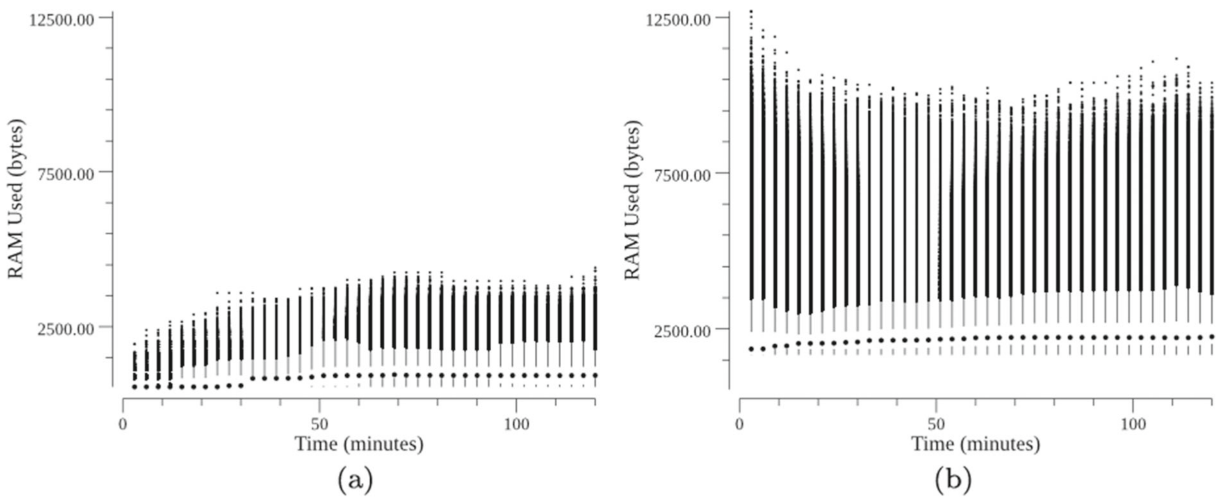


Fig. 9 RAM used per node, in the 2^{20} node's network (Quartile Plots)

due to the WAN networks that connects the nodes. More hops/messages mean 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 the results with the simulation of the Random approach with only 1 retry are not presented. With 1 retry the deployment request efficiency became the same as Multi-Offer because it only uses one Chord lookup too. However the deployment request efficacy presented in Section 6.2 decreased, resulting in having the Multi-Offer solution fulfilling ≈70% more requests than Random.

6.3 Analysis

Based on the previous results we can conclude:

1. The Swarm's master node concentrates too much load in terms of network's usage and computational power used to decide where to place the

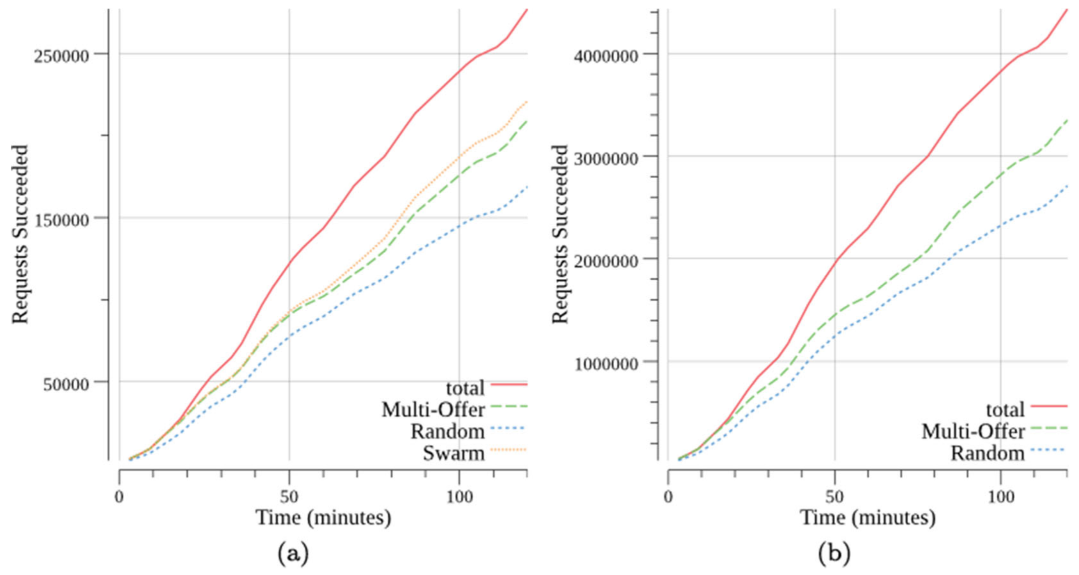


Fig. 10 Deployment requests successfully fulfilled

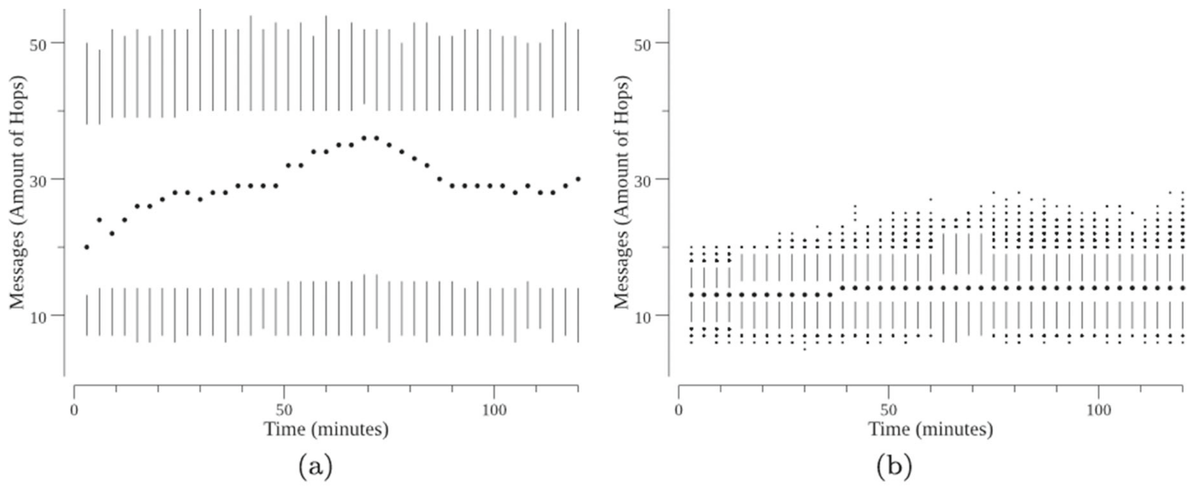
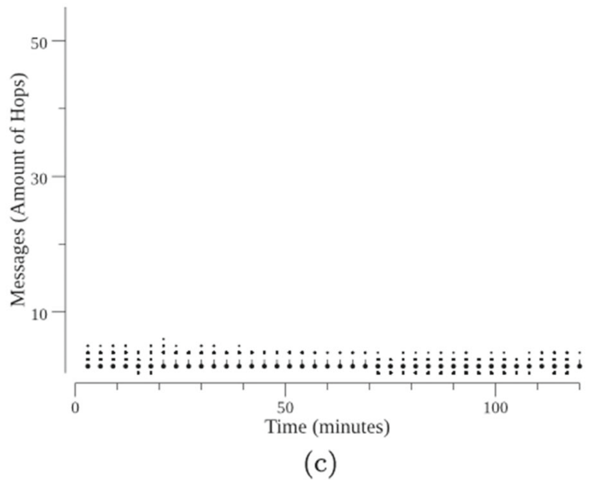



Fig. 11 Distribution of the number of messages exchanged per deploy request submitted, in the 65k node's network

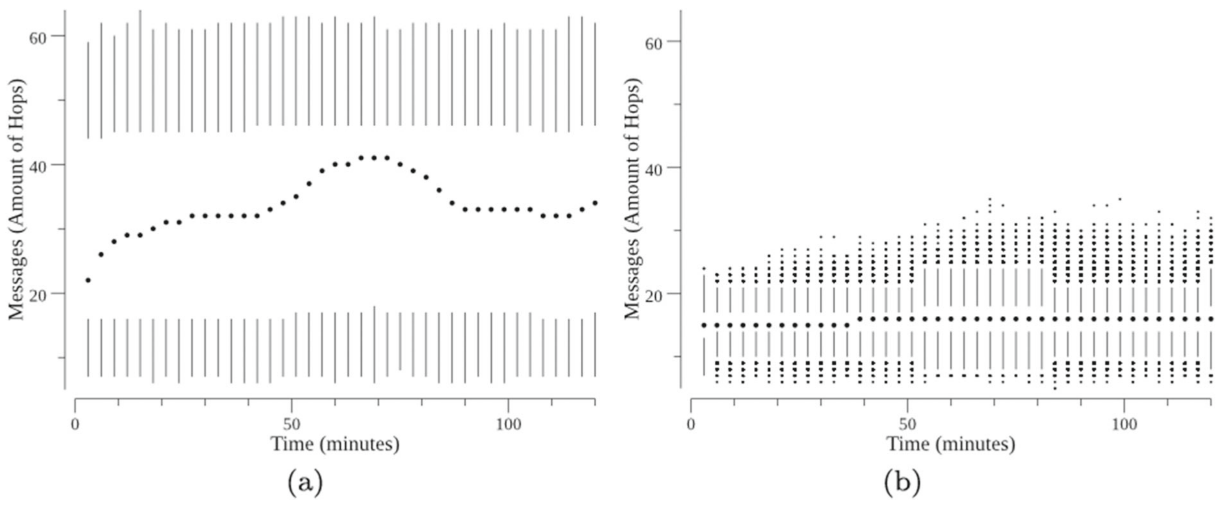


Fig. 12 Distribution of the number of messages exchanged per deploy request submitted, in the 2^{20} node's network

requests, therefore it is unfeasible for networks with tens of thousands of nodes. On the opposite side, for smaller networks it can schedule containers using $\approx$ 3 messages, and it can enforce a global policy "perfectly", choosing the best nodes which is better than the other two approaches;
2. Random is the one that scales better in terms of bandwidth and RAM consumed but, in contrast, its number of requests attended with success, the deployment request efficiency and the efficiency of global resources consolidation are the worst of the three;
3. Multi-Offer has the strengths of the other two, while minimizing the effects of their weaknesses. So, it is an approach that would fit in a deployment of an Edge Cloud with 1M of nodes or even more, while maintaining a very interesting performance in discovering the resources and scheduling the containers.

7 Conclusion

The Internet of Things and a reliable set of cloud services have shown the need to build a new level of computing, Edge Computing, where computation can be made with the computing power available near data sources. Community networks and volunteer computing complement this vision by providing underutilized resources that can be seen as prosumers, in a cloud-like platform, were nodes consume and provide resources.

Current solutions to aggregate computing resources have very centralized internal architectures and algorithms that mostly fit small to 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 in most cases a centralized management prevails. 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 [17].

Caravela was devised to serve as a fully decentralized Docker orchestrator to be deployed in an Edge Computing environment, where there are tens of thousands of nodes participating, high latency between the nodes and no natural 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.

A typical centralized solution is defeated by the scale. The Random approach was scalable with a low overhead per node, but it had extremely low deployment request efficiency. It also cannot enforce the binpack global scheduling policy that is interesting to leverage the maximum of the system/nodes.

Acknowledgements This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference PTDC/EEI-COM/30644/2017 and UIDB/50021/2020, and by Instituto Superior de Engenharia de Lisboa and Instituto Politécnico de Lisboa.

References

1. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: SETI@Home: an experiment in public-resource computing. Commun. ACM **45**(11), 56–61 (2002). https://doi.org/10.1145/581571.581573
2. Apolónia, N., Ferreira, P., Veiga, L.: Trans-social networks for distributed processing. In: Bestak, R., Kencl, L., Li, L.E., Widmer, J., Yin, H. (eds.) Networking 2012 - 11th International IFIP TC 6 Networking Conference, Prague, Czech Republic, May 21-25, 2012, Proceedings, Part I, Lecture Notes in Computer Science, vol. 7289, pp. 82–96. Springer (2012). https://doi.org/10.1007/978-3-642-30045-5_7
3. Apolónia, N., Freitag, F., Navarro, L., Girdzijauskas, S.: Socially aware microcloud service overlay optimization in community networks. Softw. Pract. Exp. **50**(5), 675–687 (2020). https://doi.org/10.1002/spe.2750
4. Babaoglu, O., Marzolla, M., Tamburini, M.: Design and implementation of a P2P Cloud system. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12, p. 412 (2012). https://doi.org/10.1145/2245276.2245357
5. Baig, R., Roca, R., Freitag, F., Navarro, L.: Guifi.net, a crowdsourced network infrastructure held in common. Comput. Netw. https://doi.org/10.1016/j.comnet.2015.07.009 (2015)
6. Benet, J.: IPFS-Content Addressed, Versioned, P2P File System. IPFS-Content Addressed, Versioned, P2P File System (Draft 3). https://doi.org/10.1109/ICPADS.2007.4447808. arXiv:1407.3561 (2014)
7. Bittencourt, L., Immich, R., Sakellariou, R., Fonseca, N., Madeira, E., Curado, M., Villas, L., DaSilva, L., Lee, C., Rana, O.: The internet of things, fog and cloud continuum: Integration and challenges. Int. Things **3-4**, 134–155 (2018). https://doi.org/10.1016/j.iot.2018.09.005. http://www.sciencedirect.com/science/article/pii/S2542660518300635

8. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12, p. 13 (2012). https://doi.org/10.1145/2342509.2342513
9. Cardosa, M., Chandra, A.: Resource bundles: Using aggregation for statistical large-scale resource discovery and management. IEEE Trans. Parall. Distribut. Syst. **21**(8), 1089–1102 (2010). https://doi.org/10.1109/TPDS.2009.143
10. Castro, P., Ishakian, V., Muthusamy, V., Slominski, A.: The rise of serverless computing. Commun. ACM **62**(12), 44–54 (2019). https://doi.org/10.1145/3368454
11. Chang, H., Hari, A., Mukherjee, S., Lakshman, T.V.: Bringing the cloud to the edge. In: Proceedings - IEEE INFOCOM, pp. 346–351 (2014). https://doi.org/10.1109/INFCOMW.2014.6849256
12. Cisco Systems: Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are. White Paper p. 6 (2016). https://doi.org/10.1109/HotWeb.2015.22. http://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf
13. Cunsolo, V.D., Distefano, S., Puliafito, A., Scarpa, M.Huang, D.S., Jo, K.H., Lee, H.H., Kang, H.J., Bevilacqua, V. (eds.): Cloud@Home: Bridging the gap between volunteer and cloud computing. Springer, Berlin (2009)
14. Huedo, E., Montero, R.S., Moreno-Vozmediano, R., Vázquez, C., Holer, V., Llorente, I.M.: Opportunistic deployment of distributed edge clouds for latency-critical applications. J. Grid Comput. **19**(1), 2 (2021). https://doi.org/10.1007/s10723-021-09545-3
15. Kabbinale, A.R., Dimogerontakis, E., Selimi, M., Ali, A., Navarro, L., Sathiaseelan, A., Crowcroft, J.: Blockchain for economically sustainable wireless mesh networks. Concurr. Comput. Pract. Exp. 32(12). https://doi.org/10.1002/cpe.5349 (2020)
16. Kargar, S., Mohammad-Khanli, L.: Fractal: An advanced multidimensional range query lookup protocol on nested rings for distributed systems. J. Netw. Comput. Appl. **87**, 147–168 (2017). https://doi.org/10.1016/j.jnca.2017.03.021. http://www.sciencedirect.com/science/article/pii/S1084804517301303
17. Khan, A.M., Freitag, F., Rodrigues, L.: Current trends and future directions in community edge clouds. In: 2015 IEEE 4Th International Conference on Cloud Networking, Cloudnet 2015, pp. 239–241 (2015). https://doi.org/10.1109/CloudNet.2015.7335315
18. Kochovski, P., Stankovski, V., Gec, S., Faticanti, F., Savi, M., Siracusa, D., Kum, S.: Smart contracts for service-level agreements in edge-to-cloud computing. J. Grid Comput. **18**(4), 673–690 (2020). https://doi.org/10.1007/s10723-020-09534-y
19. Mayer, P., Klarl, A., Hennicker, R., Puviani, M., Tiezzi, F., Pugliese, R., Keznikl, J., Bure, T.: 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, pp. 89–94 (2014). https://doi.org/10.1109/SASOW.2013.16
20. Mell, P., Grance, T.: The NIST definition of cloud computing recommendations of the national institute of standards and technology. Nist Special Publication **145**, 7 (2011). https://doi.org/10.1136/emj.2010.096966
21. Mohan, N., Kangasharju, J.: Edge-fog cloud: A distributed cloud for Internet of Things computations. 2016 Cloudification of the Internet of Things CIoT 2016. https://doi.org/10.1109/CIOT.2016.7872914 (2017)
22. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. www.bitcoin.org p. 9. https://doi.org/10.1007/s10838-008-9062-0. https://bitcoin.org/bitcoin.pdf (2008)
23. Peinl, R., Holzschuher, F., Pfitzer, F.: Docker cluster management for the cloud - survey results and own solution. J. Grid Comput. **14**(2), 265–282 (2016). https://doi.org/10.1007/s10723-016-9366-y
24. Pouwelse, J., Garbacki, P., Epema, D., Sips, H.: 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, pp. 205–216. LNCS (2005). https://doi.org/10.1007/11558989_19
25. Ryden, M., Oh, K., Chandra, A., Weissman, J.: Nebula: Distributed edge cloud for data intensive computing. In: Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014, pp. 57–66 (2014). https://doi.org/10.1109/IC2E.2014.34
26. Satyanarayanan, M., Bahl, P., Cáceres, R., Davies, N.: The case for VM-based cloudlets in mobile computing. IEEE Pervas. Comput. **8**(4), 14–23 (2009). https://doi.org/10.1109/MPRV.2009.82
27. Selimi, M., Cerdà-Alabern, L., Freitag, F., Veiga, L., Sathiaseelan, A., Crowcroft, J.: A lightweight service placement approach for community network micro-clouds. J. Grid Comput. **17**(1), 169–189 (2019). https://doi.org/10.1007/s10723-018-9437-3
28. Selimi, M., Cerda-Alabern, L., Sanchez-Artigas, M., Freitag, F., Veiga, L.: Practical service placement approach for microservices architecture. In: Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, pp. 401–410 (2017). https://doi.org/10.1109/CCGRID.2017.28
29. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: Vision and challenges. IEEE Int. Things J. **3**(5), 637–646 (2016). https://doi.org/10.1109/JIOT.2016.2579198
30. Singh, S., Chana, I.: A survey on resource scheduling in cloud computing: Issues and challenges. J. Grid Comput. **14**(2), 217–264 (2016). https://doi.org/10.1007/s10723-015-9359-2
31. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup protocol for Internet applications. IEEE/ACM Trans. Netw. **11**(1), 17–32 (2003). https://doi.org/10.1109/TNET.2002.808407
32. Vaquero, L.M., Rodero-Merino, L.: Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. ACM SIGCOMM Comput. Commun. Rev. **44**(5), 27–32 (2014). https://doi.org/10.1145/2677046.2677052
33. Varghese, B., Buyya, R.: Next generation cloud computing: New trends and research directions. Future Gener. Comput.

Syst. **79**, 849–861 (2018). https://doi.org/10.1016/j.future.2017.09.020. http://www.sciencedirect.com/science/article/pii/S0167739X17302224

34. Verbelen, T., Simoens, P., Turck, F.D., Dhoedt, B.: Cloudlets : Bringing the cloud to the mobile user. In: Proceedings of the third ACM workshop on Mobile cloud computing and services, pp. 29–36 (2012). https://doi.org/10.1145/2307849.2307858
35. Vishnumurthy, V., Chandrakumar, S., Emin, G.: Karma: a secure economic framework for peer-to-peer resource sharing. In: Workshop on Economics of Peer-to-peer Systems, p. 34 (2003)
36. Yang, S., Butt, A.R., Fang, X., Hu, Y.C., Midkiff, S.P.: A fair, secure and trustworthy peer-to-peer based cycle-sharing system. J. Grid Comput. **4**(3), 265–286 (2006). https://doi.org/10.1007/s10723-006-9039-3