Project - Auctions@FaaS (revised)

Manuel Domingues
Number: 82437
Email: manuel.domingues@hotmail.com

Instituto Superior Técnico

Abstract. The use of cloud computing and backend services have become a rising necessity. With new technologies and the rise of serverless computing, users have seen an ease on deployment of applications and cloud management. Users can also take advantage of the emergence of edge or fog computing paradigms to have lower latencies on their requests. However, the pricing for cloud resources is set by cloud providers, this can mean less advantageous pricing for users and sometimes even the providers themselves.
In this work we propose a double auction based pricing where both users and providers make bids and compete for cloud resources. In order to achieve this, we present a new middleware node between users and edge nodes, StockWhisk, that will serve as an auctioneer and delegate requests from users to the edge nodes. This work is an extension of OpenWhisk[1] and will expand upon the previous work of AuctionWhisk [4].

Keywords: Function-as-a-service; Serverless Computing; Edge Computing; Fog Computing; Auction;

[1] https://openwhisk.apache.org/

Table of Contents

1 Introduction

The use of serverless computing has been rising, as developers find it more attractive due to the reduction of cloud management costs and of work for application deployment. Not only that, but Function-as-a-Service (FaaS) solutions can provide cost savings for application execution, depending on the use case, and are specially used for bursty and on-demand workloads [10], and aiming at better sustainability [20].

Many of the functions that run on FaaS have small execution times and are prejudiced greatly by latency, there is a lot of research done on trying to reduce latency of FaaS executions [8], [9].

In fog computing, edge nodes are the closest to end users which give the benefit of lower latencies when users send their requests, one of the works extensively studied, AuctionWhisk [4], combines the use of fog computing [5] by trying to bring FaaS computation closer to edge nodes.

The current price methods of FaaS are usually based on memory consumption or execution duration [26] and are set by each provider, the user can only choose which provider they use.

1.1 FaaS and edge pricing

Edge nodes have limited resources and in times of high demand the decision needs to be made on which requests get the edge resources.

Similarly to computation in cloud nodes, the pricing is not dynamic. Amazon for example, offers a similar FaaS pricing model for computation in edge nodes[2], this pricing is set by Amazon and does not change quickly according to market demand. This way of pricing the service might be detrimental to users, for example, if they want to take advantage of low demand time frames.

Likewise, Amazon and other cloud providers could benefit from higher profits in moments of high demand.

A suggested alternative for pricing is to use auctions for cloud resources, AuctionWhisk provided a decentralized auction system for computing nodes where clients would compete through bidding on who got their request accepted. Rejected requests would trickle down to the cloud, getting further away from the edge, until they would be accepted. This provides an advantage to users who are willing to pay more for the privilege of using edge nodes with lower latency. Furthermore, users could now take advantage of times of lower demand to have lower costs.

1.2 Shortcomings

Although AuctionWhisk demonstrated that it is possible to allocate resources on the edge, the pricing model was not compared to existing ones, nor did

[2] https://aws.amazon.com/lambda/pricing/

they compare different pricings models. Auction-based systems, including those designed for the edge, .e.g. [7,11] are not tailored to FaaS.

Furthermore, the type of auction used is a single auction, providers do not compete for winning requests. The change for a double auction and different auction mechanisms could prove beneficial to maximize providers' revenues and minimizing the users' costs.

Lastly, if a user bids a low value, their request could take a long time until it finds an accepting processing node, this added latency could make it costly for users that need requests to be handled as soon as possible.

1.3 Objectives

The goal of this work is to introduce a double auction mechanism for cloud request allocation at the edge. We try to achieve this by introducing an extension to AuctionWhisk and in tandem OpenWhisk, called StockWhisk. In order to achieve this we will work as follows:

1. Research the state of the art, current implementations and technologies of FaaS, edge computing and auction theory.
2. Design and implementation of StockWhisk, bringing a double auction mechanism at edge cloud, by extending the existing AuctionWhisk and creating a more centralized middleware that sits between clients and edge nodes.
3. Evaluate the made solution using the metrics and workloads mentioned in Section 4, in order to check if our solution is efficient, fulfills requirements and meets expected results.

1.4 Document organization

This work has the following structure:

- Section 2 is a study on current state of the art on FaaS and auctions.
- Section 3 presents the proposed architecture, StockWhisk.
- Section 4 provides the evaluation methodology to be used.
- Section 5 provides some final remarks on the work.
- Appendix A is the planning for the development of StockWhisk.

2 Related Work

In this section we will study the current state of the art in order to better implement our solution. Section 2.1 explains how FaaS works and how it is used, with a little insight on edge computing. Section 2.2 provides an insight on the type of auctions and mechanisms that exist. Finally, Section 2.3 showcases two examples of double auctions used in computer networks and their results.

2.1 Function-as-a-Service and edge computing

As time went by and technologies matured, we have seen a distancing from the traditional methods of providing servers and backend services for applications and a rise in use of new ones[3]. With the introduction of Infrastructure-as-a-Service, users are provided with instances with Virtual Machines where they choose the full software operation within them. Users can rent a variety of instances, paying more or less depending on the amount of resources they hold, e.g. CPU cores or RAM size. In the end, the user will pay for the instances they allocate and the resources they hold. Although the cloud provider removed the hassle of physically managing the servers, it is still up to the developers, through these Virtual Machine instances, to manage most software components, mainly load balancing and scaling methods.

The first major provider of this type of architecture was the Amazon Elastic Compute Cloud (EC2)[4] provided in Amazon Web Services in 2006, this service is widely used today and other providers have since joined the market, such as Azure from Microsoft[5], Google Cloud[6].

With these developments, more engineers were needed that were specialized in managing these cloud services, depending on the application, load balancing and auto scaling might prove to be non trivial challenges when the goal is to minimize costs and maximize efficiency.

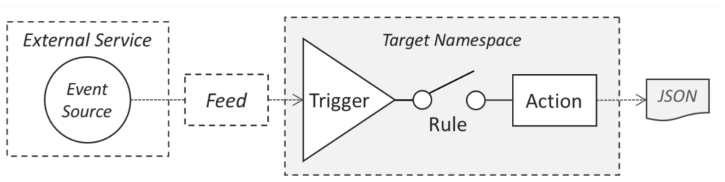


Fig. 1. OpenWhisk Programming Model [1]

In order to further remove this responsibility of cloud management from the user, cloud providers like the ones mentioned above, or the open source Apache OpenWhisk[7], vizualized at Figure 1 offer a different service model, Function-as-a-Service. In FaaS, the auto-scaling and runtime environment are handled by the cloud provider. The developer only needs to worry about producing a function that will be run on this environment. First triggered by some event, like a storage

[3] https://www.datadoghq.com/state-of-serverless/
[4] https://aws.amazon.com/ec2/
[5] https://azure.microsoft.com/
[6] https://cloud.google.com/
[7] https://openwhisk.apache.org/

update, the request with the desired function to run is sent to the environment and the function is loaded into it. After it is run, the result is then sent back and all the resources on this environment are freed. One drawback of FaaS is that running operations are stateless, meaning that they have no memory on their own. After the operation is done, for most use cases, there needs to be another form of storage where the result can be sent.

In fact, by studying how FaaS is being used by cross examining papers [10], it was found that 61% of FaaS applications are used in conjunction with storage services and 48% with databases. Only 12% of apps are standalone, meaning that most FaaS implementations are only parts to bigger systems and used for more specific cases.

Maybe due to this use of FaaS and because of its advantages to scalability, it is no surprise to see that 84% of its use cases are for bursty workloads. In tandem, 86% are triggered on demand calls as FaaS advantages shine with its great ability to auto-scale and with the reduction of overheads in high volume requests due to warm starts i.e. the environment already being set up for the function.

It is interesting to see however the remainder of applications that correspond to scheduled triggering, proving that FaaS is also versatile in the way it is used. About half of these scheduled events are for operations and monitoring functions, automating management and DevOps pipelines.

Although FaaS is typically thought out to be used for single function purposes, these can be stacked and connected, creating what's called serverless workflows. Most of these workflows are still relatively small however, 72% being less than 10 functions, and half of them are sequential in nature.

One of the main issues with FaaS is the need for low latency. Some functions are small and have very low execution times, like HTTP requests, by having high latency these requests spend a high percentage of their lifespan not in execution. A big issue that causes latency in FaaS is the way the execution starts, since the idea of FaaS is that providers manage the environment. This environment might not be ready to receive the function, with a runtime (like JVM for java applications) having to be loaded on demand on the container, this is called a cold start and adds a significant amount of latency. Generally, a container keeps a runtime for some time so it does not need to load it again, should further requests for that runtime be received. However, it discards this runtime should a different function arrive, that needs resources, or a certain amount of time has passed.

To combat this issue, *Catalyzer* [8] saves a checkpoint image from a function instance. Furthermore, it introduces a new function sfork that copies the environment of another sandbox instance without the application specific data.

Photons [9] presents another solution where functions share the same runtime, *Photons* manages to keep information private in the application layer with locking primitives and unique identifiers. *Photons* also reduces the memory used since there is no longer a need for all applications to load their own environment.

Another way to reduce latency is to have functions run closer to end users, as Varshney et al. [27] put it, Fog computing is a "resource layer that fits between the edge devices and the cloud data centers, with features that may resemble either.". Although edge nodes (devices) offer advantages over latency, they have limited resources, significantly less than what the classic cloud nodes can provide, and as such their availability is not always guaranteed [5].

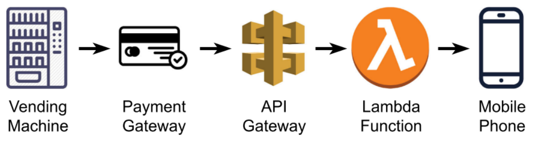


Fig. 2. FaaS backend of a coca-cola vending machine

Demonstrated by two real world use cases of FaaS, different motivations result in different uses and approaches for the architecture. On the one hand, Coca-cola's vending machine backend, Figure 2, has the goal of minimizing costs, with a simple and straightforward one function only implementation.

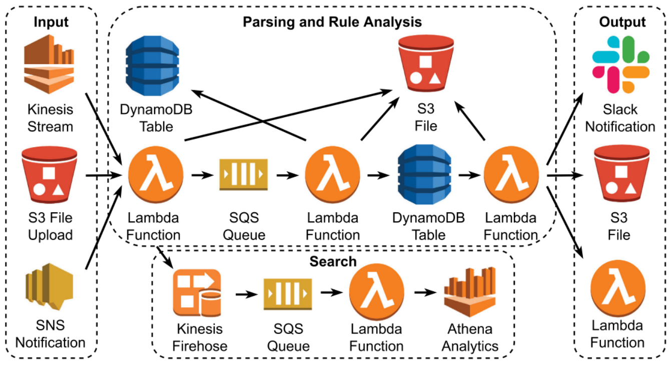


Fig. 3. Airbnb's StreamAlert, using a FaaS workflow

On the other hand, AirBnB's stream alert's, Figure 3, goal is to minimize operational overheads, it encompasses a small, but complex workflow, having more than 2 functions within it.

2.2 Auctions

An auction is a way to sell goods without initially setting the price of said goods, instead, the buyers are the ones that set the good's price by competing with each other during the sale. The most commonly known being the ascending auction, where buyers offer bids, which are offers to the good being sold, openly, the one that offers the highest value wins the auction and so, the good being sold.

Auctions have been present in human history for a long time, first recorded in Greece, detailing the sale of wives in Babylon, it was later widely used by the Romans [18] when selling their spoils of war. Although after Roman times they were much less common, they saw a resurgence in pre-modern history in England with candle auctions [21]. In a candle auction, a candle is lit and the auction ends when the candle expires and the fire is out. It was made this way so no buyer really knew when the auction would end, the expiring of the candle signals the end of the auction and therefore the last bid made when the auction ends, is the winning bid. Since the first known auction house opened in Stockholm, Sweden in the 17th century, more auction houses have opened up throughout the world. Auctions have seen a great rise in usage in modern times, governments using them for the sale of a variety of goods, such as treasury bills, contracts and mining rights. With the appearance of the world wide web and online sales, auctions followed, perhaps the most known example being with Ebay.com where sellers can post their items for sale online in an ascending auction.

2.2.1 The four basic Types

Although many different types of auctions exist, there are four basic types of auctions. [13]

The english auction. As mentioned before it is also known as the ascending auction. In this auction buyers make bids for the item being sold, continuously rising the bid price as they compete against each other. The auction ends when noone else makes a bid over the previous one. The winning bid pays the bid amount and gets the item.

The dutch auction. Also known as descending auction. In this auction a high price is announced by the seller, this price is brought down little by little until a buyer makes a bid, this first bid being the winning bid. This type is usually faster in concluding the auction, as there is only one bid being made. Similarly to the english auction, all bidders have information on each other's bids, or rather, lack of bids.

The first-price sealed-bid auction. For this auction, buyers make one and only one bid in secret. All bids are revealed at the same time and the highest one wins. Contrary to the previous two auction, buyers have no information on each other's bids as the bids are secret.

The second-price sealed-bid auction. Also known as Vickrey auction, it is similar to first-price sealed-bid auction, all buyers make a bid in secret and these bids are revealed at the same time. However, although the highest bid is the winning one, the amount the buyer needs to pay for the item is not his own bid, the highest, they pay the amount on the second highest bid instead.

So far all of these auction types share one thing in common, there is either only one seller or, if there is more than one item being sold, there is no direct competition between sellers. One auction type that changes this and provides not only competition among the buyers, but also among the sellers is the double auction.

2.2.2 Double auction and mechanisms

In a **double auction**, the buyers aren't the only ones making bids, the sellers make bids as well, competing with each other. Whereas the buyers compete to see who offers the most for a certain good, the sellers compete to see who is willing to receive the least for their offered good. This creates a rather complex market where bids are made from all sides, with one example of a double auction being the stock exchange.

Finding the payment price for a double auction is not a trivial task, Myerson and Satterthwaite provide four requirements that a perfect economic mechanism design should accomplish [19]. These four requirements are as follows:

1. **Individual Rationality (IR)**. None of the participants should lose from participating in the auction, to be more precise, no buyer should make a payment (p) higher than his bid (b) and no seller should receive payment lower than their asking price (s). In summary, $p \leqslant b$ and $p \geqslant s$.
2. **Balanced Budget (BB)**. Either a Strong Balanced Budget (SBB), where the auctioneer does not win or lose any money with the trades, or a Weak Balanced Budget (WBB), where the auctioneer does not lose any money with the trades but might win some.
3. **Incentive compatibility (IC)**. The best strategy for all participants should be to report, for a bid or asking price, the true value of the good. There should be no advantage to one participant if they knew the other participants bid values beforehand; therefore their bid should not change regardless of their knowledge of other participants intents.
4. **Pareto efficiency (PE)**. After trading has concluded, the goods should be in the hands of the participants who valued them the most.

However, their own findings show that a perfect mechanism is impossible to achieve, this is called the Myerson–Satterthwaite theorem, as at least one of the requirements ends up being violated. Therefore, when conducting a double auction one must take this into account and pick the mechanism to employ, choosing carefully which requirements will be broken.

Babaioff and Nisan [3], provide a good summary on how different mechanisms work and how the payment values are found for each one. Commonly across them however, at the start of the double auction each participant begins by reporting their bid values, a seller i reports their good's cost S_i, which might be different from their real cost s_i, likewise each bidder i reports a bid B_i, which might be different from their real value s_i. Then, in order to find the payment price p we first need to find the *supply and demand* curves, we do this by sorting the asking price bids from the sellers, $S_1 \leqslant S_2 \leqslant S_3$... and the same for the buyer's bids, $B_1 \geqslant B_2 \geqslant B_3$.... If the participants reported their true values then the maximum efficient trade quantity l is the largest index where $B_l \geqslant S_l$. All participants that do not satisfy this condition do not trade in the auction and have a payment of 0.

For a **k-double auction** a parameter k is chosen, where $k \in [0, 1]$. This value is then used to find the payment price: $P = k \cdot S_l + (1 - k) \cdot B_l$. The payment value is, therefore, always inside the interval $[S_l, B_l]$

For example, for the $\frac{1}{2}$-double auction, $P = \frac{1}{2} \cdot S_l + \frac{1}{2} \cdot B_l = \frac{S_l + B_l}{2}$, this mechanism is also called the average mechanism.

Although this mechanism satisfies the other requirements, it falls short on incentive compatibility. The participants dominant strategies are not to be truthful, in fact, due to the way the payment method is calculated, a buyer has an advantage by reporting an untruthful value lower than the true one, likewise a seller's dominant strategy is to report a value higher than the true one. Despite this shortcoming, this mechanism has still shown to have good results in some situations [24].

One mechanism that aims to satisfy the incentive compatibility requirement, aiming for all participants to report their true values, is the **Vickrey-Clarke-Groves (VCG)** mechanism. In this mechanism, the trade quantity is also maximized at l, but the payment value is different between the buyers and the sellers. The sellers receive payment $p_S = min(S_{l+1}, B_l)$ and the buyers pay $p_B = max(S_l, B_{l+1})$, this insures that reporting a non true value is not a dominant strategy, as the payment will be the report of the other side and the participant might run the risk of not trading at all.

The shortcomings of this mechanism lie exactly on this payment difference, since the buyer pays a value that is lower than what the seller receives, $p_B \leqslant p_S$, the trade runs at a budget deficit, not satisfying the balanced budget condition, the auctioneer would have to subsidize the trade.

Due to this budget deficit and other constraints [23], **VCG** mechanism is generally not practical, rarely being used outside theoretical experiments. A more practical and still truthful mechanism is the **Trade Reduction (TR)** mechanism, with it, we forego on efficiency, with l-1 units being traded. Payment is simply decided by the values reported by the lth participants, these will be different for buyers and sellers, with $p_B = B_l$ and $p_S = S_l$. Since $B_l \geqslant S_l$, despite having a weak balance budget, the auction does not have a strong balanced budget, there is a budget surplus. This surplus can either go to the auctioneer or be used to pay for additional costs, for example, the cost of moving the items across locations.

By foregoing the "last" l trade we can achieve truthfulness, the participants up to l-1 have no incentive to not being truthful since it will not change their paying price and lth participants do not trade, hence they have no incentive of reporting a different value either.

McAfee provided an extension to the **TR** mechanism in order to sometimes have no efficiency loss [17]. In a **McAfee** mechanism we first see if the payment value $p = \frac{S_{l+1}+B_{l+1}}{2}$ is viable for the l buyer and seller, meaning, if $p \in [S_l, B_l]$. If so, l trades are conducted with payment p, otherwise, a regular **TR** auction is conducted. Although **McAfee**'s mechanism offers an improvement, it might not always be applicable and when not so it suffers from the same shortcomings as **TR**.

A different approach to mechanism design is by having a random factor built into it. In the α **Reduction** mechanism, a fixed value $0 \leqslant \alpha \leqslant 1$ is set, then the auction has a probability of α of proceeding with a **TR** mechanism and a probability of $1-\alpha$ of proceeding with a **VCG** mechanism. Since both mechanisms employed are incentive compatible, α **Reduction** is as well. Its budget balance and its efficiency vary depending on the mechanism employed and the value of α.

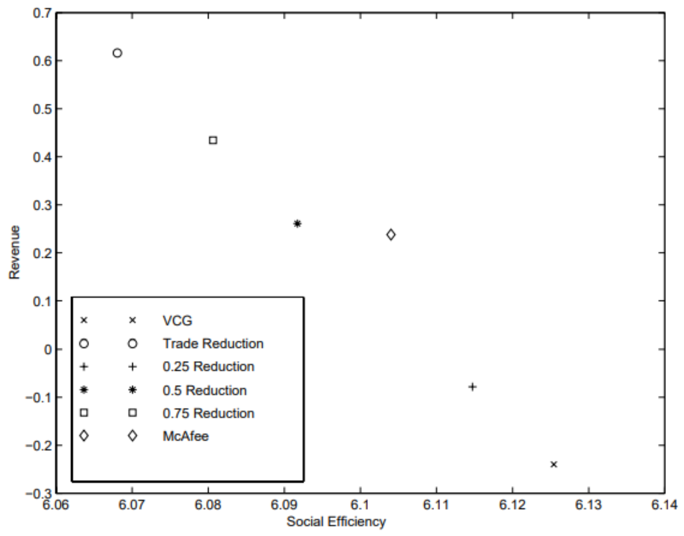


Fig. 4. Revenue/Efficiency tradeoff simulations of different mechanisms, using 25 buyers and 25 sellers [3].

2.3 Auctions applied to computer networks

As previously mentioned the use of auctions evolved with technology, being used in computer networks around the world, many stock markets of the modern age are digitized. In order to accomplish these digitized auctions successfully one must analyze the platforms they are held on. The way participants communicate and to who must be considered, as well as the software tools used for it as well as the constraints they might have.

Researchers are turning to auctions in a variety of studies of software markets to see if they bring an advantage to these markets.

2.3.1 P2P auction market for Transactive Energy systems

Khorasany et al. [15] propose an auction solution for transactive energy systems, held through peer-to-peer (P2P) networks, their decentralized nature makes having an auctioneer more difficult. In this case, the auction should have a strong budget balance, with no surplus or deficit.

In their proposed framework they have four layers, demonstrated in Figure 5. The decision making layer is where participants make their bids, there is a market data center which monitors network constraints which in turn is connected to Distribution System Operator. At the business layer energy related services are handled in an unregulated environment. The cyber layer contains smart meter, data centre and all communication infrastructures. Finally the component layer contains all the used hardware components distributed across the grid.

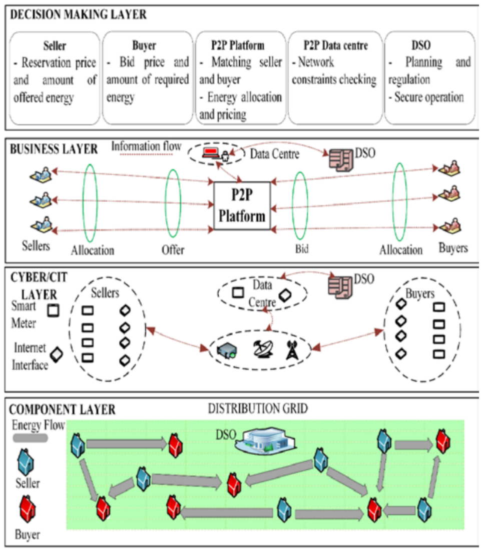
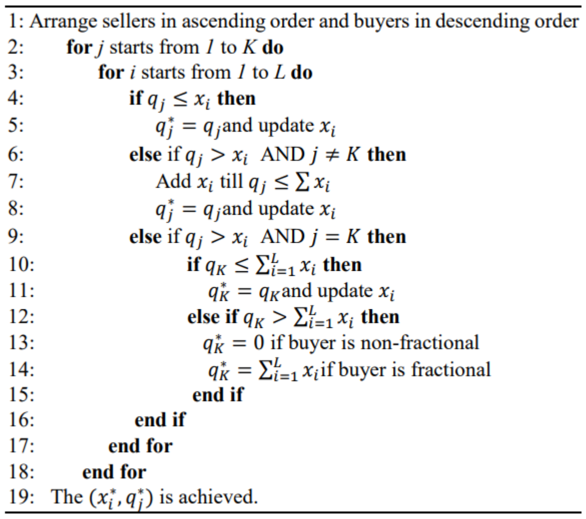


Fig. 5. Energy trading framework in distribution network [15].

In this framework, the network is divided in blocks, these blocks depend on the connections between participants, each auction is held per block. One participant can trade energy with multiple participants, there is not an one on one trade restriction, and provided they are in the same block, they do not pay any additional fees to transfer energy. For auction purposes different bids coming from the same participant counts as coming from a different participant, so it is easier to do a supply and demand curve of the bids.

There is a three step process to the auction. *Determination* of participants, *allocation* of energy shares between participants and finally *payment* discovery. The *allocation* phase is done through a "greedy" algorithm [2] and participants have the option to participate in a fractional or non-fractional manner, meaning, they can choose to trade a lower volume of energy than their bid if the full amount is not possible, as based on the fractional knapsack problem [12]. This is the main difference between already existing mechanisms, and the fractioned trade insures a better trade efficiency.



```
Arrange sellers in ascending order and buyers in descending order
    for $j$ starts from $1$ to $K$ do
        for $i$ starts from $1$ to $L$ do
            if $q_j \leq x_i$ then
                $q_j^* = q_j$ and update $x_i$
            else if $q_j > x_i$ AND $j \neq K$ then
                Add $x_i$ till $q_j \leq \sum x_i$
                $q_j^* = q_j$ and update $x_i$
            else if $q_j > x_i$ AND $j = K$ then
                if $q_K \leq \sum_{i=1}^{L} x_i$ then
                    $q_K^* = q_K$ and update $x_i$
                else if $q_K > \sum_{i=1}^{L} x_i$ then
                    $q_K^* = 0$ if buyer is non-fractional
                    $q_K^* = \sum_{i=1}^{L} x_i$ if buyer is fractional
                end if
            end if
        end for
    end for
The $(x_i^*, q_j^*)$ is achieved.
```

Fig. 6. Allocation algorithm from buyers' perspective [15].

There are two possible algorithms, one that will run from the buyers' perspective, shown in Figure 6 ,and another for the sellers', depending on the trade volume from each side. If the total demand for energy from the buyers' side is greater than the energy offered by the sellers' the allocation algorithm is run from the buyers' perspective and the last buyer will make the fractional decision; if the sellers' is greater, then the algorithm is run from the sellers' perspective and the last seller will make the decision. If they are the same, both algorithms will have the same result no matter which one is run.

For payment discovery a proposed mechanism is used, it is similar to average mechanism, but instead of setting the price for all trades as the average of the buyers' and sellers' l index bid and offer, the price is specific to each trade. The price will be set as the average of bid and offer of that specific trade, making the payments different across trades. The goal is to maximize the total revenue and cost saving (TRC) of all participants.

A simulation was created with 16 participants, 8 buyers and 8 sellers. Different payment mechanisms were used and compared. Some single auction payment methods recognized by the energy market were also tested, these were uniform price, Vickrey, Pay as bid, and Generalized second price auctions.

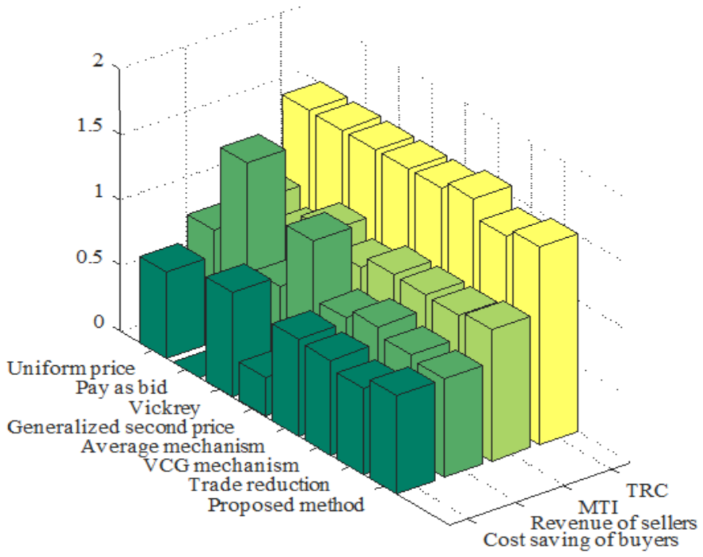


Fig. 7. TRC and MTI results of each auction type types [15].

Represented in Figure 7, the proposed mechanism shows the best results. In terms of TRC it is beaten by the VCG mechanism, but this mechanism is not viable in this market due to not having a strong balanced budget. The only compared double auction mechanism with strong balanced budget is the average mechanism and it has a Market Tendency Index (MTI) very close to 1, meaning that the trades are only very slightly skewed, but the proposed mechanism beats it by having a MTI of 1 with no favor to buyers nor sellers. Similarly to VCG, the trade reduction mechanism is not SBB, additionally it had a lower TRC due to having a lower trade volume.

The other payment methods are skewed to one side of participants. On the one hand, for Pay as bid, uniform price and the Generalized second price methods, the MTI has a value lower than 1, being more favorable to sellers. On the other hand, Vickrey's MTI value was higher than 1, being more favorable towards buyers.

This study shows that having a double auction mechanism in peer-to-peer markets can be beneficial, both to buyers and sellers as shown by the good MTI and TRC values of the proposed mechanism. However, it did not take into account some practical problems like network constraints.

2.3.2 AuctionWhisk

The previously mentioned FaaS payment opportunities might not be the most advantageous. As it was demonstrated previously, having an auction based approach to payment may not only be better for buyers seeking cloud resources, but also preferable to cloud providers as they can take advantage of higher demand.

AuctionWhisk [4] is a proposed auction based extension for OpenWhisk, using first-price sealed-bid or Vickrey as payment methods, but they are considered a surcharge and what really is evaluated are how the bids are handled.

AuctionWhisk is a solution for fog environments where clients send their requests to the edge nodes, these requests have two bids attached to their executable, one for storage of the executable and another for the actual execution, a processing bid.

The edge nodes try to handle their requests, but if they are overloaded and have no remaining space, they will delegate their requests to an intermediary node, similarly, should the intermediary nodes run out of capacity, they will delegate them to the cloud.

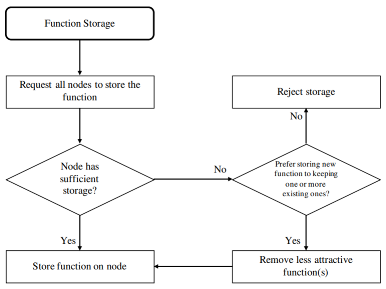

Function Storage
Request all nodes to store the function
Reject storage
No
Node has sufficient storage?
No
Prefer storing new function to keeping one or more existing ones?
Yes
Yes
Store function on node
Remove less attractive function(s)

Fig. 8. Deployment and storage process of a single function [4].

The determining factor of which requests get to stay on each node is their storage bid, Figure 8. Requests are ordered by their storage bid, when a node is out of capacity and receives further requests it will compare the already stored requests' storage bids with the incoming ones, keeping the ones with the greater value and offloading the others.

In order to check which of the stored requests is executed, the requests are ordered by their processing bids, highest first, and scheduled while compute capacity is left, requests that exceed this capacity are rejected and relayed to the next node. This process can be visualized in Figure 9.

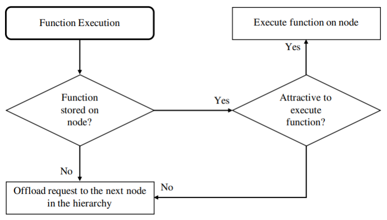

Function Execution
Execute function on node
Yes
Function stored on node?
Yes
Attractive to execute function?
No
No
Offload request to the next node in the hierarchy

Fig. 9. Function execution process for a single node and a single function execution. [4].

Bermbach et al. [4] identify four main challenges of creating an auction solution, which they tackle in the practical implementation of AuctionWhisk. To note that a simulation in Kotlin was also made before the practical implementation, but we will only talk about the practical implementation as it is the most relevant, since it should have closer results to a final commercial solution.

1. **Auctions Arrive in Batches**

   The way requests are chosen for execution is only viable if requests arrive in batches so the bids can be properly compared, in a more practical scenario this would not be the case, as requests would come in arbitrarily and independently. This would create a "first come first served" scenario, as requests would be run as soon as they arrived, provided there was enough capacity.
   In order to solve this challenge two solutions were implemented. In the first one, the controller creates a window of time, e.g. 1s, where it will not start executions, it will instead wait and as more requests arrive they are auctioned within this time window, the auction ending when the time window ends. This solution presents another practical problem, however; due to requests having to wait for the time window to end in order to be executed or offloaded, there is an additional latency of half the time window, on average, which is unacceptable for a request with an execution time that is not considerably larger than the time window.

The second solution escapes a little bit from the auction concept, a new decision manager component is introduced in the controller that saves statistical aggregates on bids encountered on a time window. With this information it creates a baseline value that incoming bids must beat in order to be accepted, those that cannot will be offloaded. Since this baseline value is constructed according to incoming request information, it will change as the nature of the requests change. For example, if this value is made using the average bid value of received requests, it will rise if the requests' bids start to go higher, likewise if they start to drop, this baseline will also drop. Although the constructed baseline is quick to adapt to changes there is still a small delay to the changes where there might be some lost revenue for the provider or risk of request offloading for some buyers. This solution is the one used in the practical experiments.

2. **Calculating storage space**

   Estimating the storage space that each function needs is not a trivial task, since their needs might be so different. Making a rule based on the number of functions, although easy, is not practical. The problem on storage is only exacerbated with the fact that most FaaS systems also store metadata that grows in size over time.
   The solution found was to set the per-function storage limit parameter of OpenWhisk to it's maximum value, 48MB per default, in this way all functions have for practical purposes the same storage size requirements, the maximum value. With this, a solution based on the number of functions in the node is used, when a request arrives, if the node's free space is larger than the maximum per-function storage limit, the node accepts it into the database. Otherwise, it will compare the lowest bid value's request with the new request's and offload the one with the lowest value, storing the other.
   This solution is not optimized, as there might be requests with a much lower storage necessity than the maximum. However, since there was already a need for a buffer due to meta data storage and due to function possible need of temporary extra storage during execution, this was considered to be an acceptable solution.

3. **Estimating Computational Costs**

   As it is for storage space, each function has a different need for computational power. In existing cloud FaaS deployment models the number of functions the node can support is estimated by dividing the node's available memory by the average of all requests' memory requisitions, this leads to an overbook of memory resources since it does not take into account the underlying needs of running a function, such as the VM's environment and other differences in memory usage.
   In addition to this metric an additional parameter to define time window length and number of parallel requests that can be accepted during this

time length is added to OpenhWhisk. With a bigger time window, some requests will queued for a short period of time, with a shorter one, more requests will be offloaded to the next node, even if there is still capacity in the node.
This solution enables for a more detailed management of requests and resourses, but it also increases the likelihood of breaking the isolation of function containers between parallel functions.

4. **Next node towards the cloud**

   Due to the dynamic nature of fog environments connections between nodes may change and with them the previous established routes. For a practical implementation to be viable, nodes need to know the next node towards the cloud and be resilient toward node failure and route changes. To achieve this it was suggested to store the full path to the cloud on every node, but this was was not implemented and it is assumed in the practical implementation that nodes already know the routing towards the cloud.
   One of the downsides of offloading requests is that there is a rare chance no node has the function stored in it, the request would be therefore continuously offloaded towards another node until getting a high-latency error message. This downside is acceptable due to its rarity and by not having to store the information of all functions on every node.

The experiments were done using three nodes, edge node, intermediary and cloud node. These nodes have higher available resources respectively and an artificial latency was introduced between nodes, seen in Figure 10.

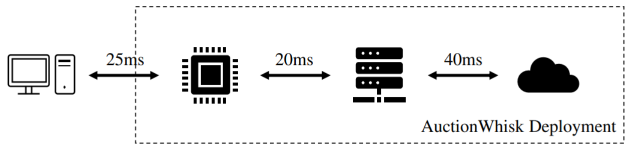


Fig. 10. AuctionWhisk deployment with set node latency. [4].

In the first experiment the request latency of AuctionWhisk was compared to one in a cloud-only deployment of OpenWhisk. As expected, as more requests are sent, the higher the latency becomes more apparent after certain thresholds of requests are passed. This is due to requests being offloaded to the next nodes when each node becomes overloaded, this can be seen in Figure 11 by the sudden slope rises followed by two "plateaus", which show how much each node tier can sustain before offloading to the next one. We can also see that AuctionWhisk handles requests at least as fast as the cloud-only deployment of OpenWhisk. Proving that it can efficiently take advantage of edge nodes which are closer.

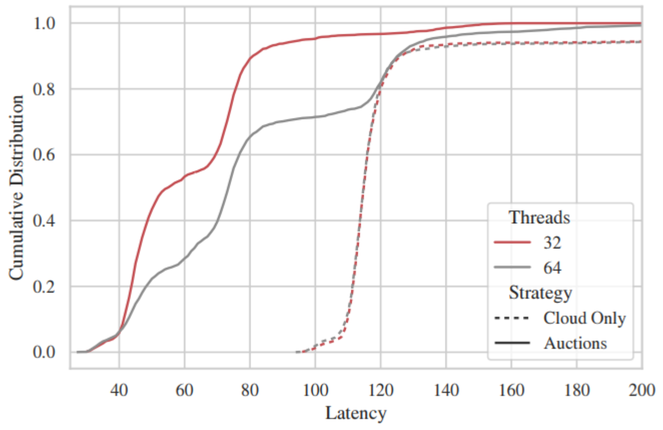


Fig. 11. CDF of AuctionWhisk vs cloud-only deployment of OpenWhisk for two different load levels. [4].

Another result of the experiments was that requests run closer to edge clouds pay a larger amount than the ones run closer to the cloud. This is expected, since the less desirable requests for a node, meaning the ones with lower bids, get offloaded to the next node. This makes an interesting situation where developers have the choice to pay more and have lower latency by having a higher chance their requests are run on edge nodes or pay less with added latency, having their requests run closer or in the cloud. This is true for both storage and processing bids.

Although AuctionWhisk provided great results, the work did not test the possibility of different auction mechanisms, specially the possibility of sellers entering the market as players as well in a double auction. In fact, it strayed away a little from the auction concept due to having the described metrics for offloading requests. It also lacks adaptability in terms of the bids since these do not change after deployment.

Other previous work on Auctions in distributed or edge computing scenarios has only targetted allocation of VMs and containers and not Function-as-a-Service, e.g. [11,16,7].

3 Proposed solution

In this section we present our proposed solution that utilizes the knowledge of the research in the previous section to bring a double auction solution for FaaS solutions at edge nodes.[8] This will be done with a new middleware node called StockWhisk.

[8] We identify the relevant issues of creating incentives to user participation in decentralized systems as an orthogonal aspect, e.g. [22,6].

3.1 Solution Design

In order to achieve better saving costs for developers seeking FaaS computing, while at the same time trying to maximize revenue for cloud providers, we propose a Double Auction extension for AuctionWhisk, and in tandem, OpenWhisk. We will call this solution StockWhisk

We present an additional middleware node between edge nodes and the client devices that will serve as an auctioneer, Figure 12. This node's purpose will be to conduct double auctions in order to allocate the client's FaaS functions to cloud providers' computing nodes. As it conducts double auctions, the middleware will receive bids not only from the clients, but also from the providers' nodes.

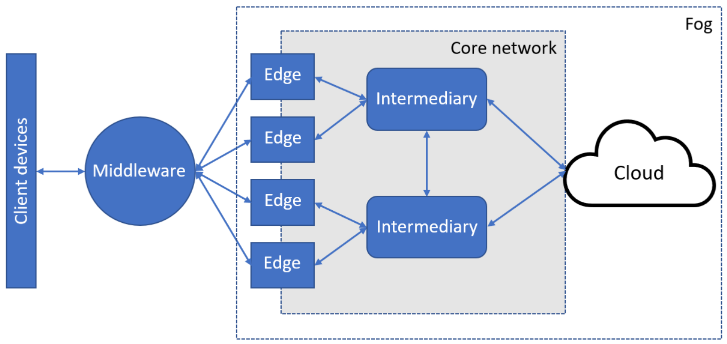


Fig. 12. Fog computing with the added middleware solution. Extended from [5].

As Bermbach et al. [4] have identified, there are some challenges to implementing an auction approach to FaaS.

We will take advantage of their solution for storage space calculation. We will assume that all functions require the per-function storage limit and handle node storage capacity based on the number of functions. Each provider's bid is correlated to a function slot, there is no risk of a provider getting overloaded since it will only send as many bids as its available space.

We will also keep AuctionWhisks method of estimating computational costs, the general method of dividing the node's available memory to the average memory requests of all functions will be complemented with the additional AuctionWhisk parameter of a time window with number of parallel requests allowed in it.

Whenever a function concludes execution and frees up space it will check if it can compute another function and send another bid to the middleware, becoming available to receive another request.

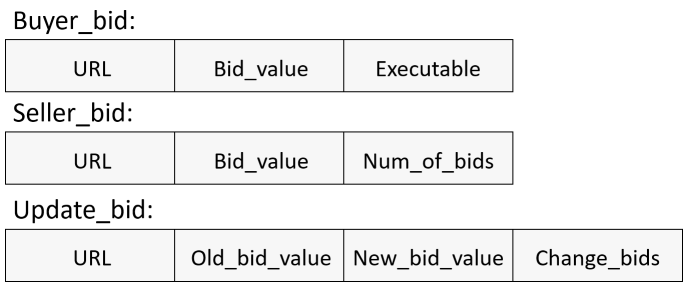
The middleware will handle load balancing, middleware nodes are static and are known to all nodes using StockWhisk, StockWhiks will also keep routing paths to the cloud. Therefore, the previous routing problem for AuctionWhisk is not present.

Since all connections pass through the middleware and computing nodes also need to bid, the middleware will have knowledge about all available edge computing power in its local group. Therefore, if a request cannot be matched with a provider then it is assumed there is no way for that request to be accepted on the edge, e.g. having an extremely low bid and is sent directly to cloud nodes where it will be treated with AuctionWhisk rules.

In order to try to save on memory costs for the middleware, we will let a node send a bid with an attached number of function slots, this way we avoid storing multiple bids from the same node.

This has the drawback of not letting nodes change their bid and not adapting to the market. To combat this, we will use an update function from the computing node's side which contains the old bid value or another variable[9], a new bid value and the number of bids to change, letting it change bid values at any time. This update message can be sent according to some metric set by the node operator, for example, if the node is over 50% capacity send an update function changing the lowest value bids.

After receiving an update function from a node, the middleware will separate the updated new value bids from the unchanged bids remaining with the old value. If the old value in the update function variable does not exist inside stored bids in the middleware for that node's URL, the middleware will not change anything. If the number of bids to change is greater than the number of existing bids with that value, the middleware will simply update them all.



Buyer_bid:

URL	Bid_value	Executable

Seller_bid:

URL	Bid_value	Num_of_bids

Update_bid:

URL	Old_bid_value	New_bid_value	Change_bids

Fig. 13. Contents of StockWhisk messages

3.2 Auction mechanism

Various double auction mechanisms will be tested. In order for a normal double auction to happen, we would need to have a time window where we can agglom-

[9] A variable that indicates change to the lowest stored bids for example

erate the clients' bids so we can sort them and create the supply and demand curve. The sellers' bids are easier, they still, but less, volatile, since during low demand times we can sort their bids and use them later. Nevertheless, the problem persists that the supply and demand curve cannot be generated without a time window delay which adds latency to the clients' requests.

We propose a solution similar to the one used by Khorasany et al. [15]. An average mechanism will be applied to each specific trade. From their results, this should have good results in terms of MTI and TRC, which we will test and evaluate against other auction types, further explained in the next section.

The middleware will sort the providers' bids as they come and save them sorted, then, when a client's bid comes, it will match it with the closest lower value from the providers' sorted bids and relay the client's message to the matched provider. The middleware will then remove the provider's bid in order to not send another request to it. Like mentioned above, after the provider has handled the buyer's request it can send another bid to the middleware where it will be inserted in the sorted group of seller's bids. The insertion of new provider bids in a sorted list and in tandem finding the best match for a client request can be easily solved using a slightly modified solution for a search insert position problem, which has a solution with time complexity of $O(log\ N)$. In the future, more complex algorithms can be studied to have a better time complexity to further decrease latency for requests.

4 Evaluation Methodology

In this section we present the metrics, Section 4.1, and the workloads, Section 4.2, that are going to be used to evaluate our proposed solution.

4.1 Metrics

We will consider a varied assortment of metrics from both a practical computational point of view, as well as an economic one.

- **CPU utilization in edge nodes** We will check the percentage of computing power utilized in the processing nodes to see how loaded they are.
- **Memory utilization in edge nodes** Similarly to the previous metric we will measure memory utilization to see how much edge nodes are occupied.
- **Average request payment** We will check the average payment of requests to see how prices fluctuate during execution.
- **Request travel latency** An important aspect of FaaS is the need for functions to start being executed fast. We will measure how long it takes for a request to go from the client node to the result returning to it, excluding the execution time.
- **Allocation time** Inside the total latency, we will specifically measure how long it takes for the middleware node to allocate a processing node to a request. With this we aim to check if the used mechanism and algorithm for allocation are appropriate.

- **Allocation success rate** On the one hand we will measure the percentage of processing nodes' bids that get a request paired to them. This is to measure edge node utilization. On the other hand, we will also check the percentage of requests that get allocated, even to cloud nodes. For both cases we want the rate to be as high as possible, meaning that no request is unfulfilled, if needed by using incremental approaches [25].
- **Scalability** Similarly to AuctionWhisk, we will progressively load the system with more requests and see how performance progresses.
- **TRC** We want total revenue and cost savings to be as high as possible, the higher this metric is the more advantageous the mechanism is economically for participants.
- **MTI** With this metric we measure if the market is skewed towards one side or the other, the closer to 1 it is the better, as a value of 1 means that neither clients nor providers are getting a favorable outcome over the other.

4.2 Workloads

Although request functions are assumed to have they the same executable size, their runtimes can be vastly different, as well as their resource utilization. Due to this we will employ a variety of workloads to test the system, the first five are also utilized to test *Photons* [9]. Experiments will be carried out in desktop and server machines, as well as in cloud deployments or simulations [14].

- **sleep(t) function** This function has extremely low resource utilization. With t set to a minimum we can set this workload as a baseline of comparison for the others.
- **REST API (HTTP request)** These function requests usually require minimal resources and have a short execution time.
- **File hashing** We will hash files fetched from OpenWhisk's CouchDB. High memory cost, but lower CPU cost.
- **Image Classification** Fetching of an image from CouchDB and then performing classification on it to simulate workloads of machine learning inference. This workload is very costly in terms of memory and CPU.
- **Video transformation** This is a very CPU and memory intensive workload. Each function processes a chunk of a video file, fetched from CouchDB individually.
- **Fibonacci sequencing** This is a CPU intensive workload, with low memory requirements.

5 Conclusion

In this work we presented a double auction solution for allocation of functions in FaaS using a new middleware node called StockWhisk. We extend the already existing AuctionWhisk [4] while adapting a double auction mechanism used for energy trading that produced favorable results [15]. With this work we have the goal of having advantageous prices for both clients and cloud providers when trading cloud resourses in FaaS while still maintaining low latency for requests. In our next work we will implement the proposed solution and evaluate it.

References

1. The Apache Software Foundation. OpenWhisk documentation. `https://openwhisk.apache.org/documentation.html`, accessed: 2023-01-13
2. Discrete-variable extremum problems. Oper. Res. 5(2), 266–288 (apr 1957), `https://doi.org/10.1287/opre.5.2.266`
3. Babaioff, M., Nisan, N.: Concurrent auctions across the supply chain. Journal of Artificial Intelligence Research 21, 595–629 (may 2004), `https://doi.org/10.1613%2Fjair.1316`
4. Bermbach, D., Bader, J., Hasenburg, J., Pfandzelter, T., Thamsen, L.: Auction-Whisk: Using an auction-inspired approach for function placement in serverless fog platforms. Software: Practice and Experience 52(5), 1143–1169 (dec 2021), `https://doi.org/10.1002%2Fspe.3058`
5. Bermbach, D., Pallas, F., Pérez, D.G., Plebani, P., Anderson, M., Kat, R., Tai, S.: A research perspective on fog computing. In: Braubach, L., Murillo, J.M., Kaviani, N., Lama, M., Burgueño, L., Moha, N., Oriol, M. (eds.) Service-Oriented Computing – ICSOC 2017 Workshops. pp. 198–210. Springer International Publishing, Cham (2018)
6. Daniel, E., Tschorsch, F.: Ipfs and friends: A qualitative comparison of next generation peer-to-peer data networks. IEEE Communications Surveys & Tutorials 24(1), 31–52 (2022)
7. Dias, D.P., Simão, J., Veiga, L.: RATEE - resource auction trading at edge environments. In: Andreolini, M., Marchetti, M., Avresky, D.R. (eds.) 20th IEEE International Symposium on Network Computing and Applications, NCA 2021, Boston, MA, USA, November 23-26, 2021. pp. 1–5. IEEE (2021), `https://doi.org/10.1109/NCA53618.2021.9685546`
8. Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., Chen, H.: Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 467–481. ASPLOS '20, Association for Computing Machinery, New York, NY, USA (2020), `https://doi.org/10.1145/3373376.3378512`
9. Dukic, V., Bruno, R., Singla, A., Alonso, G.: Photons: Lambdas on a diet. In: Proceedings of the 11th ACM Symposium on Cloud Computing. p. 45–59. SoCC '20, Association for Computing Machinery, New York, NY, USA (2020), `https://doi.org/10.1145/3419111.3421297`
10. Eismann, S., Scheuner, J., Eyk, E.v., Schwinger, M., Grohmann, J., Herbst, N., Abad, C.L., Iosup, A.: The state of serverless applications: Collection, characterization, and community consensus. IEEE Transactions on Software Engineering 48(10), 4152–4166 (2022)
11. Fonseca, A., Simão, J., Veiga, L.: Faircloud: Truthful cloud scheduling with continuous and combinatorial auctions. In: Panetto, H., Debruyne, C., Gaaloul, W., Papazoglou, M.P., Paschke, A., Ardagna, C.A., Meersman, R. (eds.) On the Move to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October

23-27, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10574, pp. 68–85. Springer (2017), https://doi.org/10.1007/978-3-319-69459-7_5
12. Goodrich, M., Tamassia, R.: Algorithm design: Foundations, analysis, and Internet examples (01 2002)
13. Kagel, J.H.: 7. Auctions: A Survey of Experimental Research, pp. 501–586. Princeton University Press, Princeton (1995), https://doi.org/10.1515/9780691213255-009
14. Kathiravelu, P., Veiga, L.: An adaptive distributed simulator for cloud and mapreduce algorithms and architectures. In: Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2014, London, United Kingdom, December 8-11, 2014. pp. 79–88. IEEE Computer Society (2014), https://doi.org/10.1109/UCC.2014.16
15. Khorasany, M., Mishra, Y., Ledwich, G.: Design of auction-based approach for market clearing in peer-to-peer market platform. The Journal of Engineering 2019 (07 2019)
16. Lavoie, E., Hendren, L., Desprez, F., Correia, M.: Genet: A quickly scalable fat-tree overlay for personal volunteer computing using webrtc. In: 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). pp. 117–126 (2019)
17. McAfee, R.: A dominant strategy double auction. Journal of Economic Theory 56(2), 434–450 (1992), https://www.sciencedirect.com/science/article/pii/002205319290091U
18. Morcillo, M.G.: Staging power and authority at roman auctions. Ancient Society 38, 153–181 (2008), http://www.jstor.org/stable/44080266
19. Myerson, R.B., Satterthwaite, M.A.: Efficient mechanisms for bilateral trading. Journal of Economic Theory 29(2), 265–281 (1983), https://www.sciencedirect.com/science/article/pii/0022053183900480
20. Patros, P., Spillner, J., Papadopoulos, A.V., Varghese, B., Rana, O., Dustdar, S.: Toward sustainable serverless computing. IEEE Internet Computing 25(6), 42–50 (2021)
21. Patten, R.W.: Tatworth candle auction. Folklore 81(2), 132–135 (1970), http://www.jstor.org/stable/1258945
22. Rodrigues, P.D., Ribeiro, C., Veiga, L.: Incentive mechanisms in peer-to-peer networks. In: 24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings. pp. 1–8. IEEE (2010), https://doi.org/10.1109/IPDPSW.2010.5470860
23. Rothkopf, M.: Thirteen reasons why the vickrey-clarke-groves process is not practical. Operations Research 55, 191–197 (04 2007)
24. Rustichini, A., Satterthwaite, M.A., Williams, S.R.: Convergence to efficiency in a simple market with incomplete information. Econometrica 62(5), 1041–1063 (1994), http://www.jstor.org/stable/2951506
25. Simão, J., Veiga, L.: Qoe-jvm: An adaptive and resource-aware java runtime for cloud computing. In: Meersman, R., Panetto, H., Dillon, T., Rinderle-Ma, S., Dadam, P., Zhou, X., Pearson, S., Ferscha, A., Bergamaschi, S., Cruz, I.F. (eds.) On the Move to Meaningful Internet Systems: OTM 2012. pp. 566–583. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
26. Stafford, A., Toosi, F.G., Mjeda, A.: Cost-aware migration to functions-as-a-service architecture. In: European Conference on Software Architecture (2021)
27. Varshney, P., Simmhan, Y.: Demystifying fog computing: Characterizing architectures, applications and abstractions. In: 2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC). pp. 115–124 (2017)

A Schedule

Tasks	**Duration**
Solution Development	105 days
Solution Evaluation	18 days
Dissertation writing	85 days