# RATEE - Resource Auction Trading at Edge Environments

Diogo Paulo Dias
Instituto Superior Técnico, Universidade de Lisboa

***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. The resources of these devices are allocated to host applications. These applications can be allocated in various ways, voluntarily, or by exchange. What we're 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.**

## I. Introduction

Cloud Computing is a mature environment heavily used because of its properties like global access, pay for what you use, resource elasticity, and others [1]. To meet these alluring properties, typically Cloud Computing is executed with a couple of 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 [2]. This information feeds the network's edge with a lot of data and this data needs to be transfered to the Cloud data centers in order to be processed. The transfer of large amounts of data will cause bandwidth saturation and a big latency. 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 [3], [4], laptops, desktops, routers, hubs and others. 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.

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.

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 [5]. 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, the supply is higher than demand. Pricing/economic 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 efficienlty overcome with an economic/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.

## II. Related Work

There are already some systems that implement allocation resources using market algorithms. We will talk a little about them and their differences.

### A. Multi-Round-Sealed Sequential Combinatorial Auction

A multi-round-sealed sequential combinatorial auction mechanism is proposed by Zhang et al. [6]. 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.

### B. Tycoon

Tycoon [7] 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. 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.

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 Quality-of-Service is not secured. The Quality-of-Service is not secured 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.

### C. PeerMart

PeerMart [8] 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 public cryptographic keys 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[1]. It is a tool that is implemented in Java and the the overlay of the network is based on Pastry.

### *D. Lin et al.*

Lin et al. [9] 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 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.

### *E. Double Multi-Attribute Auction*

Wang et al. [10] 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.

### *F. Combinatorial Double Auction Resource Allocation*

Samini et al. [11] 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.

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

## III. *RATEE*

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.

### A. *Requirements*

Our target is the edge environment, meaning we want to support thousands of users. Volunteer Networking and Computing platforms, like the GUIFI.net[2] [12] 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 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.

### B. *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 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.

[2]guifi.net

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.

### C. *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

### D. *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[3]. This library creates a p2p network where its structure is based on Kademlia DHT.

*1) Kademlia DHT:* Kademlia [13] 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

[3]https://libp2p.io/

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.

## E. 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. This scale better but is much more difficult to implement and maybe with some inefficiencies in small environments with a low number of users.

Libp2p supports both approaches but the second one has a lot of problems in public networks 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 search 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). The way the method provides works is by getting the key and transforms it to a DHT Id. After that it finds the nearest neighbors to that id, and they will save the DHT id and the provider of that key. To 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.

**Listing 1** Offer structure

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

**Listing 2** Get Price request message

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

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.

The response contains the information about the owner, and

**Algorithm 1** Handler of a get price request command

```
0: function GETPRICEHANDLER($resourceId$, $isBid$)
0:     $allOffers \leftarrow isBid?askTable : bidTable$
0:     $requiredOffers \leftarrow allOffers.filter(a => a.resourceId == resourceId)$
0:     $response \leftarrow createReponseMessage(requiredOffers)$
0:     return $response$
0: end function=0
```

**Listing 3** Get Price response message

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

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 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

**Listing 4** Send bid request message

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

**Algorithm 2** Handler for bid request command

```
0: function BIDREQUESTHANDLER($bid$)
0:     $ask \leftarrow askTable.single(a- > a.id == bid.askId)$
0:     if $ask! = null$ & $askRequested.lastReserved.getTime() + 4000 < Date.now$ then
           ASKTABLE.REMOVE($ask$)
           $ask.lastReserved \leftarrow Date.now$
           $ask.ownerId \leftarrow bid.ownerId$
           $response \leftarrow CreateSuccessResponseMessage(ask)$
           return $response$
0:     end if
0:     return $CreateErrorResponseMessage()$
0: end function=0
```

**Listing 5** Send bid response message

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

for that bid. In case the buyer doesn't 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 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.

**Listing 6** Send transaction request message

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

**Listing 7** Send transaction response message

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

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't 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** Node behavior after a user creates a bid

```
0: function CREATEBID($bidOffer$)
0:    $bidsTable \leftarrow bidOffer$
0:    $resourceId \leftarrow ResourceMapper(bidOffer)$
0:    $providers \leftarrow libp2p.getProviders(resourceId)$
   $provider \in providers$
0:    $prices \leftarrow libp2p.getPrices(resourceId)$
0:    $price \in prices$
0:    $response \leftarrow libp2p.sendBid(bidOffer)$
0:    if $response == ACCEPTED$ then
0:       $libp2p.StartTransaction(response, bidOffer)$
0:    end if
0:
0: end function=0
```

Algorithm 3 summarizes the bid creating process.

### F. Implementation

The algorithms presented previously 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.

*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 responsibility, This modularity helps to have code more maintainable, modifiable, and testable. When writing code, SOLID principles [14] 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 1 change we can observe each module and interactions between themselves.

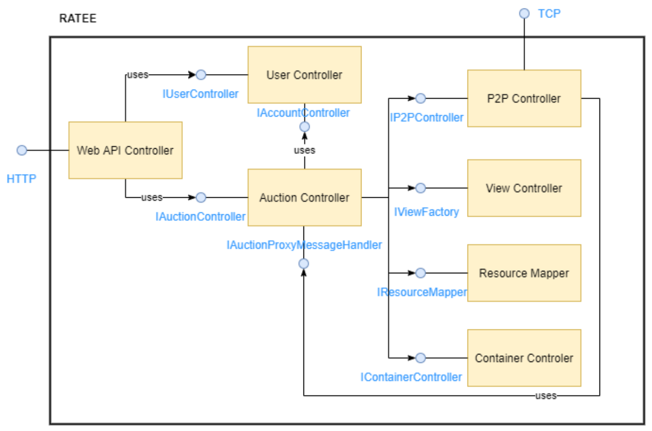

Fig. 1: RATEE components and interfaces relationships

## IV. Results

### A. Evaluation Methodology

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

*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 didn't return a match, it is because it didn't exist. But in a peer-to-peer environment, we don't 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.

*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, has 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.

*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 don't buy the cheapest resource because of the distributed environment.

*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.

### B. Experimental Evaluation

To test, it was used computer with 8 GB of memory and a dual core i7 as processor. Some limitations in the number of instances created for the results were due to limitations of the machine used to test.

*1) Allocation Success Rate:* In order to test the Allocation Success Rate, dummy users were created which would have offers that wouldn't 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 2, we can observe the results of those runs (we didn't show all the results of the eight runs, due to results being equal).

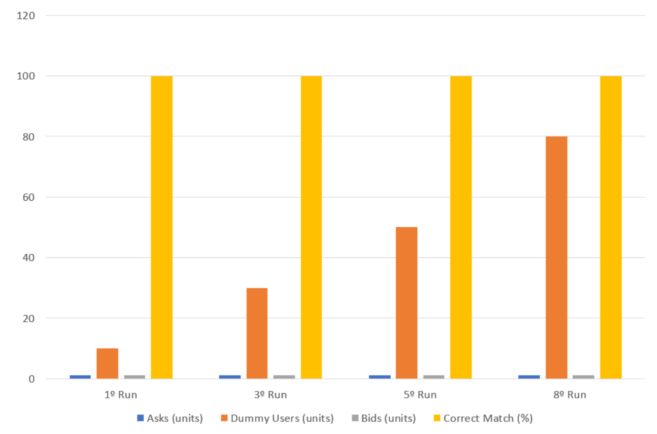

Fig. 2: Results of a bid finding an ask for the same resources

From the results we can observe that even by incrementing the number of dummy users with dummy offers, the user that created the bid, always found the respective ask. Resuming, for the 8 runs, the hit success rate was 100%.

*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 3 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 4.

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 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

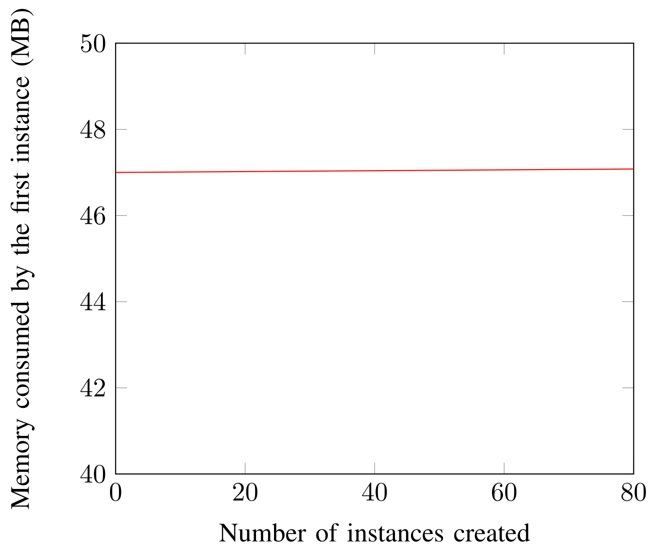

Fig. 3: Memory consumption of the first instance created

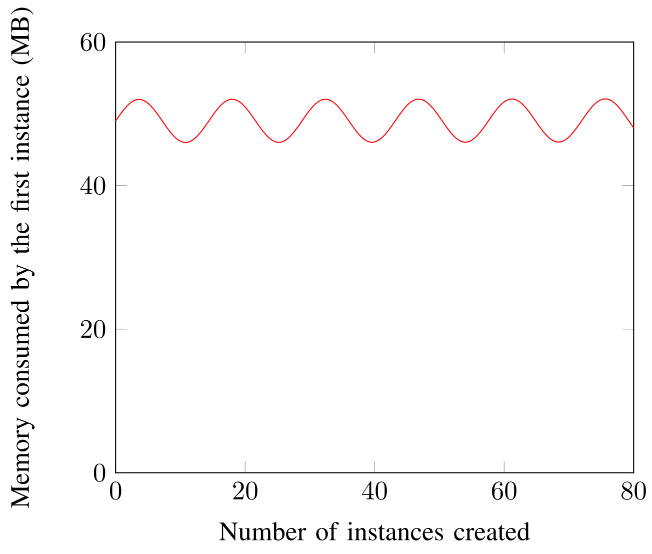

Fig. 4: Memory consumption of the first instance with offers created

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.

*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.

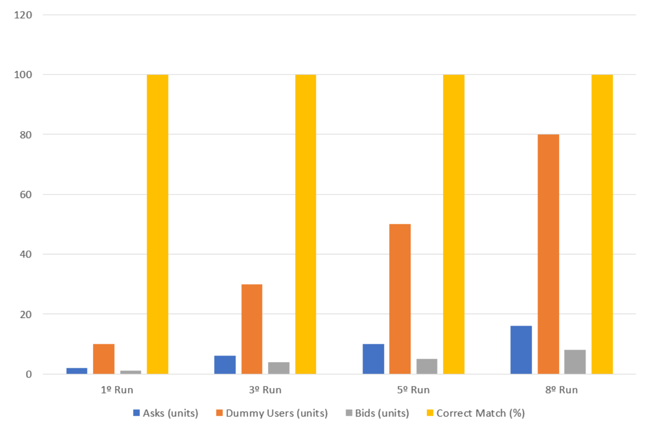

Fig. 5: Results of a bid finding an ask with the cheapest price

As we can observe in Figure 5, we had a 100% matches success. Meaning, for a bid searching one given resource, it found an ask with the cheapest price.

*4) Scalability:* This test is different from the other because it doesn't involve getting results, we know how many messages we sending 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 6, 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 wasn't added in this function, was the number of messages sent to find all the offers that match with ours. For that would be necessary to explore the libp2p code.

## C. 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

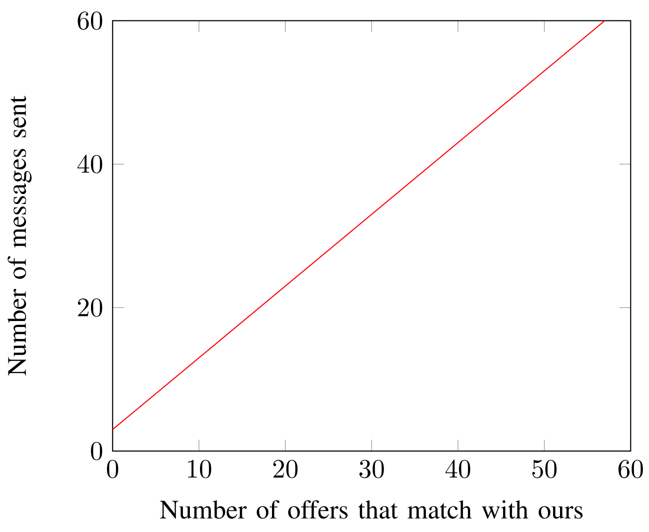

Fig. 6: Number of messages sent based on the number of offers found

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 exponentially 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.

## V. Conclusion

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.

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).

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.

## References

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

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

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

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

[5] 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.

[6] 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*.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] R. Baig, R. Roca, F. Freitag, and L. Navarro, "guifi.net, a crowdsourced network infrastructure held in common," *Computer Networks*, vol. 90, pp. 150 – 165, 2015, crowdsourcing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128615002327

[13] P. Maymounkov and D. Eres, "Kademlia: A peer-to-peer information system based on the xor metric," vol. 2429, 04 2002.

[14] R. C. Martin, "Design principles and design patterns," 2000.