Self-adapting Applications Based on QA Requirements in the Cloud Using Market-Based Heuristics

Vivek Nallur and Rami Bahsoon

Conference : ServiceWave 2010

Lecture Notes in Computer Science, 2010, Volume 6481, Towards a Service-Based Internet, Pages 51-62

# Self-Adapting Applications Based on QA Requirements in the Cloud Using Market-Based Heuristics

Vivek Nallur and Rami Bahsoon

University of Birmingham,
Birmingham, B15 2TT, United Kingdom
{v.nallur,r.bahsoon}@cs.bham.ac.uk

**Abstract.** There are several situations where applications in the cloud need to self-manage their quality attributes (QA). We posit that self-adaptation can be achieved through a market-based approach and describe a marketplace for web-services. We simulate agents trading web-services on behalf of self-managing applications and demonstrate that such a mechanism leads to a good allocation of web-services to applications, even when applications dynamically change their QA requirements. We conclude with a discussion on evaluating this mechanism of self-adaptation, with regards to scalability in the cloud.

**Keywords:** self-adaptation, web-services, quality attributes, cloud

## 1 Introduction

Many users experience a need to change the *quality attributes*(QA) exhibited by their application. By QA, we mean non-functional aspects of an application like performance, security etc. The need for change can occur due to dynamic changes in the users' requirements and the environment in which the system operates. An unexpected time constraint, for instance, can require an organization to increase the performance of its service provision. Governmental regulation can require an organization to want to increase the audit trail or the security level, whilst not changing the functionality of its application. Spikes and troughs in business demand can spur a change in storage and performance requirements. These examples (and requirements) are fairly commonplace, and yet mechanisms to achieve them are not very clear. It seems obvious (and desirable) that applications should adapt themselves to these changing requirements, specially QA requirements. Users find it difficult to understand, why a change that does not affect the functionality of an application, but merely its security requirements is difficult to achieve. However, in an arbitrary case, making an application adapt is a difficult process. It has to be architected in such a manner that adaptation is possible. This means that it should be able to:

1. Dynamically identify changed requirements, which will necessitate runtime adaptation;

2. Initiate the search for new services, which better address the changed requirements
3. Substitute old web-services for new web-services

In this paper, we look at the second of those three steps. Web-Service composition allows an application to switch one web-service for another, at runtime. However, searching for new services, even ones that are functionally the same but exhibit different QA levels, is a difficult process. This is exacerbated when the application is resident on a cloud and has a plethora of web-services to choose from. Applications can typically find web-services, based on their functionality, using WSDL and UDDI registries. However, the number of available web-services to chose from, is typically large, and the number of parameters on which to match, adds to the complexity of choosing. Most of the current research on dynamic web-service composition focuses on ascertaining the best match between the QA demanded by the application and the QA advertised by the services. The best match, however, is difficult if not impossible to find in a reasonable time, given the dynamic nature and scalability of the cloud. Given time constraints, our approach advocates getting a *good match*, as fast as possible. A good match is one where the chosen service meets the application's minimum QA requirements, but is not necessarily the best possible choice.
The problem can be illustrated in more detail by an example from the domain of rendering 3D graphics(section 3). In [9], we proposed using a self-adapting mechanism, viz *Market-Based Control* for finding a fast, valid solution. We propose that principles and tools of economics be used as a heuristic to explore the search space. This paper is structured as follows: we first review other approaches in self-adaptation (section 2), present a motivating example for our approach (section 3), propose the design of a market (section 4), and report on the scalability of the approach. We conclude by highlighting the strengths and weaknesses of our approach (section 5) and discuss mechanisms that could improve it further.

## 2 Related Work

There have been a plethora of attempts at creating self-managing architectures. These attempts can be classified as approaching the problem from the following perspectives:

- Map system to ADL, dynamically change ADL, check for constraints, transform ADL to executable code [6].
- Create a framework for specifying types of adaptation possible using constraints and tactics thus ensuring 'good adaptation' [4].
- Using formalized methods like SAM [12], middleware optimizations [11] and control theory [8]

However, all of these approaches envision a closed-loop system. That is, all of them assume that

1. The entire state of the application and the resources available for adaptation are known/visible to the management component.

2. The adaptation ordered by the management component is carried out in full, never pre-empted or disobeyed.
3. The management component gets full feedback on the entire system.

However, in the context of the cloud, with applications being composed of services resident on several (possibly different) clouds, the assumptions above do not hold true. It is unreasonable to expect that third-party services would obey adaption commands, given by an application's self-management component. Therefore, instead of making third-party services adapt, self-adaptive applications simply choose services, that meet their changed requirements. That is, they substitute old services providing a given QA, with new ones providing better-matched QA. *Zeng et al.* [14] were amongst the first to look at non-functional attributes while composing web-services. They proposed a middleware-based approach where candidate execution plans of a service-based application are evaluated and the optimal one is chosen. The number of execution plans increase exponentially with the number of candidate services and linear programming is used to select an optimal plan. *Anselmi et al.* [1] use a mixed integer programming approach to look at varying QA profiles and long-lived processes. Increasing the number of candidate services however means an exponential increase in search time, due to the inherent nature of linear programming. *Canfora et al.* [3] have used genetic algorithms to replan composite services dynamically and also introduce the idea of a separate triggering algorithm to detect the need for re-planning. While much better than the time taken for an exact solution, GAs are still not scalable enough to deal with hundreds or thousands of services in real-time. *Yu et al.* [13] propose heuristics to come up with inexact, yet good, solutions. They propose two approaches to service composition, a combinatorial model and a graph model. The heuristic for the graph model (MSCP-K) is exponential in its complexity, while the one for the combinatorial model (WS_HEU) is polynomial.

The main difference between our approach and these algorithms, lies in the context of the problem that is being solved. We are looking at the **simultaneous adaptation** of hundreds of applications in the cloud, whereas the research above looks at optimal selection of web-services for a single application. The cloud, by definition, offers elasticity of resources and immense scale. Any solution for the cloud therefore, needs to dynamic and fast, and not necessarily optimal. Adaptation also emphasizes timeliness and constraint satisfaction, which is easier to achieve using decentralized approaches. Most middleware type approaches (as outlined above) are centralized in nature. A marketplace, on the other hand, is a decentralized solution, which adapts quickly to changes in conditions. A double auction is known to be highly allocation-efficent [7], thus meeting the criteria of timeliness and goodness of adaptation within constraints. Also, the market-based approach is resilient to failures of individual traders or even marketplaces. In the event of one marketplace failing, the buyers and sellers simply move to another market.

## 3 Rendering 3D graphics

Consider a company that specializes in providing 3D rendering services. To save on capital expenditure, it creates an application that lives on the cloud and uses web-services to do the actual computation. This allows it to scale up and scale down the number of computational nodes, depending on the size of the job. However as business improves, the company implements a mechanism for automated submission of jobs, with differentiated rates for jobs with different priorities, requirements and even algorithms. For instance, a particular job may require texture and fluid modelling with a deadline of 30 hours, while another might require hair modelling with a tighter deadline of 6 hours. Yet another job might require ray-tracing with a deadline of 20 hours. Also, any of these jobs might have dynamically changing deadlines. This means that the application would need to change its *performance QA* dynamically. Depending on the kind of data and algorithm being used, the application's storage requirements would change as well. For certain tasks, the application might require large amounts of slow but reliable storage, while others might require smaller amounts of higher-speed storage. This company would like to create an application, that self-managed its QA, depending on the business input that was provided, on a per-task basis. Each of these self-management decisions are constrained by the cost of change. Now consider a multiplicity of such companies, each with several such applications, present on a cloud that provides web-services for computation, storage as well as implementations of sophisticated graphics algorithms. Each of the providing web-services have differing levels of QA as well as differing prices. Matching a dynamically changing list of buyers and sellers, each with their own QA levels and prices, is an important and difficult task for a cloud provider. It is important because, if a large number of buyers do not get the QA levels they desire, within their budget, they will leave for other clouds that **do** provide a better matching mechanism. It is also difficult because the optimal assignment at one time instance could be different from the optimal assignment at the next time instance. *Aradgna et al.* have shown in [2] that assignment of web-services when QA levels are static is equivalent to a Multiple-Choice Multi-Dimensional Knapsack Problem, which is NP-hard. Thus, the problem of assignment, when QA levels change dynamically, is also NP-hard. We contend therefore, that cloud implementations should provide a mechanism that solves this problem. Ideally it should be a self-managing approach that accounts for changes in operating environment and user preferences.

## 4 Description of Market

We describe a marketplace where multiple applications try to self-manage, in order to dynamically achieve their targetted QA. Each application has a private utility function with regard to certain fixed QAs. From utility theory, we use a von Neumann-Morgenstern utility function [10], that maps a QA($\omega$) to a real number, $u_\omega : X_\omega \longrightarrow \mathbb{R}$, which is normalized to$[0, 1]$. Each application is

composed of several services that contribute to the QA of the application. In the marketplace, each of these services is bought and sold by a trading agent. The agents trading on behalf of the application (the buyers) do not know the application's private utility function and attempt to approximate it. The only information that flows from the application to the agents, upon the completion of a trade, is a penalty or a reward. This enables the agent to work out two pieces of information: the direction it should go and the magnitude of that change. Although this meagre information flow makes the task of approximation harder, we think that it is more realistic since application architects would concentrate on creating the right composition of web-service functionality while third-party trading agents would concentrate on buying in the marketplace. There is no overriding need for trust amongst these two entities. In the motivating example above, the application for rendering graphics would be buying services that provide CPU, storage and algorithmic functionality. Each of these three services would be managed by a buying agent, that monitors the required QA, assesses the available resources and trades in the market. Enterprises that sell CPU time, storage space and implementations of graphics algorithms would be the selling agents in their respective markets. The market is a virtual auction house, created in a cloud, to whom the buyers and sellers would submit their trading requests.

We populate several markets with buying agents and selling agents. Each market describes a certain service class ($s_x$). A service class ($s_x$) contains services that are fundamentally identical in terms of functionality. The only differentiating factor, amongst the various services in a class, are the QA that each one exhibits. Thus, storage services belong to one service class, while services offering CPU time would belong to another service class, and so on. All the buying and selling agents in this market, trade services that deliver $s_x$, where $s_x \in S_x$. The number of markets is at least equal to $\mid S \mid$, with each market selling a distinct $s_x$. In other words, we assume that there exists a market for every $s_x$, that is needed by an application.

*Buyer:* The Buyer is a trading agent that buys a service for an Application, *i.e.,* it trades in a market for a specific service class $s_x$. Each service, available in $s_x$, exhibits the same QAs ($\omega \in QA$). Hence, if an application has $K$ QAs that it is concerned about, then the QAs that it gets for each of the $S_x$ that it buys is:

$$\Omega^{S_x} = \langle \omega_1^{s_x}, \omega_2^{s_x}, \omega_3^{s_x}, \ldots, \omega_K^{s_x} \rangle \tag{1}$$

The amount that the buyer is prepared to pay is called the *bid price* and this is necessarily *less than or equal to* the $B_{s_x}$, where $B_{s_x}$ is the budget available with the buyer. The combination of $\Omega$ demanded and the *bid price* is called a *Bid*.

In its attempt to reach the Application's QA requirement, the Buyer uses a combination of *explore* and *exploit* strategies. When trading begins, the Buyer randomly chooses an $\Omega$ (explore) and *bid price*. Based on whether it trades successfully and how close its guess was to the private QA, the Application provides with feedback in the form of how close the buyer is, to the private QA.

Based on this feedback, the Buyer either continues with the previously used strategy (exploit) with slight changes in price or changes strategies (explore). The trading strategy is essentially a combination of reinforcement learning and ZIP [5]. We expect the simultaneous use of this, by large number of traders, to result in an efficient search for a good assignment.

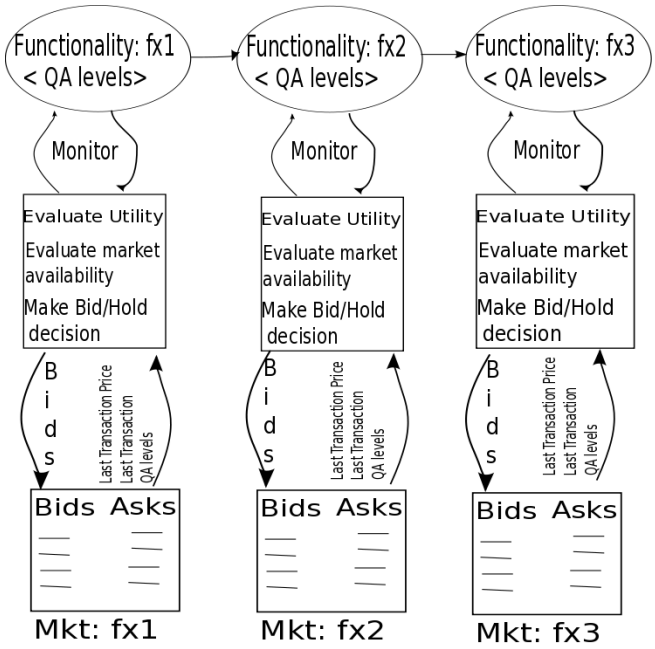



**Fig. 1.** Decentralized MAPE Loop; each service, in the application, self-adapts

*Seller:* Each seller is a trading agent, selling a web-service that exhibits the QA required in (1). The degree to which each QA($\omega$) is exhibited in each $s_x$ being sold, is dependent on the technological and economic cost of providing it. Hence, if the cost of providing $s_x$ with $\Omega = \langle 0.5, 0.6 \rangle$ is low, then there will be many sellers providing $s_x$ with a low *ask price*. Conversely, if the cost of providing $s_x$ with $\Omega = \langle 0.8, 0.9 \rangle$ is high, then the *ask price* will be high. An individual seller's *ask price* can be higher or lower based on other factors like number of sellers in the market, the selling strategy etc., but we mandate that the *ask price* is always *greater than or equal to* the cost. The combination of $\Omega$ and *ask price* is called the *Ask*. The obvious question that arises, is that of truth-telling by Seller. That is, how does the buyer know that the Seller is not mis-representing the $\Omega$ of her web-service? We currently assume that a Seller does tell the truth about $\Omega$ of her web-service and that a Market has mechanisms to penalize Sellers that do not.

*Market:* A market is a set of buyers and sellers, all interested in the same service class $s_x$. The factor differentiating the traders are:

- $\Omega$**:** The combination of $\langle \omega_1, \omega_2, \ldots \omega k \rangle$
- ***Price:*** Refers to the *bid price* and *ask price*. The buyers will not pay more than their respective *bid price* and the sellers will not accept a transaction lower than their respective *ask price*.

The mechanism of finding a matching buyer-and-seller is the *continuous double auction*(CDA). A CDA works by accepting offers from both buyers and sellers. It maintains an orderbook containing both, the *bids* from the buyers and the *asks* from the sellers. The bids are held in descending order of price, while the asks are held in ascending order, *i.e.*, buyers willing to pay a high price and sellers willing to accept a lower price are more likely to trade. When a new *bid* comes in, the offer is evaluated against all the existing *asks* in the book and a transaction is conducted when the price demanded by the *ask* is lower than the price the *bid* is willing to pay **and** all the QA attributes in $\Omega$ of the ask are *greater than or equal to* all the QA attributes in the $\Omega$ of the *bid*. Thus, a transaction always meets a buyer's minimum QA constraints. Each transaction generates a corresponding *Service Level Agreement*(SLA), which sets out the QA levels that will be available for $x$ invocations of a particular web-service. After a transaction, the corresponding *bid* and *ask* are cleared from the orderbook. Since this procedure is carried out for every offer (*bid/ask*) that enters the market, the only bids and asks that remain on the orderbook are those that haven't been matched. It has been shown that even when buyers and sellers have Zero-Intelligence, the structure of the market allows for a high degree of allocative efficiency [7]. A market is said to be allocatively efficient, if it produces an optimal allocation of resources. That is, in our case, the optimum assignment of sellers to buyers, based on multiple criteria of QA and price. Obviously, the market is not guaranteed to produce a completely efficient allocation, but even a high degree of efficiency is acceptable to us.

*Applications:* The Application is a composition of buyers from different markets. In our example, it would be the agent performing the orchestration of the CPU service, the storage service and the graphics service. An application is composed of at most $\mid (S_x) \mid$ buyers with a privately known $\Omega_{s_x}$ for each buyer. The buyer gets a certain budget $(B_{s_x})$ from the Application for each round of trading. The total amount of budget $B$ represents the application's global constraint on resources available for adaptation. After each round of trading, depending on the $\Omega$ obtained by the buyer, the Application has procured a total QA that is given by:

$$\forall s_x \in S_x, \sum \Omega^{s_x} \tag{2}$$

Buying and using a web-service involves many costs, and these must be compared against the projected utility gain to decide whether it is worthwhile to switch. These may be enumerated as:

– *Buying Cost:* The price of purchasing the web-service for $n$ calls ($p_{s_x}$)
– *Transaction Cost:* The amount to be paid to the market, for making the transaction ($t_{s_x}$)
– *Switching Cost:* The amount of penalty to be paid, for breaking the contract with the current web-service ($sw_{s_x}$)

Thus, the total cost that the application must consider is:

$$TC_{s_x} = p_{s_x} + t_{s_x} + s_{s_x} \tag{3}$$

We assume that, for every application there exists a function that maps $TC_{s_x}$ to an equivalent $\Omega^{s_x}$. The application could easily buy the best possible web-service(s) available, if it was prepared to spend an infinite amount of money. However, in reality, every application is constrained by a budget($B$) that it is willing to spend. Allocating $M$ amongst the functionalities that it is buying ($s_x \in F_x$) is a matter of strategy and/or the relative importance of each $s_x$.

$$\forall s_x \in S_x, \sum B_{s_x} = M \tag{4}$$

After every round, the application compares the privately known $\Omega_{s_x}$ with the $\Omega^b_{s_x}$ and determines whether to punish/reward the buyer. From this piece of information, the buyer can only guess whether it needs to improve the $\Omega^b_{s_x}$ that it obtained or the price that it paid. Once the $\Omega^b_{s_x}$ is close enough to the private $\Omega_{s_x}$, then the application refrains from trading any further, until it changes the privately known $\Omega_{s_x}$. Changing of the private $\Omega_{s_x}$ simulates the change in the environment, which has prompted adaptation by the application. The buyers have to now re-approximate the private $\Omega_{s_x}$.

*Trigger for Adaptation:* There are two possible scenarios for adaptation, continuous and criteria-based. In *continuous* adaptation, the application never really leaves the market, even after the Buyer agents have successfully fulfilled its QA requirement. It continues to employ Buyer agents in the hope that the buyer agent might procure a web-service with comparable QA, but at a lower cost. In *criteria-based* adaptation, a separate monitoring agent watches the Application's utility function as well as its budget. Any change in these, beyond a threshold, would trigger the monitoring agent to re-employ Buyer agents. This scenario also allows a human to enter the adaptation loop, by manually triggering the Buyer agents. We opted to simulate the continuous adaptation scenario, since we're interested in observing a long-lived, large system where human intervention might not be possible.

## 5 Evaluation

We intend to find out whether an application, following rules outlined above, could achieve the overall QA that it expects from its constituent web-services, keeping in mind the fact that the buying agents responsible for the individual

services, are not aware of the QA levels expected by the application. They merely attempt to approximate it, based on the feedback given by the application. The evaluation of the market-based approach is particularly hard, since it is a probabilistic approach. After every trading round, the *bid price* and *ask price* of every buyer and seller changes. This changes the optimal assignment, for every trading round. Hence, we do not consider whether the mechanism achieves optimal assignment, rather only whether it results in achievement of the Application's QA levels. In our simulation, each application composes 5 web-services to function. For each of these web-services, it has a private, different QA expectation. At the end of each trading round, the application examines the QA achieved by each buying agent. For each agent, it compares the private QA level with the achieved QA level, using a cosine similarity function. The cosine similarity compares two vectors for similarity and returns a result between 0 and 1(1 denoting an exact match). The combined similarity score is reflective of the degree to which the application's QA needs were satisfied. The theoretical maximum score, it could ever achieve with 5 web-services, is therefore 5.0.

### 5.1 Adaptation Level

The following graphs show simulations under three different market conditions. All values are the mean value of 30 simulations, each simulation consisting of 300 rounds of trading. The band (shaded area) around the line are the standard deviations. The line marked as 'ideal score'represents the theoretical maximum level of QA that the Application privately wants to have. The QA values for various services in the market were generated from a gaussian distribution. All simulations were done a dual-core P4 class machine with 2GB of RAM.

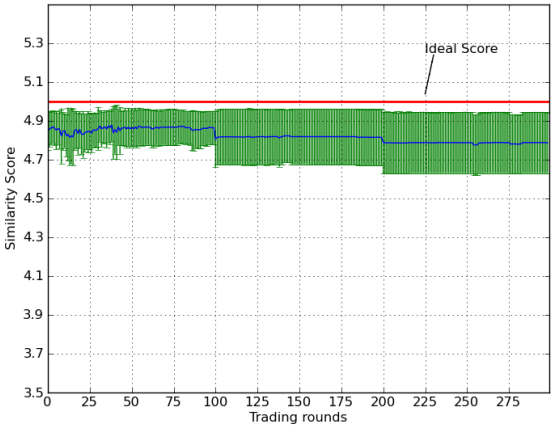
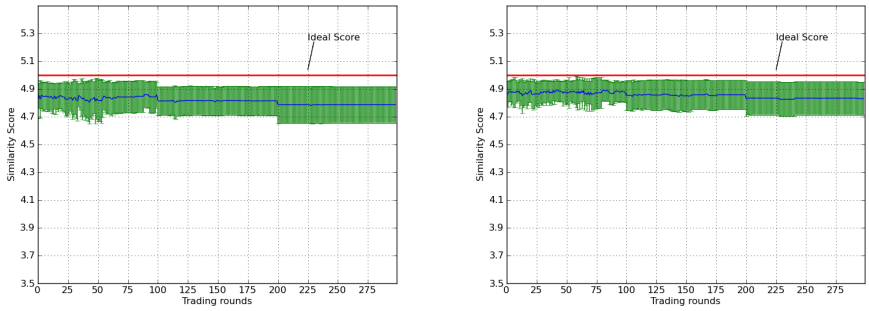



**Fig. 2.** Self-adaptation in the presence of equal demand and supply




The figure above shows markets with excess supply (left) and excess demand (right). Notice that in the figure with extra demand (right), the line showing the self-adaptation squiggles around for slightly longer than the other two figures (left and above). But, this is in line with intuition, where, the application finds it more difficult to get the QA that it demands, since there are a lot of buyers in the market demanding the same QA. It is interesting to note that even with simple agents, the application, as a whole, is able to reach very close to its ideal level of QA.

### 5.2 Scaling

One of the central themes of this paper has been scaling to the level of the cloud, which might contain hundreds or even thousands of services. We ran our simulations with increasing number of service candidates (in each market), and we found that the auction mechanism scales gracefully(Fig.3 ). As the number of service candidates (the number of buyers and sellers) increases from 100 to 1600, the time taken to reach an assignment rises in a near-linear fashion.

### 5.3 Strengths

We see in all cases, that the mechanism allows an application to reach very close to its desired level of QA. The nature of the mechanism is that it is highly distributed and asynchronous. This allows for a highly parallelizable implementation, and hence is very scalable. The buying agents, that trade on behalf of the application, are very simple agents with a minimal learning mechanism. Also, the concept of an auction is very simple to understand and implement. All the entities in the mechanism are simple and it is their interaction that makes it richer.

### 5.4 Limitation

Our implementation currently does not allow for more than one global constraint. By global constraint, we mean QA levels or other dimensions that the application has to meet, as a whole. Since, each market addresses one service class only, the

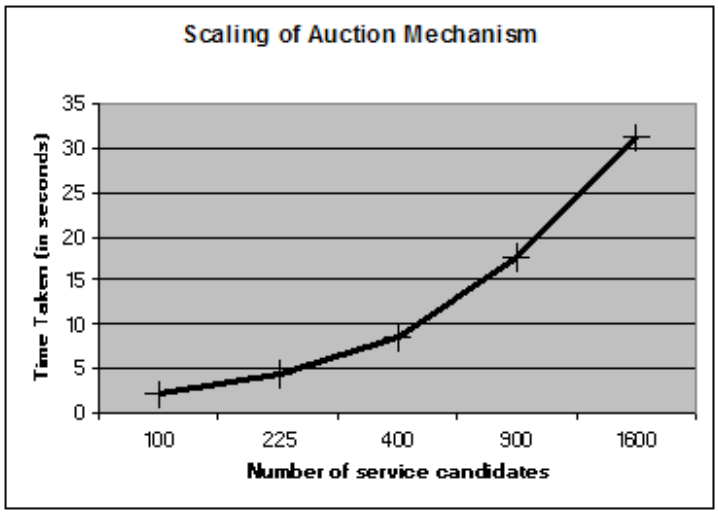



**Fig. 3.** Time taken by auction mechanism to perform matching

QA constraints for each individual service are met. However, the collective QA generated by the composition of services is not subjected to any constraint, other than price. This, however, is a limitation of our current implementation, and not of the approach. It is easy to envision a more complex Application entity that tests web-services, for their contribution to the total utility, before accepting it for a longer term contract.

The market-based approach does not currently reason about the time taken for adaptation. In cases where the timeliness of adaptation is more important than the optimality of adaptation, this could be an issue.

## 6 Conclusion and Future Work

We consider that the Market-based approach to self-adaptation is a promising one. Since the entities in the market are inherently distributed and asynchronous, the market-based approach lends itself to highly parallelizable implementations. However more formal work needs to be done to prove its scalability. There is little literature on assessing the quality of a self-managing architecture. As future work, we will be looking at measures to assess the quality of the solution found by the Market-based approach as compared to other search based approaches. We aim to complement our approach with sensitivity analysis, where we analyze the impact of changing individual QA of a given service, on the entire application's utility.

## References


1. Anselmi, J., Ardagna, D., Cremonesi, P.: A qos-based selection approach of autonomic grid services. In: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches. pp. 1–8. SOCP '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1272457.1272458
2. Ardagna, D., Pernici, B.: Global and local qos constraints guarantee in web service selection. In: ICWS '05: Proceedings of the IEEE International Conference on Web Services. pp. 805–806. IEEE Computer Society, Washington, DC, USA (2005)
3. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: Qos-aware replanning of composite web services. In: ICWS '05: Proceedings of the IEEE International Conference on Web Services. pp. 121–129. IEEE Computer Society, Washington, DC, USA (2005)
4. Cheng, S., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems. ACM, Shanghai, China (2006), http://dx.doi.org/10.1145/1137677.1137679
5. Cliff, D., Bruten, J.: Less than human: Simple adaptive trading agents for cda markets. Tech. rep., Hewlett-Packard (1997), http://www.hpl.hp.com/agents/papers/less_than_human.pdf
6. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: Towards architecture-based self-healing systems. In: WOSS '02: Proceedings of the first workshop on Self-healing systems. pp. 21–26. ACM Press, New York, NY, USA (2002), http://dx.doi.org/10.1145/582128.582133
7. Gode, D.K., Sunder, S.: Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality. The Journal of Political Economy 101(1), 119–137 (1993), http://www.jstor.org/stable/2138676
8. Hellerstein, J.: Engineering Self-Organizing Systems, p. 1 (2007), http://dx.doi.org/10.1007/978-3-540-74917-2_1
9. Nallur, V., Bahsoon, R., Yao, X.: Self-optimizing architecture for ensuring quality attributes in the cloud. In: Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2009), Cambridge, UK, September 14-17 (2009)
10. von Neumann, J., Morgenstern, O.: Theory of Games and Economic Behavior. Princeton University Press, 3rd edn. (January 1953)
11. Trofin, M., Murphy, J.: A Self-Optimizing container design for enterprise java beans applications. In: In Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004 (2003), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.2979
12. Wang, J., Guo, C., Liu, F.: Self-healing based software architecture modeling and analysis through a case study. In: Networking, Sensing and Control, 2005. Proceedings. 2005 IEEE. pp. 873–877 (2005), http://dx.doi.org/10.1109/ICNSC.2005.1461307
13. Yu, T., Zhang, Y., Lin, K.J.: Efficient algorithms for web services selection with end-to-end qos constraints. ACM Trans. Web 1(1), 6 (2007)
14. Zeng, L., Benatallah, B., H.H. Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: Qos-aware middleware for web services composition. IEEE Trans. Softw. Eng. 30(5), 311–327 (2004)