TÉCNICO LISBOA

RATEE - Resource Auction Trading at Edge Environments

Diogo Paulo Dias

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Luís Manuel Antunes Veiga
Prof. José Manuel de Campos Lages Garcia Simão

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Luís Manuel Antunes Veiga
Member of the Committee: Prof. Rolando da silva Martins

January 2021

Dedicated to me

Acknowledgments

I would like to thank my supervisor Prof. Luís Veiga for his support and share of knowledge to make this thesis possible.

Then I would like to thank my family, especially my parents and my cousin, for their support and encouragement.

Lastly but not less important, I would like to thank Discord's people (they know who they are), for their ideas and their support.

Thank you all, I will be here for you.

Resumo

Tem havido um aumento do uso da *Cloud* para multiplos usos, tais como de computação e de armazenamento. Para conseguir fornecer estes serviços para milhões de pessoas com uma grande confiabilidade são necessário grande *data-centers*. Mas isto tem as suas limitações, tais como a largura de banda, porque os dados são todos transportados para esses *data-centers*, e também latência, devido à distância entre esses *data-centers* e os dispositivos pessoais. Com isso emergeu um novo paradigma the *Edge Computing*. Este paradigma envolve realizar computações ou outros tipos de operações em dispositivos mais próximos dos dispositivos pessoais dos utilizadores. Estes dispositivos tem menos potência que os *data-canter* mas fornecem um menor latência e como se encontram mais perto dos *end-users*, os dados são imediatamente processados. Os recursos destes dispositivos são alocados para hospedar applicações. Estas applicações podem ser alocadas de várias formas, voluntariamente, ou por troca. A que vamos nos focar é usar pagamentos, mais especificamente, o uso de leilões para alocar recursos para uma aplicação. Neste trabalho criamos um prototipo, chamado *RATEE* que tem como proposito alocar recursos (que são usados para fazer *deployment* de aplicações) usando leilões como mecanismo de troca entre os recursos e o pagamento para obte-los. Este prototipo é usado em redes *Edge*, próximas do utilizadores, e é descentralizado, a informação não se encontra armazenada num ponto.

Palavras-Chave: *Edge Computing*, P2P, Leilões, Docker, Aplicações Distribuidas.

Abstract

Right now we use the Cloud for multiple operations, such as computing and storage. To be able to provide these services to millions of people with great reliability, large data centers are needed. But these data centers have its limitations, such as bandwidth, because the data is all transported to these data centers, and also latency, due to the distance between those data centers and personal devices. With that limitations a new paradigm emerged, Edge Computing. This paradigm involves performing computations or other types of operations on devices closer to users' personal devices. These devices have less power than data-centers but provide a shorter delay and as they are closer to end-users, data is immediately processed. We could use the resource of those devices in order to deploy applications due to them being underutilized. These applications can be allocated in various ways, voluntarily, or by exchange. What we are going to focus on is using payments, more specifically, using auctions to allocate resources to an application. In this work, we created a prototype called RATEE that aims to allocate resources (which are used to deploy applications) using auctions as a mechanism for exchanging resources and paying to obtain them. This prototype is used in Edge networks, close to the users, and is decentralized, the information is not stored in a single point.

KEYWORDS: Edge Computing, P2P, Auctions, Docker, Distributed Applications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud Computing is a technological environment heavily used because of its properties like global access, pay for what you use, resource elasticity, and others [Pet11]. To meet these alluring properties, typically Cloud Computing is supported with big data centers at the Internet's backbone. This causes Cloud Computing to work mostly at a long distance from the Internet's edge with Wide Area Network latencies and expensive bandwidth for data to reach it.

The Internet Of Things paradigm is rising to new levels, enabling new concepts such as smart cities. A smart city is one which uses information and communication technologies to make the city services and monitoring more aware, interactive and effective [Jio14]. This information feeds the network's edge with data, which is then transfered to the Cloud data centers in order to be processed and stored. The transfer of large amounts of data will cause bandwidth saturation and a high latency [CWSR12]. An effective pre-processing mechanism at the network's edge would reduce the amount of data to send (to be processed or stored at Cloud) reducing the bandwidth consumption and the latency to transport that data. The computing power and storage are also growing at the networks edge: Raspberry PI [Cla16], [Pao16], laptops, desktops, routers, hubs and others. Also, those edge devices can be used to do some processing that uses sensitive data. Instead of all that data goes to the Cloud provider and be processed there. Some privacy concerns may arise when some sensitive data is processed at the Cloud providers. Most of the time these resources are actually underutilized. This inefficiency is opening the doors to new studies, tools and migrations to the Edge Computing in order to provide services with low latency and low bandwidth requirements.

1.1 Motivation

There are still many difficulties in handling edge cloud resources. A structured view of the existing resources, their characteristics and the role of all entities involved in the edge cloud environment is vital. In addition, reconfigurations often need to be performed in order to modify or allocate existing virtual resources, depending on the usage or the service level agreement. Inefficient allocation of resources has a detrimental impact on performance and costs and also impact on the usability of the system.

Market-based resource allocation. Developing resource management techniques that guarantee scalability, performance, manageability and adaptability for the edge cloud environment is crucial to resolve the aforementioned challenges. Traditional approaches, such as system optimization, focus solely on system performance metrics rather than economic factors, such as revenue, cost, income, and profit [Ngu17a]. Comparing with the system optimization approach, economic approaches and pricing can provide the following advantages:

1. The demand for resources depends on the needs of the users. Also, the resources provided depend on the capacity and needs of the providers. There may be times when the demand is higher than the supply or vice-versa. Pricing/economical strategies can be used to solve the problem of scarce or abundant resources originated from dynamic demand and supply prices.

2. There are various entities, e.g. stakeholders, end-users, cloud providers, in edge cloud environment that have different objectives, e.g. cost, profit, revenue, income, utility, performance, scalability, as well as different constraints, e.g., the budget and the technology. There are times when these objectives often clash with each other, and this conflict can be efficiently overcome with an economical/price model. Using economic/pricing models for negotiation mechanisms can result in optimal solutions for entities with different objectivies, achievied in a mostly decentralized manner.

3. In edge cloud environments, the resource providers' profit must be maximized while fulfilling the client requirements. For this reason, price models based on cost minimization and benefit maximization may be used.

4. One of the most important services in the cloud is Video on Demand. This is a service which offers video for people to watch, e.g., Netflix, HBO, Youtube. These providers offer tons of terabytes of media, overwhelming the networks' bandwidth. Price/economic

approaches, e.g. smart data pricing, have been used to regulate user demands and have an efficiently use of bandwidth. For areas with lower resources capabilities may have greater prices to reduce the consume.

Therefore, economic and pricing approaches for resource management have been researched, developed and sucessfully adopted to manage cloud computing deployments.

Cloud Containerization. In order to promote the use of multiple technologies and the deployment of multiple technologies in a heterogeneous environment, it is necessary to have virtualization, isolation, and security. The most used tools to promote virtualization are System Virtual Machines, managed by hypervisors (e.g., ESXi, Xen, QEMU, ...), or containers (e.g., Docker[1], Warden Container, OpenVZ, LXC containers), which are much more lightweight. In Table 1.1 we show a comparison between Virtual Machines and Containers in several quality attributes. The first one compares the operative system that is running in each sandbox environment. Then the protocols that can be used to communicate between sandbox of the same type. The performance by running the same program. The time it takes to startup time. At the end the storage space.

Parameter	Virtual Machines	Containers
Guest OS	Each VM runs in its virtual hardware and Kernel is loaded into its memory region	All the guests share the same OS and Kernel. Kernel is loaded into physical memory.
Communication	Will be through Ethernet Devices	Standard IPC mechanisms like Signals, pipes, sockets, etc...
Performance	Virtual Machines suffers from a small overhead due to the translation of guest OS instructions to host OS instruction	Containers provide near native performance
Startup time	Virtual Machines take a few minutes	Containers take a few seconds
Storage	Virtual Machines have more storage as the whole OS and the programs that are associated to run	Containers images have lower storage consumption due to the base OS being shared

Table 1.1: Virtual Machine and Container feature comparison, based on [Raj14]

Due to the advantages of Containers (mainly performance, startup, and storage), their popularity is increasing and started being used at large scales in cloud couply deployments.

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

1.2 Shortcomings of current solutions

There is a lack of decentralized edge-based cloud solutions. Most of cloud solutions: OpenStack[2], OpenNebula[3], Swarm[4] and others, have centralized architectures and operate only in a controlled environment (not using volunteer solutions). The use of market algorithms to allocate resources in a volunteer environment has been studied, and techniques to improve fairness, utility and truthful price have been applied, but there is a lack of auction-based solutions in a peer-to-peer environment [Sau11].

1.3 Document Roadmap

The rest of the document is organized as follows: in Chapter 2 it is presented the study and analysis of related work. In Chapter 3 we explain our solution to address the shortcomings mentioned before. In Chapter 4 we talk about our code, and the implementation of our system. In Chapter 5 we describe the metrics and techniques used to evaluate our solution; in Chapter 6 we wrap up all the important information and present some concluding marks.

[2]https://www.openstack.org/
[3]https://opennebula.org/
[4]https://docs.docker.com/engine/swarm/

Chapter 2

Related Work

In this section we will discuss Edge Clouds in Section 2.1, describing its taxonomy, what aspects differ across various environments of Edge Cloud and in the end we classify some tools based on the taxonomy we proposed. The second theme we explore is the different Market Models in Section 2.2. There are many different market models, and each one has is own advantages and disadvantages and differ depending on the usage context.

2.1 Edge Clouds

In this section we will present the main design dimensions of an Edge Cloud environment and the different values/types for those dimensions. These dimensions are **Resource Ownership**, **Architecture**, **Service Level**, **Target Application** and **Access Technology**. The taxonomy can be seen in Figure 2.1.

Resource Ownership is the dimension which represents the owner type of the resources. Resource Ownership can be divided in three types: Single Owner, Volunteer and Hybrid.

In Single Owner, the devices used to support an Edge Cloud environment are owned by a single entity. Normally these entities already have a group of specific devices and already are configured for the purpose of sharing their own resources to create an edge network. This type of ownership is normally used in big companies which implement their own edge computation infrastructure. They have more control of the system, having more security, reliability and other quality attributes, due to: controlling all the devices and making their own configurations, controlling what types of work will be computed, and the protocol used to communicate between the devices and others.

The Volunteer type differs from the previous one. The resources are shared by end-users

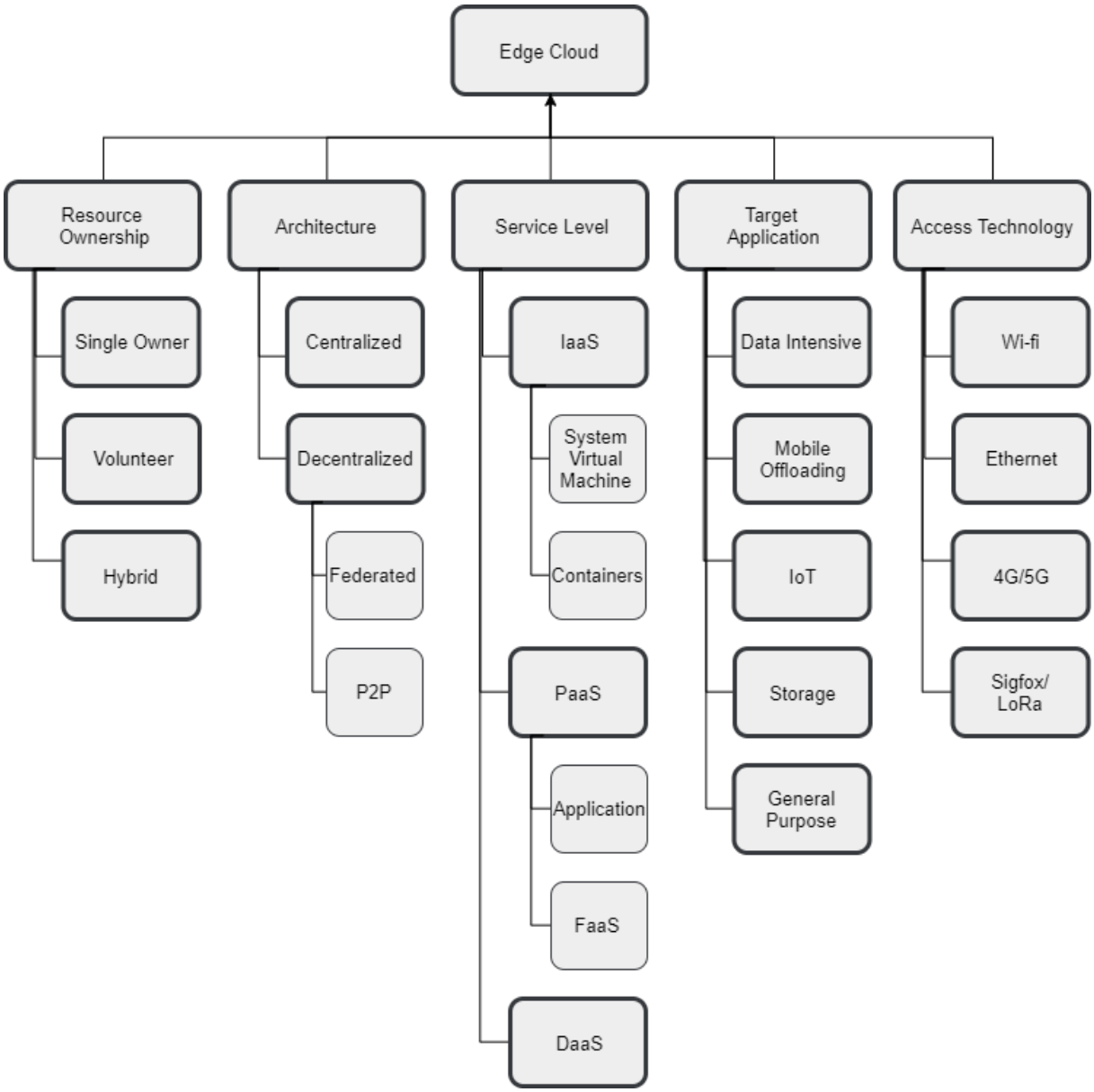

Figure 2.1: Edge Cloud Taxonomy

and these resources are normally their personal devices like computers, tablets, mobile phones, and others (e.g. Costa et al. [Fer13], Cloudlets [Tim12], Satyanarayanan et al. [Mah09], Cloud@Home [Vin09], Babaoglu et al. [Oza12] and Mayer et al. [Phi13]). One of the biggest differences in this type is that the devices aren't owned by a single entity. Due to that, enforcing a Service Level across multiple authoritative domains is challenging and prone to data leakage. This type of ownership may have more end-devices if almost every person shares the resources of their own devices. But these devices (normally personal) could have lower capabilities resources in contrast with Single Owner because the purpose is to share devices, so, they normally buy devices with high requirements.

The last one is a Hybrid type and is a mix of the last two. In this environment, some resources belong to a single entity while others belong to users who share their resources voluntarily. This type of environment with single and volunteer ownership normally happens when comes from personal devices like computers, mobile phones, and the single ownership comes from Internet Service Provider devices like routers and hubs (e.g. Nebula [Alb17], Chang et al. [Hyu14] and Mohan et al. [Nit16]). This type brings much more power resources to the Edge Cloud but suffers from the disadvantage that some resources may be owned by some malicious users. Volunteer Ownership also suffers from this disadvantage.

Architecture is another relevant design decision dimension. This dimension can be divided in two main types: Centralized and Decentralized. When we are speaking about Architecture we are speaking about the managers/orcherstrators distributed architecture.

Centralized architectures have dedicated nodes to manage and orchestrate all the resources of the Edge Clouds. One problem that arises with this approach is the bottleneck which can be created by the large number of messages between the managers and the worker devices. Also, it suffers from Single Point of Failure issues, i.e., if the managers crash the work stagnates.

Decentralized architectures are the opposite, as all nodes are equal and there isn't a privileged group of managers or orchestrators. This type of architecture tends to scale out better for having a better distribution than the previous one, reducing the bottlenecks and it doesn't suffer from a single point of failure. Decentralized architectures type is divided in two categories: *Federated* and *P2P*. In Federated we have autonomous small clouds (also designated zones in some articles [Ami15]) which provide services. These small clouds can group with other autonomous small clouds to supply a greater/powerful service. Email is a great example of a federated architecture. Similar, a Peer to Peer architecture makes all nodes equal in terms of responsibilities/work, even if some fails the cloud continues operating (e.g. Babaoglu et al. [Oza12]), and they interconnect themselves creating one Edge Cloud environment. Some nodes may have some special roles like bootstrap that helps new users to connect to the network. In some works, they use peer-to-peer architecture to implement federated networks. Moving this topology to a sub-set of peer-to-peer.

Service Level is a dimension that refers to the type of service which is provided by a service in an Edge Cloud environment. Similar to Cloud Computing, an Edge Cloud infrastructure offers similar services and for those services different levels. The three main types are Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Data-as-a-Service (DaaS).

IaaS provides infrastructure with CPU, RAM and network capabilities. As infrastructure, we have two different categories: *System Virtual Machines* and *Containers*. In the Virtual System Machine, the devices run a Virtual Machine like VMWare or Oracle Virtual Box, and

on top of that runs the operating system and applications supplied by the client. The System Virtual Machine enables to run the clients' applications in a sandbox environment, and also, the client can deploy multiple applications, which can have a relation between them, in one instance of a virtual machine. Virtual Machines use a component called Hypervisor. It is a piece of software that enables the virtualization between the machine and the operative system that they are running. There are two types of hypervisors: type 1 and type 2. In type 1, the hypervisor runs on bare metal, they are the machine's operating system, and on top of that runs the virtual machines. Xen[1] is one example of type 1 hypervisor. The type 2, the hypervisor runs on top of an operative system. Type 1 has better performance results because there isn't the overhead of the machine operative system. But type 2 is sometimes more frequently used because normally we have machines that have already an operating system because they are also used for other purposes, and not only for virtualization. Using Container technology, the clients only need to provide the application code and their dependencies. Containers are a newer technology that is emerging and is being preferred over System Virtual Machines because they package the application's code and that image is more lightweight than the full system image used to deploy in a System Virtual Machine.

This difference in the image size between a container and a system virtual machine is due to the containers, already use and share the operative system that is running on the machine. Due to containers sharing the same operative system, it is more prone to attacks than the system virtual system machines, where each one has its respective operating system (but being much more heavy-height).

PaaS already supplies the runtime platform (Common Language Runtime, Java Virtual Machine, WebAssembly which is gaining great popularity, etc) and the client only needs to send the application code to run remotely. This code differs depending on what PaaS level (which are two) we are using. If it is *Application*, the user must send all the application's code and its dependencies to be deployed (.jar, .exe, .dll) as a self-contained program. The scalability is managed by the PaaS's provider. The other type is Function-as-a-Service (*Faas*) [Mic17], [Ale17], a new service with rising popularity, and is also named *serverless architecture*. In this type of service, the client only needs to implement some functions to deploy his own application. These functions only contain business logic and are stateless. All the data is saved persistently in databases or blobs and all the logic to receive requests and the code to scale out/in is handled by the FaaS's provider.

DaaS is a service where the provider is offering data storage to the end-users. In contrast

[1] https://xenproject.org/users/virtualization/

with the other types of services, this service doesn't involve computation, the user only needs to provide the data that needs to be stored remotely. The data that is being stored can be files [Ant01] or structured data [Avi10]. Also, different types of data services offers different properties. These properties are based on CAP[2] theorem, where we can only choose two of these three quality attributes: consistency, availability and partitioning. For consistency, we may have strong consistency, where the information is always seen in the same way for all users, or it can be eventually consistent. Where eventually all users that want to obtain the same information, will see the same result.

Target Application refers to the type of application we want to deploy on the Edge Cloud. These applications take advantage of some characteristics of Edge Clouds, like low latency. We have five types of relevant applications which could use Edge Clouds: *Data Intensive*, *Mobile Offloading*, *IoT*, *Data Storage* and *General Purpose*.

Data Intensive applications are applications that process large chunks of data. These processes may be cleaning, transformation to other data type/models or aggregation. The chunks of data can't all be processed in one device, so, they are divided into multiple end-devices to distribute the work, augmenting performance. By distributing the data chunks, the network will suffer a higher bandwidth demand which will also cause latency by the processing of moving data. To reduce these disadvantages, data intensive applications exploit the geo-location [Alb17] of the data sources themselves and try to process the data on the closest nodes to avoid the overhead and cost of transfering the data.

Mobile Offloading applications are primarily used by mobile devices or devices which have lower hardware specifications than those required to compute some work. These lower specifications result from the trade-offs in small and mobile devices. To do heavy work in these environments, mobile phones offload their work to end-devices residing on an Edge Cloud infrastructure to make computations on behalf of them. This preserves the battery of mobile phones and the computation is much faster due to better hardware specifications. The Cloudlet work [Tim12] attempts to offer a transparent method for offloading mobile application components to Edge Clouds so as to use the apparently vast amount of resources.

IoT applications are similar to data intensive applications but in this case, while the messages have a smaller payload, the number of messages is much higher. IoT devices try to have a long lifetime to reduce the maintenance and to do that, they take into account the battery drain on daily use. To reduce the battery consumption they send, periodically or eventually, small messages. But a city can have a big number of IoT devices, resulting a large number of messages,

[2] `https://en.wikipedia.org/wiki/CAP_theorem`

burdening the network. In order to reduce this burden, some applications aggregate the data in the network's edge before reaching the Cloud [Nit16].

Storage applications [Lan03], [Ant01] differ from the previous ones because they don't have computation. Their purpose is only to store and retrieve data. Using the millions of devices of an Edge Cloud environment they can replicate the data having high availability and reduce the consistency. One downside is also the data may be inspected when stored in malicious' users end-devices. To have fast retrieval, data's location can be used to find the nearest nodes with the required data.

General Purpose applications are the last all-encompassing type. This type covers all the other applications that didn't fit the previous types. These types of applications can be simulators, servers, and others.

Access Technology is the dimension referring to the technology used to access edge cloud services. One way is by using Ethernet protocol, if the service is near and a physical connection can be used. Another similar approach, but without any physical connection, is by using Wi-Fi, more practical to the end-users (being wireless) but can suffer more attacks from spoofing. Mobile devices can also use 4G/5G communication, for example, to offload work to edge cloud end-devices. IoT devices can also use Wi-Fi, but this type of protocol has a high battery consumption. To get high efficiency in the communications other communications protocols like SigFox[3]/LoRa[4] which target IoT devices specifications.

Of the design choices of the work In Table 2.1 we have a summary. This classification is based on the taxonomy presented in Figure 2.1. Fields with value '-' mean no information was found or given about that entry. Because of the frequent lack of specific information about Access Technology (relying on higher level protocols such as TCP/UDP over IP), this dimension was removed from the table.

2.2 Market Models

Market models have been used to solve many issues in cloud environments, solving some challenges and having the advantages mentioned in Chapter 1. Different market models offer different characteristics and provide different specifications. To choose the market model to use in one service it is first necessary to make a study about the domain where that application will exist and the consumers of that service.

In this section, we will study different market models, their characteristics, and their advantages and disadvantages. We divided the market models into two categories, based on how the

System/Tool	Resource Ownership	Architecture	Service Level	Target Application
Edge-Fog Cloud [Nit16]	Hybrid	P2P	-	IoT
Chang et al. [Hyu14]	Hybrid	Federated	Containers	General Purpose
Cloudlets [Tim12]	Volunteer	Federated	Application	Mobile Offloading
Cloud@Home [Vin09]	Volunteer	Centralized	SVM	General Purpose
Satyanarayanan et al. [Mah09]	Volunteer	Federated	SVM	Mobile Offloading
Mayer et al. [Phi13]	Volunteer	P2P	Application	General Purpose
Nebula [Alb17]	Hybrid	Centralized	FaaS	Data Intensive
Babaoglu et al. [Oza12]	Volunteer	P2P	SVM	General Purpose
Samsara [Lan03]	Volunteer	P2P	DaaS	Storage
Past [Ant01]	Volunteer	P2P	DaaS	Storage

Table 2.1: Edge clouds tools dimensions classification

prices are set: Provider-Based Pricing and Auction-Based Pricing.

Provider-Based Pricing

In Provider-Based Pricing, the prices are set by the resource provider. We address three types of provider-Based Pricing, e.g., cost-based pricing, differential pricing, and profit maximization, where the prices are set by the provider but the approach to find these prices differs from each other.

Cost-based pricing: This technique is a popular pricing strategy to calculate a resources' price based on calculating the resources' total cost and adding, as a desired benefit, a value e.g. percentage of the cost or constant value. The objective of this strategy is for the resources' provider, ensure some revenue or, in the worst case, guarantee the minimum price to be equal to the cost of providing the resources. The total-cost is created from the fixed cost and the variable cost. The fixed cost is the cost that does not change depending on the supply or the number of requests. These are normally hardware costs, e.g. RAM's price, CPU's price, disk storage's price. In contrast, the variable cost varies with the number of requests or the service provider, e.g. bandwidth used, the disk used, number of servers, and energy.

One of the advantages of the cost-based pricing is the easy way of setting the price, being one function of the internal cost, e.g. the cost to generate the service [Cos03]. One of the disadvantages is that the price does not take into account the price of other providers (provider A may offer the same service with the lowest pricing, getting more sells) or the value which the users are willing to pay. Another disadvantage may be the precision to calculate the variable cost. It is necessary to have good metrics and a good monitor to convert those metrics to a cost. This technique has been mostly used to calculate the service cost in geo-distributed data

centers [Alb08], [Sha10].

Differential pricing: the previous market model, cost-based pricing, does not take into account the value users are willing to pay. This misuse of information reduces substantially the market area of users we want to target. To maximize the profit of providers, it is necessary to attract these types of users. Therefore, it is necessary to know the requirements of these users, know how much they are willing to pay, and find a good price for them.

This technique is called differential price because the provider, may offer resources at different prices to different users, depending on the information aforementioned. The user surplus here is the difference between the overall amount of money users are willing to pay and the total amount of money they pay.

This strategy brings a greater profit for the provider can be considered unfair to the users, e.g. one user may pay a much higher value than another for almost similar services. One of the examples of differential prices in the cloud market is the Alibaba[5] group, which offers a discount to use their services, but they need to pay for one year. Another application of differential pricing is used in bandwidth location for dynamic demands in data center networks [Din14].

Profit maximization: The profit maximization is the usual technique used to maximize the number of resources shared and the corresponding price to get the highest profit possible to the provider. To apply that technique we assume we have the number of resources shared, denominated Q, and the price to share each resource, denominated P. The profit of a provider is given by the formula, π = R(Q,P) - C(Q), where the π is the profit, the R is the revenue of one provider based on a number of resources shared and the price to share that resources, and the C is the cost to operate and share the resources. The cost could be divided into fixed cost and variable cost (these costs were explained in the Cost-based pricing technique).

We want to find the optimal Q* where the profit is maximized Q* = max(π). With the optimal Q* we find the optimal price, to share our resources, based on demand curve. The demand curve is a linear function and represents how much the clients are willing to pay for a resource based on its quantity. The demand curve is represented by the following expression: P = a - bQ. The constant a and b are proper parameters. If we find the optimal Q* using the demand curve we find the optimal price P* = a - bQ*. To find the optimal Q*, we first compute the revenue function and the cost function. Using these functions we obtain the profit function, we just need to subtract the revenue function from the cost function. After obtaining the profit function, we find the Q* where the profit was highest. Then, we use this Q* value and obtain the optimal price (P*) by intercepting the Q* value (which is a straight line) with the demand

[5]https://www.alibabacloud.com/

curve.

These iterations of steps can be seen in Figure 2.2. One downside it's the price that doesn't take into account the market competition. One difficulty in this approach is to determine the demand curve. To create this curve, different techniques are applied, some use randomness, others follow a distribution function like gaussian.

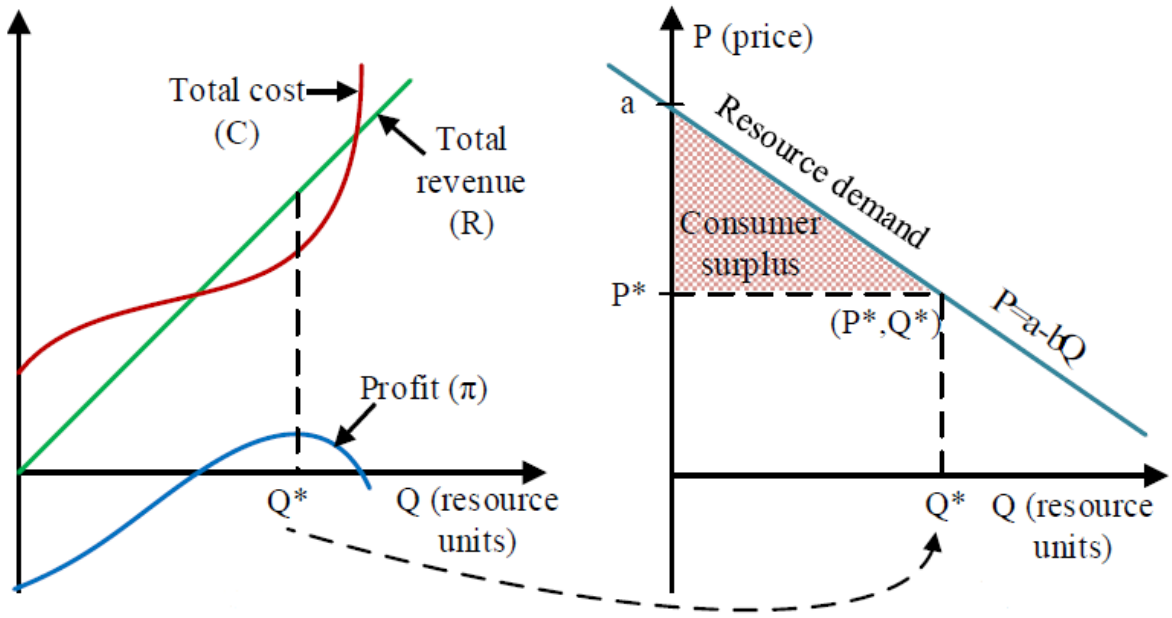

Figure 2.2: Find the optimal price based on profit maximization, extracted from [Ngu17b]

2.2.1 Auctions

Auctions are one very relevant type of Market model. An Auction, in its simplest form, is a process where one item is being bid by a group of buyers and the item is sold to the highest bid [Hui14]. This is one of the processes of the auction but there are others with different actions and we will speak about it bellow. In an auction, we may have up to four types of peoples involved in a transaction. The *bidder* is the person who wants to buy the product and *bids* to obtain it, e.g., end-users, cloud tenants. The *seller* is the person who offered the product to be sold to obtain some profit, e.g., cloud provider. Sometimes the *seller* is the same person as the *auctioneer*. The *auctioneer* is normally an intermediary agent that conducts the auction and ensures a winner is decided. In this section we will speak about the different dimensions an

auction algorithm it can have. These dimensions are **Confidentiality**, **Direction**, **Unity** and **Symmetry**. The taxonomy structure can be seen in Figure 2.3.

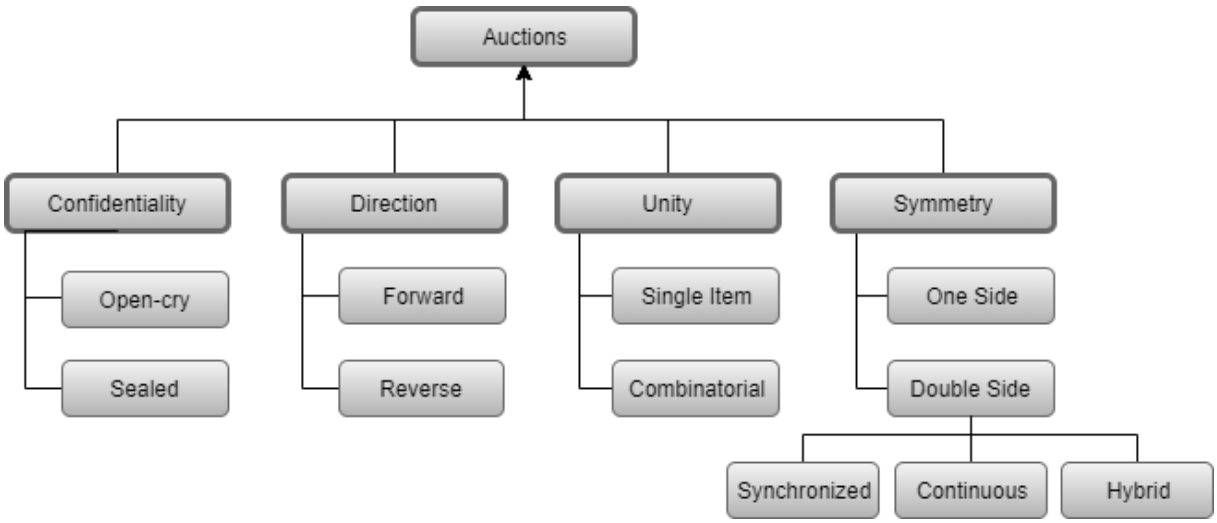

Figure 2.3: Auctions Taxonomy

Confidentiality: this dimension refers to if the price that the bidder's bid, is known by the other participants or not. It is open-cry if the bids are known by all participants. If the bidders don't know the bids of the other participants are called sealed-bid.

Direction: this type of dimension focus on where the competition comes from. It is a forward auction if the bidders fight among themselves for getting the item. To do that the winner must bid the highest value. It is called reverse auction, when we only have one buyer, and the sellers, compete among themselves, by lowering the price of their product, until the buyer choose the item with the lowest price.

Unity: this dimension classifies whether the good being bought is a solo unit, or a combination of items. In a single item the customers only bid for one item, for combinatorial, they can make combinations of goods and buy them as a bundle.

Symmetry: this dimension is related to whether the competition is only among the buyers or between the buyers and the sellers. It is referred to as one-sided if the competition is only between the bidders or the sellers (never both at the same time). It is called double-sided if the competition is between bidders and sellers, meaning, the buyers submit *bids* and the sellers can submit *asks*.

There are many of one-sided auctions. Some of them are: *English* auction, *Dutch* auction, *first-price sealed-bid* auction and *second-price sealed-bid* auction (the second-price sealed-bid auction can also be named Vickrey auction).

In an *English* auction, the auctioneer starts to sell an item from a low price (normally this low

price is protected to ensure some return to the seller) and the buyers bid the item by ascending the price. When no more buyers bid the item, the last bidder or the person who bid a higher value wins the item. It is also open-cry, forward auction and can be a combinatorial or single item.

A *Dutch* auction is also open-cry, can be combinatorial or single item and also follows a forward auction, but the price to buy the item is descending. The auctioneers start selling the item from a high value (normally higher than the real value), and at each time reduces the value until some buyer bids the item to buy. Normally this type of auction is faster than the English auction because there is no possibility of recursing competition of bids between buyers; the first one to accept the price, wins the resource.

The *first-price sealed-bin* differs from the previous ones, in the the confidentiality dimension. The first-price sealed-bid is sealed, meaning, bidders don't know the value of others' bids. Each bidder secretly bids an item. After all the buyers bidding, the seller/auctioneer, orders the bids based on the price bid, and the buyer with the highest bid wins the item, paying the value bidded. There only exists one bid per buyer.

In *second-price sealed-bid*, the process is the same as the first-price sealed-bin, the difference is that the winner of the auction, does not pay the value of his bid, but pays the value of the second highest bid.

In double-side auctions, the buyer bids and the seller asks, creating more competition compared with one-sided auctions, and much more fairness due to both participants, the seller and the buyer, participate actively in the auction, generating a demand and supply profiles. The winner of this type of auction depends on two different aspects: aggregation and resource divisibility. If aggregation is not allowed, for each ask, only one bid can be assigned. Resource divisibility means whether the resource can be divided among multiple buyers. Gode and Sunder [Dha04] further divide double auction into three categories: *synchronized* double auction (or *discrete-time* double auction), *continuous* double auction (*CDA*) and *semi-continuous* double auction (or *hybrid* double auction). In a continuous double auction, a buy order or sell order can be submitted at any time, and if there is a match between the buyer and the seller, the trade can be made at that exact moment. In contrast, in synchronized double auction, traders move at the same time.

In combinatorial auctions. The users may want to bid multiple resources like CPU time, memory, and network bandwidth. The buyers normally, in this case, bid a bundle that contains multiple resources/goods. The advantage of this approach is that, in order to obtain a group of resources, the bidder only needs to attain to one auction, in contrast to participating in multiple

auctions, in each one, obtaining just one type of resources. The disadvantage is the difficulty to obtain the correct price for multiple combinations of resources/goods, then it is hard to find the set of bids that maximizes the revenue generated. This problem is considered a NP-Hard problem [Mur05].

2.2.2 Relevant Systems

In this section we will speak about systems who proposed different auctions algorithms (often based on one-sided or double-sided but with some nuances) or use other type of market mechanisms.

Envy-Free Auction Mechanism

The mechanism proposed by Bahreini et al. [Tay18] handles allocation of resource available at two levels of the system by combining two auctions, position and combinatorial auctions. The position auction disables the use of resources from different levels (levels represent where the resources are located, in edge or cloud). The combinatorial enables to bid a bundle of resources in contrast to bid n times for n different/equal resources.

A bundle is composed of a set of virtual machines. A virtual machine is composed of three types of resources that will be used to allocate: CPU, memory, and storage. A bundle request that is allocated to a user, is never associated to more than one level. They defined a preference factor for a given level (edge or cloud) based on the distance between the users and resources that are associated with that level. They assumed that the users are single-minded, they only have interest in a single bundle and all super-sets of that bundle. Valuing the other bundles as 0. They do not target truthfulness for their mechanism, focusing on more two other properties: individually-rational and produces envy-free allocations. The first one is to guarantee that users participate in the mechanism and the second that no user would be happier with the outcome of the mechanism of another user.

Multi-Round-Sealed Sequential Combinatorial Auction

A multi-round-sealed sequential combinatorial auction mechanism is proposed by Zhang et al. [Hel]. This auction is combinatorial, meaning the good can be a bundle of items, also it is multi-round. In one-round auctions, all the bidders submit their bids at the same time, and the auctioneers choose the winners and match them with the resources. This approach was not used because the architecture proposed by Zhang uses multi-service providers, and one-round

auctions need one controller following a centralized architecture.

The auction mechanism is thus divided into three stages: *bid strategy*, *winner determination*, and *payment rule*. At each round, the users send their resources requirements and bids to all auctioneers (service providers) (bid submission). After that, the service provider chooses the winner based on who brings the highest utility to the service provider. The utility is based on the bidding value provided by the bidders. The bidders who fail to obtain one service provider are moved to the loser vector. The bidders from the loser vector will bid again in the next iteration/round receiving a bid improvement. The iterations finish when the number of max rounds has been achieved or the difference between the utility of the current round with the previous round is lesser than a threshold. In the last stage, payment rule, the bidders who won the auctions pay the value equal to the second-highest bid, following the sealed second-price auction (Vickrey Auction) approach.

Tycoon

Tycoon [Kev] is a distributed market-based resource allocation on an Action Share scheduling algorithm. In their architecture, the authors separated the mechanism from the strategy. The strategy interprets users' and applications' specifications and the resources desired. One example of that is: one web server may have more concern about latency than throughput and is, therefore, willing to consume lesser resources, but that resources should be located near the clients. The mechanism provides incentives for users truthfully value the resources, and the providers provide good resources. Their Auction Share is similar to proportional share, but enables them to specify how they trade-off throughput, latency, and risk.

In **proportional share**, a group of buyers offers some value to buy a group of products that are divisible. Then the products will be divided into the buyers based on their bids. The percentage of products provided to each buyer is proportional to the bid value in comparison to other bid's value. This mechanism maximizes the allocation of the product to a buyer because everything will be assigned to one buyer. In Figure 2.4 we can observe three end-users bidding for a memory RAM of 100GB. After each one bid his own price, the auctioneer divides the RAM proportional based on the bid's value. Proportional share has various variants, another type of use is, instead of dividing resources to multiple buyers, the buyers always obtain the total resources, and what they are buying is the time of CPU usage, each one obtains the CPU proportionally to the bidding price.

One of the advantages of Proportional Share is that it offers a higher time of utilization and lower time of reservation.

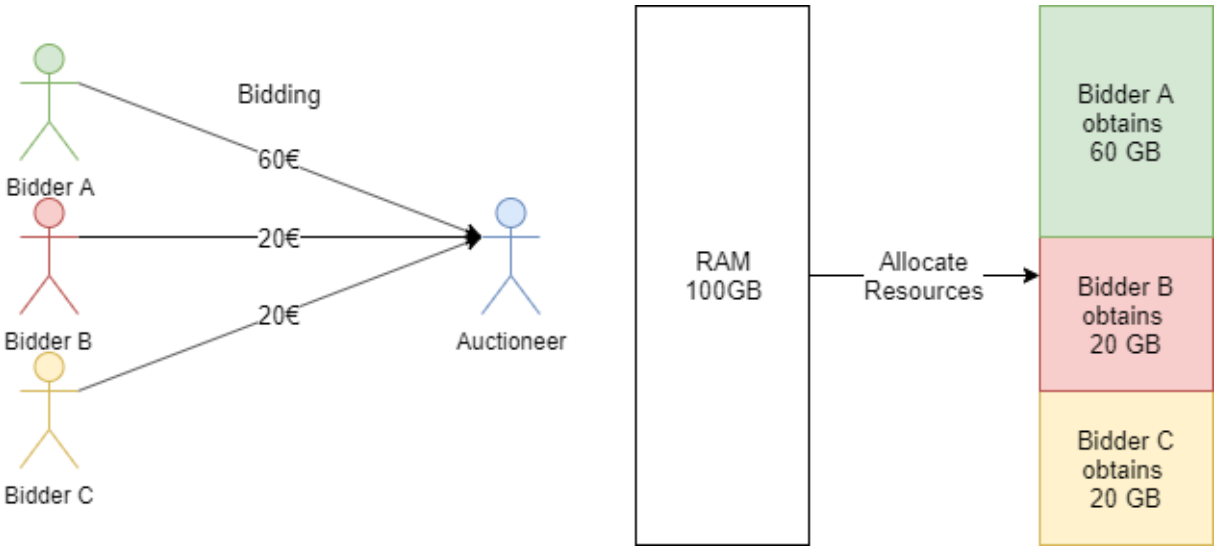

Figure 2.4: Proportional share example

Two drawbacks are that the product must be divisible for a group of bidders to bid, the product will be divided, based on the bidder's value, to all the bidders, and the totality of the resource is not guaranteed. the totality of the resource is not guaranteed because one may want to rent all the products or none, but using the Proportional Share, if another buyer bids too, one will lose some products to that bidder, ruining one goal of getting all or none.

For fine-grained resources, it is used as a first-price sealed-bid auction or the second-price sealed-bid auction. The bidder with the highest bid wins allocation to a window slice time processor (each buyer gets their respective time to run their applications on device's cpu). Because of being distributed, the system is fault-tolerant and allocates resources with low latency.

PeerMart

PeerMart [Dav05] is a distributed technology which enables trading of services using Double Auction algorithms over a peer to peer network. Their goal was to maximize the consumers' utility and find sellers offering a particular service at a low price, and the providers goal is to offer their services at the highest price possible and maximize their profit. The intermediary peers are responsible to match consumers with the providers respecting the consumers' and providers' requirements efficiently.

The authors chose using pricing mechanisms to incentivize peers to provide services, like file storage. By choosing a peer to peer architecture, they don't have a central authority to maintain the prices bid by the bidders or the sellers. One way to communicate could be using broadcasts, but this does not scale and doesn't guarantee that all peers are reached. They proposed to

maintain routing tables at intermediary peers. Then peers use these routing tables to find the peer who offered a given service for a given price. The use of double auctions is derived from single-sided auctions and it has the disadvantage of being consumer- or provider-oriented. One of the problems of implementing a peer to peer infrastructure is that malicious or faulty users may exist. To solve this problem, PeerMart uses PKI (Public Key Infrastructure) to identify the sender.

The auction algorithm works as follow: A provider (consumer) who wants to provide its service (consume a specific service), sends an offer (request) to the respective broker, which is composed of a set of peers. The broker replies with the highest buy price (lowest sale price) offered by another peer. After the provider (consumer) receives the information, it sends a bid to the broker applying its own strategy. Then the broker receives the bid and with that, chooses one of the two options: After receiving the offered price, there is no match if the offered price is higher (lower) than the current bid price (ask price). Therefore, the offered price is either dropped or stored on the table for future use. If there is a match, the price is sent to the peer who has the highest bid. The price paid to the provider by the buyer, is the mean between their offered prices.

To implement a peer-to-peer network, they choose to use FreePastry[6]. It is a tool that is implemented in Java and the the overlay of the network is based on Pastry.

Bellagio

Bellagio [Alv04] is a resource management tool that allocates resources using combinatorial auctions. To discover the resource existing in the network they use SWORD [OAPV04]. The users submit the bids to a centralized auction, and they use XOR language as the bidding language in order to simplify the learning curve of the end-users. The auction is periodic and receives requests for heterogeneous resources like disk space, memory and bandwidth.

Lin et al.

Lin et al. [Wei10] propose a dynamic auction mechanism to allocate resources in a edge computing environment. They made two contributions: i) the introduction of peak/off-peak concepts into the resource allocation, ii) the system contains two types of tasks, background, and float. The first contribution enables the cloud provider to increase efficiency and its revenue in a varying demand environment. The second contribution enables the devices to distribute the resources to end-users and have its own background process. The revenue is obtained from the

[6]https://www.freepastry.org/

inputs to the background task and also the resources shared with the users. They use second-price sealed bid, each user bids to a cloud service provider. The cloud service provider collects the bids and orders them. They find how much capacity they can provide, e.g. k, and from this capacity, they say the price is the (k + 1) highest bid. The k highest bidders obtain the resources with the price from the (k + 1) highest bid. They employ a truth-telling method due to the price to pay being determined by their own bids.

Double Multi-Attribute Auction

Wang et al. [Xin14] proposed a resource allocation model based on the Double Multi-Attribute Auction (DMAA). Their model focus on three important steps. Firstly they transform the non-price attributes in a Quality Index that represents the assesment to the previous transactions. After that, they use Support Vector Machines to predict the price. Lastly, they use Mean-Variance Optimization to obtain an efficient solution to allocate the resources (choose the winners) to different users.

Their system is divided in three actors: the Cloud Resource Provider (CRP), Cloud Resource Consumer (CRC), and Auction Organizer (AO). The CRP provides the resource in exchange of a payment. The CRC pays to a CRP to allocate resources. The AO is the auction organizer responsible to collect the bids and asks, match the transactions, and select the winners.

To calculate the price submited by the CRP/CRC a group of steps are necessary. In CRP, they obtain three non-price attributes, namely, Quality of Service (QoS), Level of Delivery (LoD) and Level of Spiteful Quote (LoSQ). The CRC also follows the same logic but doesn't have the QoS attribute. Then they use these attributes and transform them in a Quality Index by using a neural networks algorithm. The activation function used was the Sigmoid function.

After having the quaility index, they use Support Vector Machines to predic the price. In order to find the estimated transaction price, they also use quality index and other metrics (created by themselves) like reserve price of provider, ration supply demand and expected sale amount of provider. After that, the CRP/CRC obtains the estimated transaction price. To train the Support Vector Machine classifier, they use historical samples from the previous auctions and input information.

Then, the AO makes the match between the CRC and CRP based on this information, and determines the winner by using Mean-Variance Optimization. This algorithm enables to find the most efficient way to distribute the resource to the users.

Auction Based Resource Co-Allocation

Auction Based Resource Co-Allocation (ABRA) [Ali09] is a model that improves upon a previous novel combinatorial auction model called multi-unit nondiscriminatory combinatorial auction (MUNCA). The new model penalizes the non-allocated resources after an auction, having a better resource utilization and hence increasing the revenue. Their model can be mathematically formulated by using integer linear programming. In the paper, it was also proposed a set of five new heuristic algorithms that are based on well-known meta-heuristic techniques. These five heuristics are: (i) simulated annealing, (ii) threshold accepting, (iii) list based threshold accepting, (iv) variable neighborhood search and (v) genetic algorithm.

Reverse Batch Matching Auction

Wang et al. [Xin12] proposed a Reverse Batch Matching Auction (RBMA), that is based on reverse auction, but has additional features like batch matching, to improve the reverse auction efficiency, and twice-punishment mechanism to prevent fraud and malicious users. RBMA has three participants in the system: Cloud Resource Consumer (CRC), AI (Auction Intermediary) and Cloud Resource Provider (CRP). AI the is key component to control the system. It stores the resource information, applies the reverse auction, batch-matching, and twice-punishment mechanism. The CRC and CRP send their tendencies/intent (e.g. bids, resource ammounts) to the AI. Then, the AI starts the auction process that is divided in three stages: waiting period, preparation period and auction period. The waiting period is the period that the buyers and the sellers send their tendencies/intent to the AI, and these are ordered in ascending order, if are seller bids, and in desdending order, if are buyers bids. The start of the auction is marked when the system receives the first buyer's parameters. In the preparation period, it is where the AI selects the buyers and sellers that can participate in the auction. This selection is a time-restriction. Auction period is the last stage. There is a matching between the CRC and CRP based on the bidding prices. At the end of the auction it is used the twice-punishment mechanism. The first punishment comes at the end of the auction round, the program will enforce to the CRP to bid its CP (cost of the rented resource), and enforces CRC to increase its RP (Reserve Price). The second punishment comes if the auction hasn't come to an end after receiving the first punishment. It will set The price of every CRP bid and the reserve price of every CRC to be the unified median price. After that, the resource allocation mechanism uses Immune Evolutionary Algorithm and the transaction price to otimize the resources allocation. Also, the CRP service is graded to improve future services that CRC obtains. To evaluate the

auction, three evaluation criteria were applied by them: market efficiency, user satisfaction and quality service.

Combinatorial Double Auction Resource Allocation

Samini et al. [Par16] proposed a Combinatorial Double Auction Resource Allocation (CDARA). To simulate the prototype of this auction, it was used CloudSim, which is a Java-based simulator for simulate cloud environments in order to extract metrics and evaluate the efficiency of the auction algorithm. In their environment, there are four entities: the user, the broker, the cloud provider, and the cloud market place. The cloud market place is composed of cloud information service (CIS) and auctioneer. From the article, it stems the cloud market place as being a centralized entity. CDARA is divided into seven communication phases.

At first phase, the cloud providers send their resources, and their respective prices, to the CIS. The users send their tasks to the broker and, for each task, the broker gets the list of resources that match the requirements to run that task.

The second phase, the broker generates bundles (a combination of resources) and the price for each bundle. The cloud provider does the same action. Both send price (bids) to the auctioneer.

In the third phase the auctioneer communicates to the broker and the cloud providers the end of the auction.

In the fourth phase, the winner is determined. In this phase, the users and cloud providers are ordered depending on what resources they are bidding/sharing and the respective price.

In the fifth phase (called resource allocation), the auctioneer checks if the cloud provider has the necessary requirements, requirements defined by the user, to run the tasks. If the first cloud provider cannot fulfill the requirements, the auctioneer passes to the second cloud provider. After the requeriments of the first user are satisfied, the auctioneer applies the same procedure for the next user.

In the sixth phase it is selected which pricing model to use to decide the payable price by a user to a cloud provider for allocating resources. To use this model, it is used the number of requested items by the user and the number of offered items by the cloud provider.

In the last phase, the user sends the task to run in the cloud provider's resources. And the user makes the payment to the cloud provider.

2.2.3 Analysis and Discussions

After studying and classifying taxonomically the edge cloud environment, in our solution, it will be followed a distributed nature, more precisely, we will aim to implement a peer-to-peer architecture. The reason of this choice is because, peer-to-peer is much more scalable and our solution will target all end-devices that users use, which could be millions of devices. Of course, this choice also brings disadvantages, and one of them is the existence of malice users. This is more dangerous in peer-to-peer because we don't know the reputation of the other peers, which could affect the application's behaviour as a whole.

On top of our overlay network, it will be used an auction mechanism to match the resources being sold to a buyer. The auctions follow more a price/demand curve comparing with the other market mechanisms because multiple sellers or providers influence the price

It was hard to choose which type of auction we will be based on to implement. This difficulty was due to the different characteristics they have between them, and these differences aimed to solve different problems. We decided to adapt auction mechanisms, that would fit better in our domain (sharing resources) and our architecture (peer-to-peer).

Chapter 3

Solution

In this chapter, we will present present the architecture of our solution which will be called RATEE (Resource Auction Trader at Edge Environment). *RATEE* is a decentralized trader that matches users that want to deploy docker containers and users that are selling its resources to deploy a docker container in trade-of money. The mechanism used to make this transaction is based on auctions. The solution target primarily edge cloud environments/end-users. After studying the existent technologies (previously talked in Section 2.2.2), we will first discuss the desired requirements that our solution should have and the conditions a user must comply to run applications in the proposed solution. The algorithms and data structures of the solution are then discussed. At the same time explaining the components of the system and their interactions. Finally, we will classify our system based on the taxonomy presented in Chapter 2.

3.1 Requirements

Our target is the edge environment, meaning we want to support thousands of users. Volunteer Networking and Computing platforms, like the GUIFI.net[1][BRFN15] community network, is growing constantly. So, our solution must be able to scale with the demand of the users to deploy containers or sell resources. This is important because, since it is a solution that depends on supply and demand trade-offs, the more users we have using the better is the solution, also, if we are sharing resources machines to deploy containers, we don't want our solution to consume a large part of those resources.

We are enabling the possibility of deploying a container in a machine, but the buyer has its own requirements to deploy the application. It must be possible to define how much our

[1] guifi.net

application requires resources to be deployed in order to be flexible and support variable groups of services. With these different requests, it is also possible to have different prices, and depending on the request we can target all groups of users. So we need to define when we are making a request the amount of CPUs and RAM that we need. Depending on that, different prices will appear. An offer is composed by a price and a resource. A resource is defined by the amount of memory RAM and CPU.

From the execution platform point of view, and to target all users, we need to have interoperability and the possibility to run workloads in all environments, independently of the operative system, and hardware architecture. This applies also to the application that has to be deployed to containers in order to run in all systems.

3.2 Prerequisites

The user to use our application must fill in predefined conditions such as:

- If we are sharing our resource to other people deploy their applications, we must have docker installed locally. If we are only buying this is not needed.
- Run our application (RATEE) in the background in order to facilitate the deployment of other containers and help other users to make the fastest trades.
- Have internet access or at least have a private connection between the peers that are going to be part of the network, because the system uses peer-to-peer communication mechanisms.

Other mechanisms may be needed in case this solution is deployed live to production, but due to limited time and be out of scope they weren't implemented and will be added to future work):

- Reputation system that gives points to the uses when a trade happens and both parts do the agreement. And the users with higher reputation will only trade with other users with also high reputation. This will mitigate and remove malicious users.
- A high distributed file system in order to persist the container's image and facilitate the download in a decentralized way. This case is for edge application and if we want to support more than one container technology.

- A client to make transactions using virtual currency. This would decouple the application for a payment provider and obfuscate the users that are participating in the transaction, protecting their identities and information.

3.3 Operations Supported

By using our tool, the users can share their resources, in trade of money, or buying the resources of other users (that are sharing). The two main operations supported are:

- Creation of a bid. This is translated by searching in the network for producers that are selling that resource. After founding it, the user pays and the application is deployed.

- Creation of an ask. This is the reverse process, we sell our resource for a given price. After finding a buyer, we deploy its application.

We also support other auxiliary operations such as:

- Get all bids/asks created that are in pending state (no matching offer was found).

- Remove bids/asks.

- Configure banking account number information.

3.4 Distributed Architecture

In order to have an application that scales and supports thousands of users, having a decentralized architecture is better because we are diving the computation between the parties. These parties are consumers or providers that will support with storage of information. If we need more resources, we just need to add more parties. Our tool work is based in a peer-to-peer architecture, meaning all nodes have the same code and responsibility. There are a lot of different approaches to implementing a peer-to-peer architecture. Each approach has its own network structure of way of sending messages and responsibility, instead of a random approach of broadcasting to all peers. To add a peer-to-peer network structure we will use a third-party library Libp2p[2]. This library creates a p2p network where its structure is based on Kademlia DHT.

[2]https://libp2p.io/

3.4.1 Kademlia DHT

Kademlia[ME02] is a peer-to-peer distributed hash table. Like many other peer-to-peer distributed hash table implementations, this one also has Keys of 160-bit, each node that participates on the network is also identified by an ID of 160-bit and values are stored on nodes with IDs close to the key. One of the benefits of Kademlia is the use of a novel XOR metric for the distance between points in the keyspace, XOR is symmetric allowing the participants of the network to receive the same number of queries for lookup. In contrast with other types of peer-to-peer which have an asymmetric structure which leads to rigid routing tables. Kademlia consists of four RPCs (Remote Procedure Calls): *ping*, *store*, *find_node* and *find_value*. The first one is used to check if a node is alive. The *store* is used to save information ¡name, value¿. Then we have *find_node* to obtain the address of a node, and *find_value* that returns the value that was stored with *store* RPC. Its routing table is a binary tree where each leaf is composed of k-buckets. These k-buckets contain nodes that start with the same prefix for the ID. Like any other decentralized peer-to-peer system, some security issues have been identifier. This caused the creation of a secure key-based routing based on Kademlia, S/Kademlia [BM08]. They implemented parallel lookups, limiting the free node identifications generation with a group of crypto puzzles. To store and replicate the data in a safe way it is used a reliable sibling broadcast.

3.5 Algorithm

The system uses an auction mechanism with bidders and askers. Each user can be a bidder and an asker at the same time, or just be one of the two. Also, our solution is peer-to-peer so we need to send messages to other peers in order to trade information. And due to the complexity of the transaction, a state machine will be useful to help in these cases.

When we start our application, the first thing *RATEE* does is connect to the peer-to-peer network. A connection is established with the boot peer. This boot peer is an application that already is running and is the entry point for the nodes to connect between themselves. Due to being a boot peer, all other peers know its IP in order to establish the connection. With a small number of boot peers, this doesn't scale at production environment, being easier with a higher number of that peers. Another solution that could be used is each node has a discover algorithm to find each other, not being dependent on an entry point. The last one scales better, because we would reduce the number of messages sent to the boot peers (to join the network) which is a point to create a bottleneck.

Libp2p supports both approaches but the second one has a lot of problems in public networks

```
{
    "id": Guid
    "price": number
    "resource": {
        numbersOfCpu: number
        memorySpace: number
    }
    "isBid": boolean
    "offerResourceHash": string
    "isReserved" : boolean
    "lastReserved" : Date
    "peerReserved" : {
        "id" : string
    }
}
```

Listing 1: Offer structure

due to firewalls and NATs. Because of that, we used the first approach. Then, in case we have multiple users using our application, we may use the first with the second approach in order to have the advantages of both worlds.

We first start the offer received by the user, in this case, a bid. A bid is similar to an ask, so the same structure is used. We just need a boolean to differentiate between themselves, and a bid also receives as information the docker image application that will be deployed after the transaction is made. We can see in Listing 1 the structure of an offer. The bid is composed by an id which is a Guid and is used to identify the offer, different offers should have different identifiers. Then the price represents the amount that we are willing to pay/sell in our offer. We have a boolean *isBid* that represents if that offer is a bid or an ask. The offer contains an object that saves the resources that we are selling or buying. The *offerResourceHash* is the mapping of our resource plus if is a bid or not. Lastly we have the fields that represent if an offer is reserved, the last time that was reserved and if it is reserved, the peer that was made the match. We can observe in Listing 1 the offer's structure.

After receiving an ask, the system searches if there are peers that are buying that resource. For that we map that resource to an Identifier, and using the libp2p we search for buyers. Due to being the first user in the network, no bidders are found, so we save our ask and notify that we are selling that resource (using a libp2p method). In this step, the Identifier/key created previously is converted to a DHT Id. Using that DTH id tries to find the closest peers to that id. Those peers will save the association between the DHT id and the provider of that id. To

```
{
    "requestResourceId": string
    "isBid": boolean
}
```

Listing 2: Get Price request message

find providers the process is identical, but instead of sending a command to save the key, we send a command to obtain the providers that have that key.

Suppose a buyer wants to buy that resource. In case the user wants to buy another type of resource, the process will be the same, what happened with the seller will happen with the buyer, no sellers will be found for that resource, so the bid will be saved and notified to other peers. In this case, he wants to buy the same, so the application will receive a request for that bid, then it will try to find if sellers are selling that resource. In this case, it receives the IP of the other seller, (if other sellers were offering the same resource, we would also receive that information). After getting the address, it sends a get price request. We can observe in Listing 2 that the message is composed by a boolean to say if is a bid or not, and then a Guid that represents that resource that we want to bid.

After sending the message the seller will receive it and check what message type is. It will deserialize the message and based on the type redirect to the correct handler. In this case, the request is to obtain the price, the seller will prepare the response message with the asks with the resources that the buyer wants and send it to the buyer. We can observe in Listing 3 the response message.

Algorithm 1 Handler of a get price request command

1: **function** GetPriceHandler($resourceId$, $isBid$)
2: $allOffers \leftarrow isBid?askTable : bidTable$
3: $requiredOffers \leftarrow allOffers.filter(a => a.resourceId == resourceId)$
4: $response \leftarrow createReponseMessage(requiredOffers)$
5: **return** $response$
6: **end function**

The response contains the information about the owner, and the offers that were associated with that resource id, its price and id to be identified in future uses. In case there are multiple sellers for the same resource, the buyer will send Get Price Request to all of them, and after that aggregate them in an array and sort them based on price. The resource that are being sold with the lowest price will have an higher priority to make a transaction.

After receiving and aggregate all the prices, it will start to send to the cheapest one a bid

```
{
    "offersList": [{
        "id": string
        "price": number
    }]
    "offersOwner": {
        "id": string
    }
}
```

Listing 3: Get Price response message

```
{
   "owner" : {
        "id" : string
   }
    "bidOffer" : {
        "id" : string
    }
    "askOffer" : string
    "resourceRequestId" : string
}
```

Listing 4: Send bid request message

request. In Algorithm 1 we can observe the bid request handler. The format of the bid request, presented in Listing 4, includes we send our identification, our bid offer and the ask offer that we are trying to match. Then the seller will receive this bid request, and check if its offer is not reserved to any other user. If it isn't, it sends a response saying if the bid was accepted or not. If it was accepted, the seller adds information to the offer that he is selling, the buyer identification and the time when it received the message.

This timestamp is the amount of time that the ask is reserved for that bid. In case the buyer does not want anymore this offer, due to a big number of reasons (e.g. he has found another seller that is selling the same or for a cheaper price), another more drastic case, he has network problems or is instance/machine goes down. This way the seller or buyer doesn't depend on the other to trade with other people. This way, if the defined time was exceeded, suppose that the user is not interested, or another problem has happened. But we don't have any kind of obligation with him, because the transaction it didn't happened at that point, so we will search for another user that matches with his offer. The time we specified was 4 seconds. We think this

Algorithm 2 Handler for bid request command

1: **function** BIDREQUESTHANDLER(bid)
2: $ask \leftarrow askTable.single(a -> a.id == bid.askId)$
3: **if** $ask != null$ & $ask.lastReserved.getTime() + 4000 < Date.now$ **then**
ASKTABLE.REMOVE(ask)
$ask.lastReserved \leftarrow Date.now$
$ask.ownerId \leftarrow bid.ownerId$
$response \leftarrow CreateSuccessResponseMessage(ask)$
return $response$
4: **end if**
5: **return** $CreateErrorResponseMessage()$
6: **end function**

```
{
    "bidOffer": {
        "id" : string
    }
    "offerSold": {
        "id" : string
    }
    "bidAccepted": boolean
}
```

Listing 5: Send bid response message

```
{
    "owner" : {
        "id" : string
    }
    "askOfferId" : string
    "dockerImage" : string
}
```

Listing 6: Send transaction request message

```
{
    "transactionAccepted" : boolean
    "message" : string
    "iban" : string
}
```

Listing 7: Send transaction response message

is enough time to send a response (even with network problems) in case the other user wants that offer. The identification, which is saved in *peerReserved* field in offer structure, helps to know the buyer, in case this user receive another bid from another user, this bid is discarded because had already reserved for another user. And in the future, the bidder instance will receive the transaction request from that user and based on the id we accept the transaction.

The buyer receives that the bid is accepted and send a transaction request. The schema of the request is presented in Listing 6, where it sends the ask that was reserved and also the information about the container image, that will be running in the seller machine. After receiving this message the seller will check if, the offer is reserved for that buyer, if it is, it sends a response of success, and deploys the container.

The response contains the IBAN that will be used to pay to the seller, and a boolean saying if the transaction was accepted or not. The message can be seen in Listing 7. In case the buyer can not make a transaction with the first seller, it will send the request to a second one, and so on, iterating the array with all the prices that was obtained in the get price request, that was sent to all providers of given resource. If is refused by all sellers, the behaviour is the same as if no seller was found. Algorithm 3 summarizes the bid creating process.

Algorithm 3 Node behavior after a user creates a bid

```
1: function CreateBid(bidOffer)
2:     bidsTable ← bidOffer
3:     resourceId ← ResourceMapper(bidOffer)
4:     providers ← libp2p.getProviders(resourceId)
5:     for each provider ∈ providers do
6:         prices ← libp2p.getPrices(resourceId)
7:     end for
8:     for each price ∈ prices do
9:         response ← libp2p.sendBid(bidOffer)
10:        if response == ACCEPTED then
11:            libp2p.StartTransaction(response, bidOffer)
12:        end if
13:    end for
14: end function
```

3.6 Taxonomy Classification

Based on the Edge Cloud taxonomy defined in Section 2.1, RATEE has the following classification:

3.6.1 Edge Cloud

This is RATEE's classification about Edge Cloud taxonomy defined in Section 2.1 of related work:

- **Resource Ownership:** About resource ownership we classify it as Volunteer. We are trading money for resources, and in Single Owner it doesn't make sense this trading, because we own the resources. It is Volunteer because users share their resources with other users in trade of money.
- **Architecture:** Our application is decentralized, more precisely peer-to-peer, meaning the information is not stored in a single point. Each user runs the same code base.
- **Service Level:** We support IaaS by deploying containers applications. This type of applications are much smaller than virtual machines.
- **Target Application:** We can target many types of applications, so we focused primarily in General Purpose. It can also be used for data intensive workloads, if that application can be deployed through a container.
- **Access Technology:** Our application uses TCP to communicate with other peers. Due to this restriction, it is necessary to use Wi-Fi, Ethernet, our another technology that

supports TCP.

3.6.2 Auction

This is RATEE's classification about Auction taxonomy defined in Section 2.2.1:

- **Confidentiality:** About confidentiality we are Open-cry. Each user knows the bids of others, which leads to a much more dispute in the environment.
- **Direction:** The direction is Forward. When we want to match with other bid or ask, we will increment it, to have a higher bid/ask.
- **Unity:** For now a user can only bid to a unit/resource. If it wants to get another resource, it needs to send another bid (which in our system is a new offer). So we classify it as Single item, each request must contain the memory and the CPU needed.
- **Symmetry:** About symmetry we are Double side. A buyer can send bids to a given item, and the seller can send asks to a given item that is selling.

3.7 Summary

We started talking about the requirements that we had to built our tool. It needs to scale, due to targeting a great number of users, and supports different environments. Then we specified the pre-requesites that need to be full-filled in order to run our application. If those requirements aren't meet, all features may not be full filled due to dependencies. Also we described the operations that we support in our tool. After that we talked about our distributed architecture, that we use libp2p, which is based on the Kademlia DHT. Then we discussed about the RATEE's behaviour, its algorithms, the message contracts that they need to respect, and some scenarios that may happen. At the end we made a taxonomy classification based on the domains that were identified in Chapter 2.

Chapter 4

Implementation

The algorithms presented in the previous chapter where used to develop a prototype of the RATEE system. This application was written in Typescript. There were a lot of programming languages to choose from, predominantly Java, Go, JavaScript, Python, etc. One difficulty to use Java was the scarce number of frameworks to create a peer-to-peer application. The ones that exist were out-dated in terms of technology and lacked documentation. The other languages had some frameworks that were recent, but right now, JavaScript is the most used and popular language, having many more tools and features lately. Another advantage, comparing with other programming languages, it can be used to run back-end programs or servers, using Node.js, or to front-end running code at browsers like Microsoft Edge and Google Chrome, which is easier to migrate an application from back-end to front-end and vice versa. But we didn't use JavaScript purely, our main program is written in Typescript, which is another language that extends JavaScript but adds typification. A typified programming language brings much more stability to the code, reducing the number of errors at run-time. It was not used TypeScript completely because some dependencies were written in JavaScript. In the following sections, it will be described the architecture of RATEE, its modules, and its major responsibilities.

4.1 RATEE

4.1.1 Software Architecture

RATEE has the functionality to do trades of system-level resources, such as CPU and memory, based on an auction mechanism. This tool is peer-to-peer, so each instance will have the same code running (in contrast with client-server architectures that have different code for client and server). This code is divided into different modules (or components), each one with is own

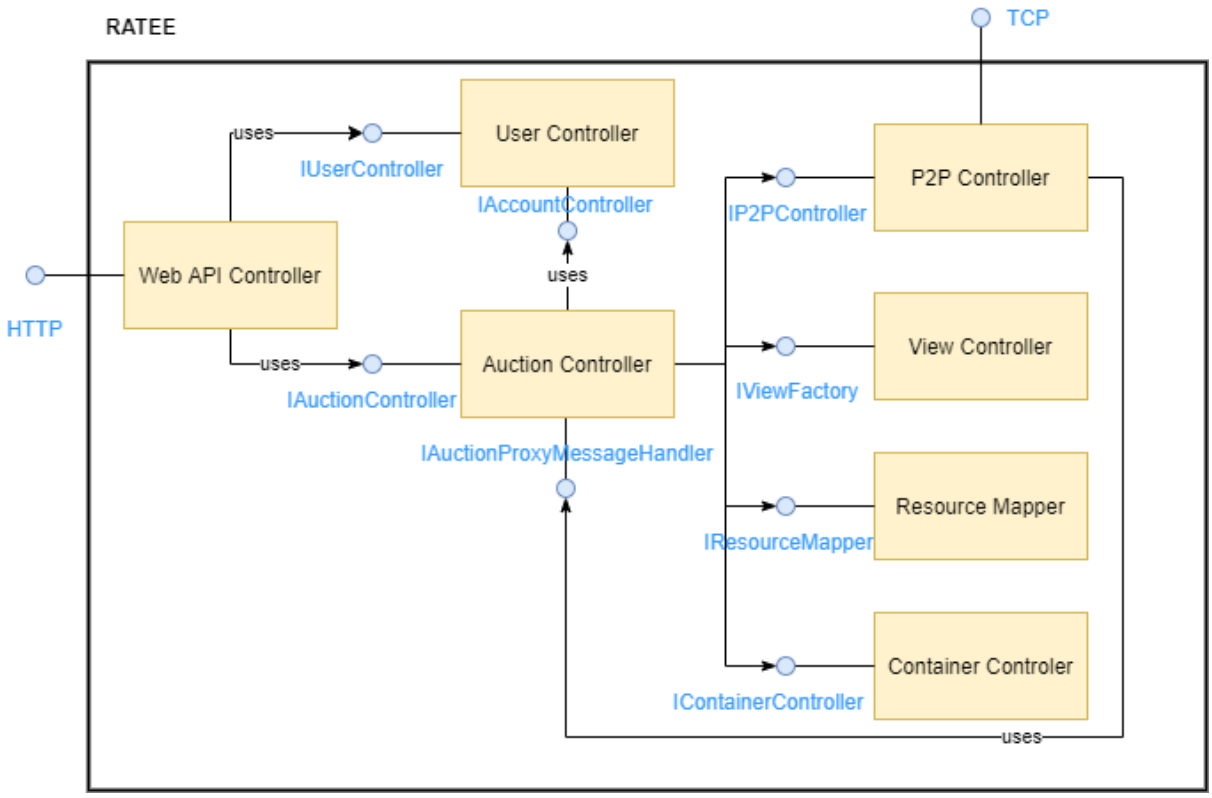

Figure 4.1: RATEE components and interfaces relationships

responsibility, This modularity helps to have code more maintainable, modifiable, and testable. When writing code, SOLID principles[Mar00] were also taken into account, they also bring positive properties for the code. So, RATEE is composed of seven modules: **Auction Controller**, **Resource Mapper**, **Container Controller**, **User Controller**, **P2P Controller**, **View Controller**, and **WebApi Controller**. In Figure 4.1 change we can observe each module and interactions between themselves. This visual help will simplify the understanding of the following sections where we will talk about each module.

Auction Controller

This is the most important module and the center of our tool. It is responsible to save the bids/asks of the user and run the logic to make the trades between other users, respecting the auction mechanism. Implements interface `IAuctionController` and other modules that depend on this one, use that interface to call it. It depends on Container Controller on interface `IContainerController` and uses to deploy a container after the user has sold his resources successfully. After receiving a bid request, it needs to map it to a Hash. Those mappings are made by Resource Mapper, through Interface `IResourceMapper`. And also uses the View Controller to create JSON objects to be returned to the user. This controller im-

plements two interfaces `IAuctionController` and `IAuctionProxyMessageHandler`. The first one (`IAuctionController`) is consumed by the Web API module, to apply the commands (create a bid, create an ask, and others) it calls the Auction Controller. The last interface (`IAuctionProxyMessageHandler`) is used by P2P Controller in order to call after receiving commands. These commands are part of the Auction domain, so it was decided that the logic should be contained in the auction module. It also depends on the P2P Controller to send the commands to the other peers.

Resource Mapper

This module is responsible to map a given Resource to an Identifier. Based on a concrete class we could have different mappings, one could take that object that applies a hash after serializing the object and that hash could be the identifier. Others could take the Resource object and create a Guid. The `IResourceMapper` interface is the contract that different implementations must follow. Right now, we are getting the resource object and a boolean to represent if its a bid or an ask. Based on this, the number '0' or '1' is concatenated with the hash of the resource object, previously stringified. The result will be used to find other peers with the same intention.

Container Controller

It is responsible to manage the containers that are deployed. These containers are from the user that are buying resources and, after being matched to a seller, this controller deploys the container in the machine. This class implements the `IContainerController` interface. Right now we only support deploy docker, but other types of technologies can be used easily, RATEE already has an interface to abstract the implementation. It is only necessary to have the concrete class with the logic to deploy the types. Container Controller depends on a third-party library (node-docker-api[1]) in order to deploy the docker containers.

User Controller

This module is responsible to save and manage user information. One functionality it could be about the user credentials and authenticate the user before starting the application. This is exposed by interface `IUserController`. It also saves other pieces of information like IBAN that is used to make the transaction between the buyer and the seller. This is hidden by the interface `IAccountController` and is consumed by the Auction module to send to the seller.

[1] https://www.npmjs.com/package/node-docker-api

P2P Controller

This module is responsible to communicate with other peers. In order to communicate we are using a JavaScript library libp2p[2] that creates a P2P network stack. The API supplied by libp2p, abstracts the programmer from all TPCs semantics, displaying P2P semantics in their methods. This controller also depends on`IAuctionProxyMessageHandler` which is responsible to know what to do after receiving a message. Those messages are strings that are parsed to JSON objects and treated as commands each one with his own information. It exposes `IP2PController` interface.

View Controller

This module is responsible to create the views that will be returned to the user after being made a request. This representation is a JSON object that will be serialized before sending it. The principal class is ViewFactory that has a method that receives the necessary parameters to fill the JSON object and then returns that object already serialized. For different views/JSON objects, we have different methods to be called. The AuctionController uses this view to return to the WebApi, so it depends on its interface `IViewFactory`. In the future, if we want to change from JSON to HTML we just need to create a new class that does that functionality, because we already have the abstraction.

WebApi Controller

It is the entry point of our program and it is responsible to handle the HTTP requests made by the client. The main functionality of our application is to receive requests to allocate resources, so We need to know, the resources and the price that the users are willing to pay/sell. This information could be read from a configuration file. But it is not dynamic, we need all the time to change it, and the application needed to read it and check if no error was present. This flow was not fluid. So we have an HTTP server with routes, where the routes are the commands send by the user like creating a bid, set IBAN, and others. Then the WebApi Controller redirects this information to the respective module. In case of creating a bid it calls Auction Controller through interface `IAuctionController`, and for the IBAN command it calls User Controller through `IUserController`.

[2]https://libp2p.io/

4.2 Swagger

Has it was said previously to communicate or issue commands to our tool, it needs to be through an HTTP request. So, we need an application that works as an HTTP client. One way could be to create a small application that receives commands through the Command Line Interface, and for each one maps it to HTTP requests commands. But there is also another possibility that has much more reduced time to implement. Is by using Swagger[3] technology. This library helps with creating an HTML page with all routes, their behavior, the necessary information to make the request (like parameters and body), and the possible returns. This information is added by two options. It is possible to annotate the code with the metadata of the API, or or we create a swagger.json file with all the information to be represented in a page. The Swagger makes the request for you to the API and returns the response to the page. This library has interoperability, the only problem is we can't have any logic in the client (because we don't have a client). Either way, the CLI tool is still supported and can be easily implemented in case of necessity.

4.3 Tests

When we are implementing it is normal to create bugs and has the code grows, the harder is to maintain it without any assurance that everything is correct without an automated process to do that check. That process is testing. After we code, by adding new functionality, we add some tests to test that code and check if is doing what was supposed to do. In that way, in the future, if we change that part of the code and add a bug, the test will fail and we know what we need to fix. If we apply this rule to all parts of the code after made some alteration we have the assurance that everything is alright with a small delay.

There are a lot of categorizations of tests: the unit tests, the integration tests, and the acceptance tests. In our project, we used unit tests and acceptance tests.

4.3.1 Unit Testing

Unit tests are tests made against the smallest unit of code, in this case, the methods of a class. In order to test the methods of a class, all its dependencies are mocked, objects that we control their behavior, and call that method and check if the dependencies were called and the method did the desired behaviour. With this type of test, we have much more control and

[3]https://swagger.io/

better verification if all the changes made were correct. As downside, because all dependencies are mocked, we don't test the behavior with integrations [Rog04].

4.3.2 Acceptance Testing

In contrast with unit tests, in acceptance tests, we don't mock anything and we test the tool as a single black box. One example is we run our tool in two different instances, and create a scenario wherein instance A we make a bid request and in instance B we make an ask request, and we check if the transaction happens. With the acceptance tests, we test scenarios, user actions, and observe the behavior that the user is observing is the desired one. The tool that was used to make the HTTP requests was POSTMAN[4], and it is great to make HTTP requests and after that write code to test the response. With this, we tested the integration between peers.

4.4 Simulation

After implementing our tool we need to have a way to measure non-functional aspects like, the number of requests processed per second, memory consumed, the number of messages sent, and other metrics that can be used to now if our tool is robust and it can scale. To do these types of tests in an easy way, it was necessary to simulate an environment to stress our tool. After some research, I could not find any tool that could simulate a peer-to-peer environment. There are some technologies that create that environment (PeerSim) but aren't for JavaScript and the technology is old. So to simulate that, we will create some scripts PowerShell to make a lot of requests and also use postman to create scenarios to make requests to our application.

4.4.1 Summary

In this section, we explained the RATEE's components, their responsibility, and interactions between themselves. Also, it was showed the interfaces used to abstract each component in order to facilitate other types of concrete implementations to be added in the future. It was described our client tool to interact with our tool, and also it can be expanded to other tools (they just need to support HTTP communication). In the end, tests were made, giving more relevance to unit tests and acceptance tests.

[4]https://www.postman.com/

Chapter 5

Results

In this chapter, we will evaluate our tool (RATEE). For that, we defined a group of criteria/methods of evaluation which will help to know what are the strengths and weaknesses. First, we will explain what will be the methods used to evaluate, applying those methods what are the results, and based on that results, discuss them.

5.1 Evaluation Methodology

This section discusses the methods that will be applied in order to evaluate our project.

5.1.1 Allocation Success Rate

As it was stated before, our application is peer-to-peer, which brings some difficulties. One of these is to find auctions that are selling the respective resource that we want. In contrast, with a centralized approach, all the information of bids and asks are saved in a server, and the users only need to interact with that server and now for sure, if the server did not return a match, it is because it did not exist. But in a peer-to-peer environment, we do not have all the information centralized in an endpoint, each user has partial information, and to know all the information of the system, we must get it from all nodes, which is not scalable. We assume a normal environment with some users already with some bids and asks, when we had new offers, and we now those offers have a match in the system. We want to evaluate if these offers match. This will measure the effectiveness of our system.

5.1.2 Overhead Memory Consumption

It is important to know how much memory an application consumes, mainly if that application will run in user's devices which are known for having different capabilities. In summary, if the system has hard memory requirements, which are difficult to comply with, we will only target a small market's piece. Another requirement is due to the nature of our application being peer-to-peer, as more users we have using our service, more transactions will happen, even if they only used as mediators. Therefore, we will calculate how much memory our application consumes and check as we add more nodes to the peer to peer network, how the memory of the previous nodes increase.

5.1.3 Ideal Price Deviation

We already calculate the match success using the Allocation Success Rate. But we can be more precise, and after finding a match check how much more the buyers had paid to get a resource that was being sold at a lower price somewhere else in the network. One example is we have two users, selling the same amount of memory (1GB) but one for 10€ another for 5€. Then a buyer arrives and offers 10€ for 1GB of memory. In an ideal environment, the transaction should happen with the user that is selling for 5€, but because of the absence of centralized control in a peer-to-peer architecture, he could buy from the user that is selling for 10€. This raises a problem where buyers may not buy the cheapest resource because of the distributed environment.

5.1.4 Scalability

As it was previously said, nodes communicate between themselves using messages. These messages grow as the number of offers grows. This could bring a bottleneck to our application, if for each offer created, the number of messages sent to other nodes grows exponentially, we will burden the network, and the application will no scale. Based on this point, is necessary to know if how much the number of messages grows, and if it is a limitation when we have thousand of users, each one creating offers.

5.2 Experimental Evaluation

This computer had all the technologies used to run the application, and also it was installed PowerShell to automate the evaluation. The computer has 8 GB of memory and a dual core i7

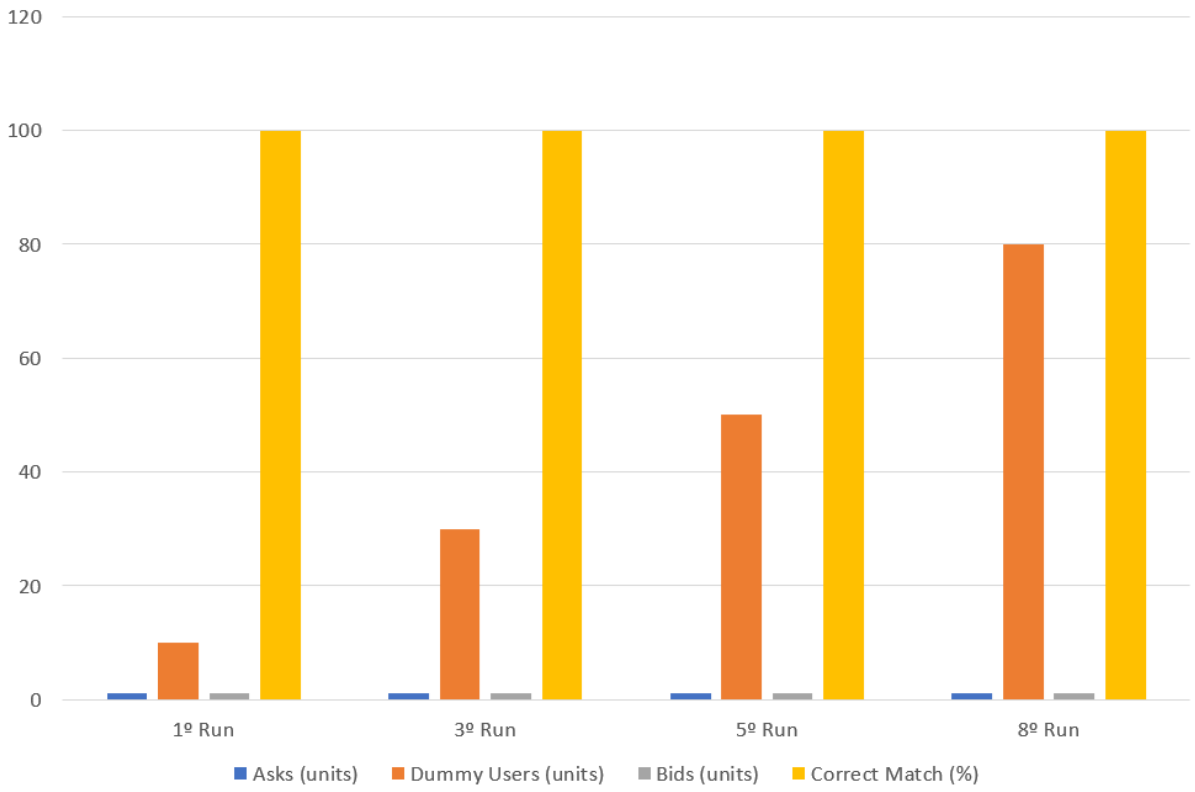

Figure 5.1: Results of a bid finding an ask for the same resources

as processor. Some limitations in the number of instances created for the results were due to limitations of the machine used to test. The following section presents the results obtained for each of those aspects mentioned before.

5.2.1 Allocation Success Rate

In order to test the Allocation Success Rate, dummy users were created which would have offers that would not be matched with any other offer. Then, create an ask, and next a bid that will be matched with the ask created. This is the worst case because of all the offers that exist in the system, only one can be matched with ours. We have done this experiment something progressively doing it 8 runs of it. At each run, we will only have one match, one ask will be associated with one bid, and at each run we will add dummy users with dummy offers. The first run will start with 10 dummy users, each one will have three asks offers, and then we will add our ask offer that will be matched, and add the bid offer. In the second run, we will increment the dummy users from 10 to 20, until the max of dummy users that will 80. We will add 10 dummy users each run. In Figure 5.1, we can observe the results of those runs (we didn't show all the results of the eight runs, due to results being equal).

From the results we can observe that even by incrementing the number of dummy users with

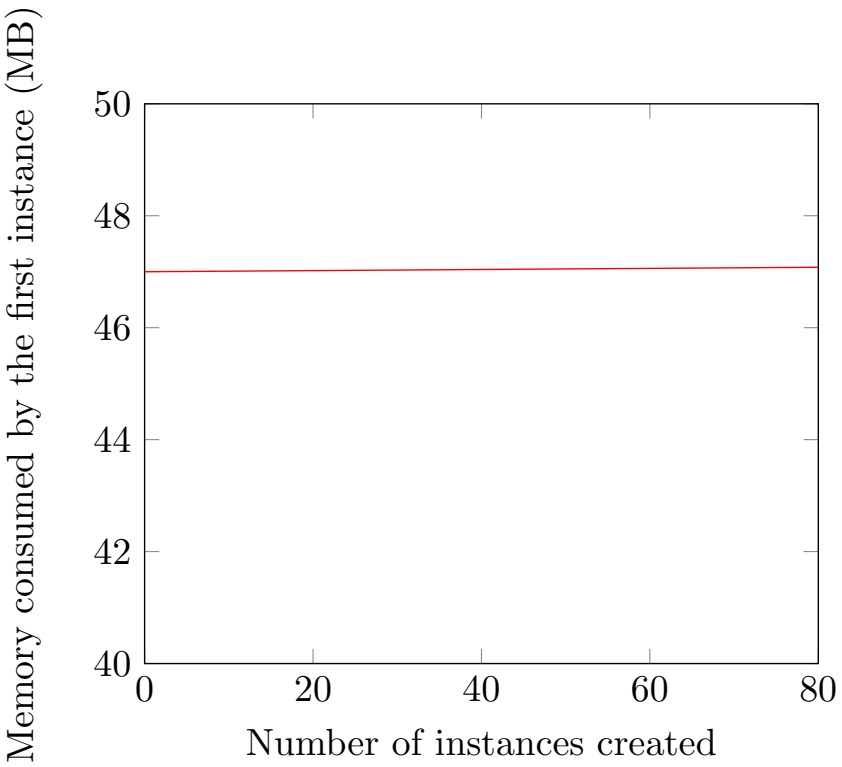

Figure 5.2: Memory consumption of the first instance created

dummy offers, the user that created the bid, always found the respective ask. Resuming, for the 8 runs, the hit success rate was 100%.

5.2.2 Overhead Memory Consumption

To test the memory consumption, we created a PowerShell script, that will create instances of our tool. The program started, the process id will be printed, and at every 100 instances created, using this process id we will get the memory consumption, also using PowerShell commands. We will calculate the memory used by the first instance that was created, which should be the one that consumes more memory due to the time that was running.

We can observe in Figure 5.2 as we had more nodes, the memory used doesn't grow. In this experiment, we didn't create any offer.

In the next one, we created some offers (without being matched in order to be persisted in memory). We can see the results in Figure 5.3. Those results simulate the behaviour observed, and not the real memory consumed for a given of number of instances created.

The memory of the first program also didn't grow. But we can see in the graph it seems similar to a sinusoidal function. First of all, the memory consumption of the first node doesn't grow as we had more offers requests because we created those offers in the other nodes. When we

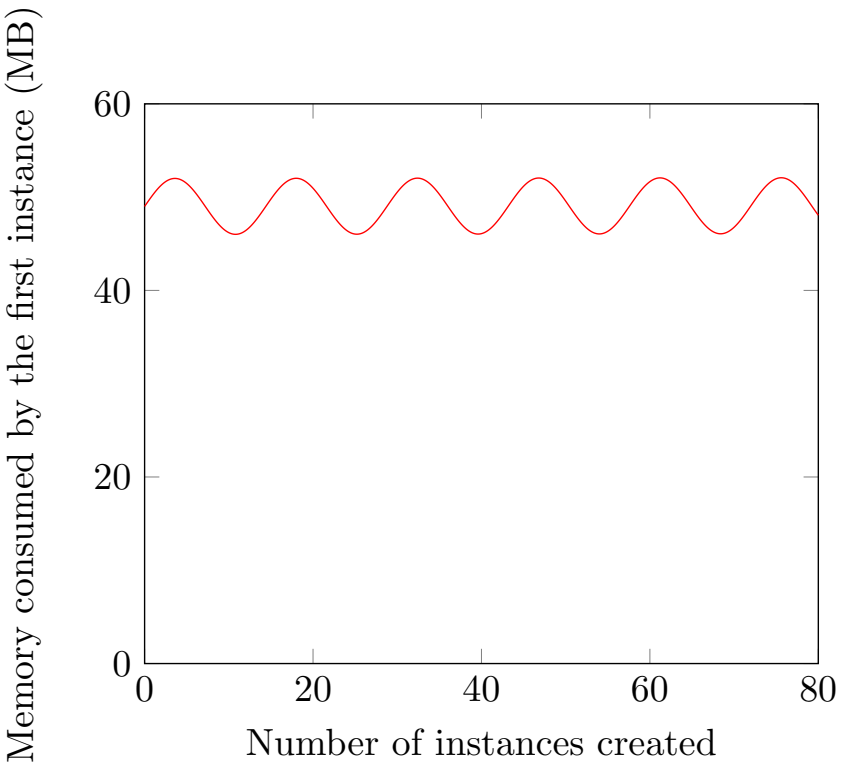

Figure 5.3: Memory consumption of the first instance with offers created

create an offer, that offers is saved in his own application/device. So, if we create one thousand offers in a node, the other nodes will not be affected by those offers.

Having these centralized/non-redundant offers brings an advantage against the malicious users that want to pollute our system. If those offers were saved in other nodes, a user could create a great number of dummy offers and increase the consumption of memory without any objective, just to degrade the system's performance.

But we can see some increase in memory, this is explained due to when we create an offer, we check with the others if there is an offer that could match ours. To do that is necessary to create a connection and send some messages. But after finding there isn't any other node with offers that could match, those objects are disposed, therefore reducing the memory used.

5.2.3 Ideal Price Deviation

A test similar to the scenario of Allocation Success Rate it was made calculate the Ideal Price Deviation. with the one made in Allocation Success Rate. We also made 8 runs, in the first run we started with 10 dummy users, whose offers will not be matched. Created two asks for the same resource where one was cheaper than the other. After that, we created a bid offer for that resource and observed if the offer that was associated with that bid is the cheapest one. At each

run we increment the dummy users, adding plus 10, we also added one more asks, for the same offer, but with the cheapest price, and added a bid for the same resource. The offers that were created by the dummy users, will not be matched.

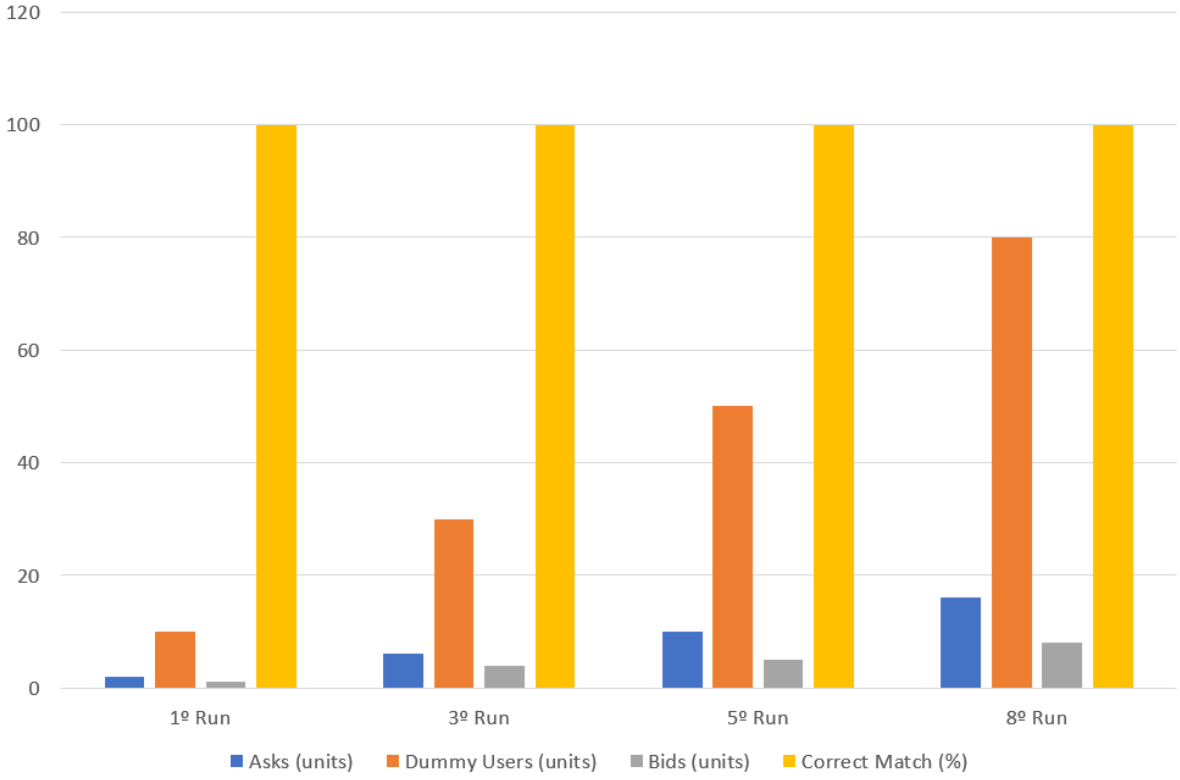

Figure 5.4: Results of a bid finding an ask with the cheapest price

As we can observe in Figure 5.4, we had a 100% matches success. Meaning, for a bid searching one given resource, it found an ask with the cheapest price.

5.2.4 Scalability

This test is different from the other because it doesn't involve getting results, we know how many messages we send for each offer created, by reading the code. The higher number of messages we sent is when we complete a transaction. The number of messages is 3: *get price request*, *send bid*, and *start transaction*, to one node. Supposing that we have x users that have an offer that matches with ours, and they will succeed to make the trade with the user with the cheapest offer, we will send x *get prices* plus 2 to finish the transaction. In Figure 5.5, we can observe the equation that corresponds to the number of messages sent.

Based on the graph, we could conclude that the number of messages sent to the user grow linearly with the number of offer that match with ours. One calculation that was not added in this function, was the number of messages sent to find all the offers that match with ours,

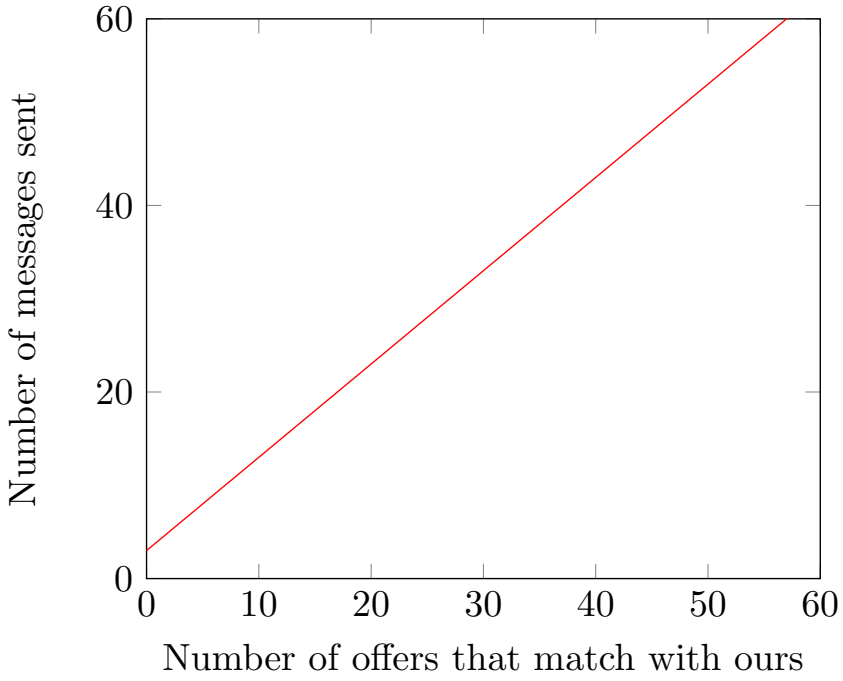

Figure 5.5: Number of messages sent based on the number of offers found

more precisely, to find the providers for a given offer. For that would be necessary to explore the libp2p code.

5.3 Discussion

After testing and observing the results, we can extract some conclusions:

- The amount of memory consumed by our application is independent of the number of peers that participate in the network. As it was explained before, the offers are not shared between users, only connections. And in the results we could observe that adding more users didn't result in an memory consumption increase, only some memory increase and after that, that memory was freed. That behavior is explained based on the creation and destruction of objects to make connections.

- Also we had a 100% to correct matches between offers. Meaning our system, even if it is peer-to-peer, can find the user that should be used to obtain the offer (if that user exists). Of course, this statement is for environments with a low number of users. In an environment with a higher number of users, it wasn't possible to test, so we can't make that statement.

- The number of messages that RATEE sends, grows linearly, which is better than expo-

nentially but worse than logarithmic. Another way can be used to reduce from linearly but would slowly reduce the time it would take to make a transaction.

Chapter 6

Conclusions

We started talking about the Cloud Computing that exists and the features they offer. The advantages and some of their limitations. Those limitations brought some changes, like shifting the computation and data storage to the edge environments, where personal devices could be used to assist that computation volunteering or by receiving any benefit.

This brought us to our work where we proposed to study the share of resources, to deploy applications, in an edge environment by using auction mechanisms. To reach that object we first have done a study about edge could and auction mechanisms to allocate resources. Creating a taxonomy evaluation for both topics. After that, we created a prototype, RATEE, which would have the previous features, an application that its purpose is to deploy applications using auction as a trade mechanism.

We implemented RATEE as an application that scales, it should handle a thousand requests, and low resources consumption because it will be stored in end-user devices. We showed the RATEE's dependencies to run in end-user devices, its operations to create offers and manage them, and the process that will match offers between themselves. RATEE is a peer-to-peer application using libp2p to implement all peer-to-peer logic and abstracting us from that complexity. Its P2P overlay network is based on Kademlia DHT which is also used by BitTorrent. In order to communicate with other peers, we use a message approach. Using libp2p a user provide a given resource and other user searches for that resource, which will get the user address. Using that address sends a message to communicate with it (getting a price, send a bid, etc).

In order to have some insurance that the application is working with the desired behavior tests were designed so that, in the future, if some change affects our functionality, those issues can be easily found by running the tests.

Finally, after design and implement RATEE, it was time to evaluate it. To evaluate we

used PowerShell scripts in order to create an environment that was necessary to make that test and obtain its results. Based on the results we could conclude that our application has a high success match rate (for the environment that was used to test), meaning all buyers found their respective sellers and vice-versa. Also, the memory consumption doesn't grow with the number of nodes that exist in the overlay.

6.1 Future Work

Like any other project, it is hard to say that something is finished, we could always improve or consolidate some feature. Bellow, we have a list of suggestion or improvements that would be interesting to make:

- One area that could be greatly improved is security. RATEE is a decentralized application that runs in user-devices, which can be easily manipulated by malicious users. We should ensure that malicious users could not create dummy offers.
- Also in part with security, add a Karma system which would penalize users that in the last instant of a transaction, they didn't deploy the container or didn't pay.
- Right now, we only provide a Web API to interact with our application. It would be great to add a user-friendly interface, which would increase the usability of the end-users.
- Adapt libp2p, making some changes in order to be more efficient for our use case. We found a lot of limitations in libp2p, one of them is the use of messages not being typified. We could only send a string (with JSON format) and deserialize it to an object.
- Create a robust simulator tool in order to create all the types of environments which could help in a better test of our tool.

Bibliography

[Alb08] Albert Greenberg, James Hamilton, David A. Maltz, Parveen Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *ACM SIGCOMM computer communication review, vol. 39, no. 1, pp. 68–73, Jan. 2008*, 2008.

[Alb17] Albert Jonathan, Mathew Ryden, Kwangsung Oh, Abhishek Chandra, Jon Weissman. Nebula: Distributed Edge Cloud for Data Intensive Computing. 2017.

[Ale17] Alex Glikson, Stefan Nastic, Schahram Dustdar. Deviceless Edge Computing: Extending Serverless Computing to the Edge of the Network. *Proceedings of the 10th ACM International Systems and Storage Conference Article No. 28*, 2017.

[Ali09] Ali Haydar Özer, Can Özturan. AN AUCTION BASED MATHEMATICAL MODEL AND HEURISTICS FOR RESOURCE CO-ALLOCATION PROBLEM IN GRIDS AND CLOUDS. *Fifth International Conference on Soft Computing, Computing with Words and Perceptions in System Analysis, Decision and Control*, 2009.

[Alv04] Alvin AuYoung, Brent N. Chun, Alex C. Snoeren, Amin Vahdat. Resource allocation in federated distributed computing infrastructures. *In Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT Infrastructure*, 2004.

[Ami15] Amin M. Khan, Felix Freitag, Luís Rodrigues. Current Trends and Future Directions in Community Edge Clouds. 2015.

[Ant01] Antony Rowstron, Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *Proceeding, SOSP 01 Proceedings of the eighteenth ACM symposium on Operating systems principles, Pages 188 - 201*, 2001.

[Avi10] Avinash Lakshman, Prashant Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review, Volume 44 Issue 2, April 2010*, 2010.

[BM08] Ingmar Baumgart and Sergio Mies. S/kademlia: A practicable approach towards secure key-based routing. volume 2, pages 1–8, 01 2008.

[BRFN15] Roger Baig, Ramon Roca, Felix Freitag, and Leandro Navarro. guifi.net, a crowdsourced network infrastructure held in common. *Computer Networks*, 90:150 – 165, 2015. Crowdsourcing.

[Cla16] Claus Pahl, Sven Helmer, Lorenzo Miori, Julian Sanin, Brian Lee. An Information Framework for Creating a Smart City Through Internet of Things. 2016.

[Cos03] Costas Courcoubetis, Richard Weber. Cost-based Pricing. *John Wiley & Sons, pp. 161–194*, 2003.

[CWSR12] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6. IEEE, 2012.

[Dav05] David Hausheer, Burkhard Stiller. PeerMart: The Technology for a Distributed Auction-based Market for Peer-to-Peer Services. *IEEE International Conference on Communications*, 2005.

[Dha04] Dhananjay (Dan) K. Gode, Shyam Sunder. Double auction dynamics: structural effects of non-binding price controls. *Journal of Economic Dynamics and Control, Volume 28, Issue 9, July 2004, Pages 1707-1731*, 2004.

[Din14] Dinil Mon Divakaran, Mohan Gurusamy, Mathumitha Sellamuthu. Bandwidth allocation with differential pricing for flexibledemands in data center networks. *Computer Networks, vol. 73, no. 1, pp. 84–97*, 2014.

[Fer13] Fernando Costa, Luís Veiga, Paulo Ferreira. Internet-scale support for map-reduce processing. *Journal of Internet Services and Applications, vol. 4, no. 1, pp. 1-17*, 2013.

[Hel] Heli Zhang, Hossein Badri, Heli Zhang, Fengxian Guo, Hong Ji, Chunsheng Zhu. Combinational Auction-Based Service Provider Selection in Mobile Edge Computing Networks. *IEEE Access*.

[Hui14] Hui Wang, Huaglory Tianfield, Quentin Mair. Auction Based Resource Allocation in Cloud Computing. *Multiagent and Grid Systems, 2014, Volume 10, Number 1, May 2014, pp. 51-66*, 2014.

[Hyu14] Hyunseok Chang, Adiseshu Hari, Sarit Mukherjee, T.V. Lakshman. Bringing the Cloud to the Edge. 2014.

[Jio14] Jiong Jin, Jayavardhana Gubbi, Slaven Marusic, Marimuthu Palaniswami. An Information Framework for Creating a Smart City Through Internet of Things. 2014.

[Kev] Kevin Lai, Bernardo A. Huberman, Leslie Fine. Tycoon: a Distributed Market-based Resource Allocation System.

[Lan03] Landon P. Cox, Brian D. Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. *Proceeding, SOSP 03 Proceedings of the nineteenth ACM symposium on Operating, systems principles, Pages 120-132*, 2003.

[Mah09] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, Nigel Davies. The Case for VM-based Cloudlets in Mobile Computing. 2009.

[Mar00] Robert C. Martin. Design principles and design patterns. 2000.

[ME02] Petar Maymounkov and David Eres. Kademlia: A peer-to-peer information system based on the xor metric. volume 2429, 04 2002.

[Mic17] Michał Król, Ioannis Psaras. NFaaS: named function as a service. *Proceedings of the 4th ACM Conference on Information-Centric Networking, Pages 134-144*, 2017.

[Mur05] Muralidhar V. Narumanchi, José M. Vidal. Algorithms for Distributed Winner Determination In Combinatorial Auctions. *Agent-Mediated Electronic Commerce. Designing Trading Agents and Mechanisms pp 43-56*, 2005.

[Ngu17a] Nguyen Cong Luong, Ping Wang, Dusit Niyato, Wen Yonggang, Zhu Han. Resource Management in Cloud Networking Using Economic Analysis and Pricing Models: A Survey. *IEEE Communications Surveys & Tutorials, Volume: 19, Issue: 2, Secondquarter 2017*, 2017.

[Ngu17b] Nguyen Cong Luong, Ping Wang, Dusit Niyato, Wen Yonggang, Zhu Han. Resource Management in Cloud Networking Using Economic Analysis and Pricing Models: A Survey. *IEEE Communications Surveys & Tutorials, Volume: 19 , Issue: 2 , Secondquarter 2017*, 2017.

[Nit16] Nitinder Mohan, Jussi Kangasharju. Edge-Fog Cloud: A Distributed Cloud for Internet of Things Computations. 2016.

[OAPV04] David Oppenheimer, Jeannie Albrecht, David Patterson, and Amin Vahdat. Scalable wide-area resource discovery. In *USENIX WORLDS*, volume 4, 2004.

[Oza12] Ozalp Babaoglu, Moreno Marzolla, Michele Tamburini. Design and Implementation of a P2P Cloud System. 2012.

[Pao16] Paolo Bellavista, Alessandro Zanni. Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi. 2016.

[Par16] Parnia Samimi, Youness Teimouri, Muriati Mukhtar. A combinatorial double auction resource allocation model in cloud computing. *Information Sciences, Volume 357, 20 August 2016, Pages 201-216*, 2016.

[Pet11] Peter Mell, Timothy Grance. The NIST Definition of Cloud Computing: Recommendations of the National Institute of Standards and Technology. 2011.

[Phi13] Philip Mayer, Annabelle Klar, Rolf Hennicker, Mariachiara Puviani, Francesco Tiezzi. The Autonomic Cloud: A Vision of Voluntary, Peer-2-Peer Cloud Computing. 2013.

[Raj14] Rajdeep Dua, A Reddy Raja, Dharmesh Kakadia. Virtualization vs Containerization to support PaaS. *IEEE International Conference on Cloud Engineering*, 2014.

[Rog04] R. Owen Rogers. Acceptance testing vs. unit testing: A developer's perspective. Springer Berlin Heidelberg, 2004.

[Sau11] Saurabh Garg, Rajkumar Buyya. Market-Oriented Resource Management and Scheduling: A Taxonomy and Survey. *Cooperative Networking, Chapter 14*, 2011.

[Sha10] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, Harbinder Bhogan. Volley: Automated Data Placement for Geo-Distributed Cloud Services. *NSDI, San Jose, California, USA, Sep. 2010, pp. 17–32*, 2010.

[Tay18] Tayebeh Bahreini, Hossein Badri, Daniel Grosu. An Envy-Free Auction Mechanism for Resource Allocation in Edge Computing Systems. *2018 Third ACM/IEEE Symposium on Edge Computing*, 2018.

[Tim12] Tim Verbelen, Pieter Simoens, Filip De Turck, Bart Dhoedt. Cloudlets: Bringing the cloud to the mobile user. 2012.

[Vin09] Vincenzo D. Cunsolo, Salvatore Distefano, Antonio Puliafito and Marco Scarpa. Cloud@Home: Bridging the Gap between Volunteer and Cloud Computing. 2009.

[Wei10] Wei-Yu Lin, Guan-Yu Lin, Hung-Yu Wei. Dynamic Auction Mechanism for Cloud Resource Allocation. *Proceeding CCGRID '10 Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing Pages 591-592*, 2010.

[Xin12] Xingwei Wang, Jiajia Sun, Min Huang, Chuan Wu, Xueyi Wang. A resource auction based allocation mechanism in the cloud computing environment. *IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012.

[Xin14] Xingwei Wang, Xueyi Wang, Cho-li Wang, Keqin Li, Min Huang. Resource Allocation in Cloud Environment: A Model Based on Double Multi-Attribute Auction Mechanism. *IEEE 6th International Conference on Cloud Computing Technology and Science*, 2014.