Using Collective Intelligence as a Self-Adaptation Mechanism in the Cloud

Technical Report: CSR-10-14

Vivek Nallur and Rami Bahsoon

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

Abstract. Self-Adaptation has traditionally been defined to work on closed-loop systems. However, in ultra-large scale distributed systems, inserting self-adaptive features via the closed-loop framework is prone to error and brittleness. We advocate a collective-intelligence based approach to self-adaptation for large, distributed systems. We use double-auctions as a mechanism to self-optimize service-based applications in the cloud. We show that the global utility obtained by self-adapting service-based applications using double-auctions is very high and that the mechanism is robust in the presence of skewed availablility of web-services. We conclude by comparing it with other adaptation mechanisms and evaluating their advantages and disadvantages.

1 Introduction

We live in a world where software applications are accessed by users from many different perspectives. These varied perspectives may come about due to differences in interface(e.g., desktop, cellphone), bandwidth (e.g. internal LAN, modem) or role(e.g., sysadmin, casual-user) etc. These perspectives lead to user-expectations about both, functional and non-functional aspects of the system. By non-functional aspects of an application, we mean quality attributes (QA) like performance, security, usability etc. While the perspectives themselves can be catered to, during the design process, the need for adaptation *within* these perspectives can occur due to dynamic changes in the users' requirements and the environment in which the system operates. These are typically QA adaptations. An unexpected time constraint, for instance, can cause an organization to want to increase the performance of its software. Governmental regulation can cause 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. 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 performance requirements is difficult to achieve. However, in an

arbitrary case, making an application adapt is a difficult process. The adapation process, in general, consists of the following steps:

1. Identify the adaptable artifact to be changed
2. Identify the change to be made
3. Effect the particular change on the adaptable artifact

An example of an adaptable artifact is a web-service that can be substituted dynamically, to achieve user requirements. Service-Oriented Architecture is acknowledged to be the best fit for developing adaptive applications that can flexibly aggregate individual services, to meet its functional and QA requirements [8]. By substituting an under-performing service for one that provides better suitability to requirements, an application is able to adapt to changing QA demands. Web-Service composition allows an application to switch one web-service for another, at runtime. However searching for new parts, even ones that are functionally identical, 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 chose from. 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. To make matters worse, every application will have some constraints in the form of resources that it can expend for the purpose of adaptation. From the perspective of the cloud provider, resident applications will want to change their constituent web-services and it is in a unique position to enable this change. The more applications find themselves satisfied with the recommended web-services, the more cloud gains in terms of reputation. Hence, from a cloud provider's perspective, the problem is one of enabling as many applications to self-adapt as possible, *i.e.*, self-adaptation at the collective level.That is, to ensure that the maximum number of applications are able to meet their adaptation objective.
In this context, we use market-based control, specifically auctions as the mechanism to chose which services to aggregate, in order to achieve the adaptation goal. In particular, the adaptation goal that we seek to achieve is to allow a multitude of applications to dynamically change their QA requirement and find web-services that meet those requirements. We use Collective Intelligence (COIN), as detailed by *Wolpert and Tumer* [30], as the guiding philosophical framework. We use a multiplicity of simple agents, unaware of the overall goal, intent on increasing their own utility, to achieve a high level of global utility. The contribution of this paper is to highlight the need for a mechanism for open-loop adaptation, propose a collective intelligence based mechanism, report on a simulation study(section 4) that we carried out and discuss the feasibility of our method. We discuss related work in self-adaptive and service-based applications (section 6) and evaluate the strengths and weaknesses of our approach.

2 Self-Adaptation

According to *Oreizy et al* [16], "Self-Adaptive software modifies its own behaviour in response to changes in its operating environment". To enable self-

adaptation therefore, an application needs to have the ability to monitor its environment, detect the need for change, decide on what is to be changed, effect that change. This is conventionally called the Monitoring-Analyzing-Planning-Exeuction (MAPE) loop [14], thus implying that self-adaptation embodies a closed-loop mechanism. This closed-loop structure has implications for the kinds of systems that are engineered to exhibit self-adaptation. A distributed system consisting of many different components, without a centralized monitoring, planning and directing unit is not considered a candidate for self-adaptation in this manner. Self-Adaptation is an umbrella term [19], referring to self-* properties:

1. Self-Awareness
2. Context-Awareness
3. Self-Configuring
4. Self-Optimizing
5. Self-Healing
6. Self-Protecting

Each of these self-* properties affect different QA attributes of an application. For instance, self-configuring affects maintainability, portability, while self-optimizing affects performance/efficiency. There are no self-adaptive mechanisms that address all of the above properties. We too, look only at self-optimization in service-based systems. We look at open-loop systems, consisting of hundreds or even thousands of adapting components and, propose a mechanism for guiding the self-optimization process. By an open-loop system, we mean a system that does not employ the conventional MAPE loop [19]; where a central monitoring or analyzing and planning engine, is not desirable or even feasible. However, creating a set of distributed adaptation agents, is difficult task. Since distributed agents do not all share the same perspective of their operating environment, their actions might work at cross-purposes. Even without actively undermining another agent's actions, it is easy to envision agent actions resulting in a net decrease in the global utility [31], viz., Tragedy Of Commons.

2.1 COllective INtelligence (COIN)

Using the actions of many distributed agents to achieve a global optimum was first addressed as Arthur's El Farol Bar Problem [2], which is now mathematically formulated as *The Minority Game* [32]. *Tumer and Wolpert* [23] have addressed the more general problem of configuring a group of distributed nodes, such that they contribute towards achieving a global goal. However, they use reinforcement learning agents as their choice of intelligence at the agent level. We propose using a simpler form of intelligence, that of simple utility theory [29] to allow agents to collectively achieve a global goal. The major difference is that it is the structure of the mechanism that allows individual action to lead us to the desired goal, as opposed to *Wolpert and Tumer's* approach of tailoring the utility functions and hence, behaviour of the individual agents. This implies that regardless of the strategies employed by the individual agents, the collective set

of agents as a whole still reach a high level of global utility.
The mechanism that we chose is the *Continuous Double Auction*(CDA). A CDA, unlike a traditional *English auction*, consists of multiple buyers **and** multiple sellers. Each buyer makes a bid quoting the price that she is willing to pay, and the seller makes an 'ask' asking for a certain price. As soon as a buyer's bid-price matches with a seller's asking price, the trade is executed. The buyers and sellers, thus, continually enter and exit the market. We contend that this is a form of collective intelligence. The structure of the market allows it to reach very high levels of allocative efficiency, inspite of the absence of a centralized price setter and extremely limited information [24]. Buyers and sellers seek only to maximize their private utility, however, as an unknown by-product they're contributing to highly efficient decentralized resource allocation. We consider the problem of matching web-services to applications, given each application's QA requirements and budget constraints. We are, however, **not** interested in the problem of a single application choosing an optimal set of web-services, for given QoS constraints. We are interested in looking at how, many applications *simultaneously* chose web-services, each according to their own requirements and constraints, in an efficient manner. In other words, consider a universe of 'n' applications, choosing from amongst 'm' services, using multiple criteria. There is a plethora of literature on the web-service selection problem, *i.e.*, finding the best possible match of web-services for a given application. In contrast, we do not attempt to find the optimal set of services for a given application. Rather, we attempt to find web-services that meet an application's minimum QA requirement within its budget, and attempt to satisfy as many applications as possible. Thus, we define global utility to be the number of applications that have successfully found web-services, matching their QA requirements and budget constraints.
The question that we seek to answer, in this paper, is the following:

– We want to consider how to maximize global utility, with no guiding, overarching layer to make decisions about individual agents' action.

Such a world can be instantiated in the form of a cloud. Each cloud provider hosts many applications, each of which is composed of several web-services. We imagine these services being provided by different service providers and functionality niches being occupied by different enterprises. The problem can be illustrated in more detail by an example.

3 Motivating Example

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 but slow-speed storage, while others might require smaller amounts but 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. The cost of change would depend on factors like supply of desired component, switching cost etc.

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. From a cloud provider's perspective, it is an important task, 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. We contend therefore that cloud implementations should provide a mechanism that alleviates this problem. Ideally it should be a self-managing approach that accounts for changes in operating environment and user preferences.

4 Mechanism Design

Mechanism-design is a sub-field of micro-economics and game theory, that considers how to design system-wide solutions to problems that involve multiple self-interested agents [17]. We design a marketplace that allows individual applications to select web-services, in a decentralized manner. This is important from the view of robustness and even practicality. The cloud is designed to be an ultra-large collection of applications, web-services and raw computing power. Given this large scale, any solution that is implemented, must not rely on global knowledge or coordination. The design of the market that we chose, is the double-auction. A double-auction is a an auction mechanism where, (unlike the traditional English or Dutch auction) both buyers and sellers place bids. In auction terminology, the buyer's bid is called a *bid* and the seller's bid is called an *ask*. We chose this mechanism because a double auction is known to be highly allocatively efficient [11].

We view an application as a composition of several web-services. There are six basic service composition patterns [34] as shown in Fig 1. In this paper, we con-

centrate on the first pattern, which describes a sequential composition. In this pattern, an application is composed of services that feed into each other sequentially. Each of these web-services contributes towards the total QA exhibited by the application. A web-service corresponds to a piece of functionality, along with associated QA levels. Thus, in a market for ray-tracing web-services, each seller will provide a web-service that performs ray-tracing, but offering varying performance and dependability levels. Varying levels of QA require different implementations and depending on the complexity of the implementations, will be either scarce or common. This leads to a differentiation in pricing based on QA levels.

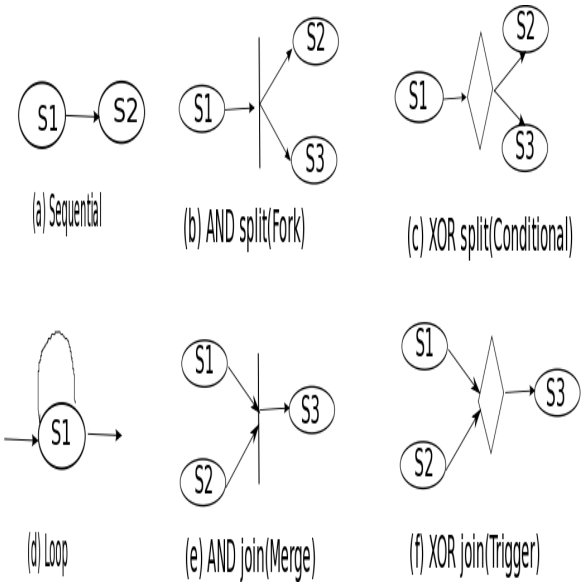

Fig. 1. Types of service composition

We populate several markets with buying agents and selling agents. Each market describes a certain functionality (f_x). All the buying agents and selling agents in this market, trade web-services that deliver f_x, where $f_x \in F_x$. The number of markets is at least equal to $\mid (F_x) \mid$, with each market selling a distinct f_x. In other words, we assume that there exists a market for every f_x, that is needed.

Buyer: The Buyer is a trading agent that buys a web-service for an Application, *i.e.,* it trades in a market for a specific f_x. Each web-service, available in f_x, exhibits the same QAs ($\omega \in QA$). The only differentiating factor is the degree to which that is exhibited ($\omega_i \in \mathbb{R}_0^1$). Hence, if an application has K QAs that it

is concerned about, then the QAs that it gets for each of the f_x that it buys is:

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

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_{f_x}, where B_{f_x} is the budget available with the buyer. The combination of Ω demanded and the *bid price* is called a *Bid*. Selecting the ω^{f_x} to bid for, is a crucial part of the adaptation process. For this purpose, the buying agent looks at the minimum QA values that the application demands from that f_x, and creates utility indifference curves. Utility indifference curves are various values of QA, for which the resultant utility is the same, and hence the application is indifferent amongst the actual values. Having created several levels of indifference curves, the buying agent selects the highest level and creates a bid for it. It exhausts the various points of QA values on that indifference curve, before jumping to a lower one. This behaviour allows the Buyer to explore different parts of the search space. While on one level, the buyer explores different QA values in the neighbourhood, to see if it is a possible solution. Then it performs a gradient descent by jumping to a lower indifference level.

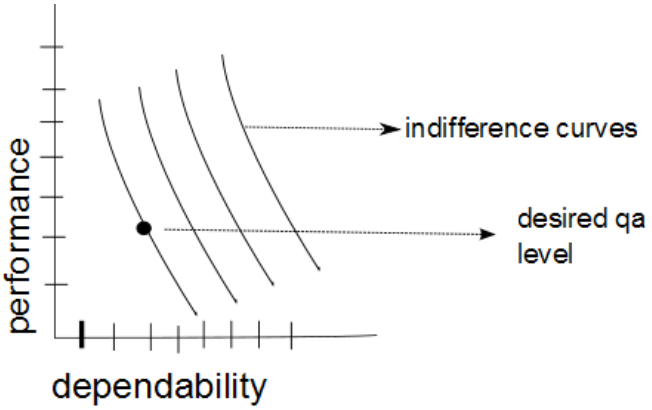

Fig. 2. Desired QA and Indifference curves

Seller: Each seller is a trading agent, selling a web-service that exhibits the QA required in (1). The degree to which each qa(ω) is exhibited in each f_x being sold, is dependent on the technological and economic cost of providing it. Hence, if the cost of providing f_x with $\Omega = \langle 0.5, 0.6 \rangle$ is low, then there will be many sellers providing f_x with a low *ask price*. Conversely, if the cost of providing f_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 for the purpose of this experiment we

consider only a simple direct relationship between cost and *ask price*, where the *ask price* is always *greater than or equal to* the cost. The combination of Ω and *ask price* is called the *Ask*.

Market: A market is a set of buyers and sellers, all interested in the same functionality f_x. The factor differentiating the traders are:

- Ω**:** 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 Ω of the ask are *greater than or equal to* all the QA attributes in the Ω of the *bid*. 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 [11]

Applications: The Application is a composition of buyers from different markets. Thus, an application is composed of at most $\cap(F_x)$ buyers with a privately known Ω_{f_x} for each buyer. The buyer gets a certain budget (B_{f_x}) from the Application for each round of trading. After each round of trading, depending on the Ω obtained by the buyer, the Application has procured a total QA that is given by:

$$\forall f_x \in F_x, \sum \Omega^{f_x} \tag{2}$$

Note, that (2) is merely one way of calculating the total QA that the application got. There are other, more complicated utility functions possible, but for the sake of simplicity, we use a simple, additive measure.
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_{f_x})
- *Transaction Cost:* The amount to be paid to the market, for making the transaction (t_{f_x})

– *Switching Cost:* The amount of penalty to be paid, for breaking the contract with the current web-service (s_{f_x})

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

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

We assume that, for every application there exists a function that maps TC_{f_x} to an equivalent Ω^{f_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 ($f_x \in F_x$) is a matter of strategy and/or the relative importance of each f_x.

$$\forall f_x \in F_x, \sum B_{f_x} = M \tag{4}$$

Traditionally, a CDA works on a single piece of information, price. However, we modify the bids and asks to be multi-attribute, *i.e.,* each bid also contains the minimum QA levels that the buyer wants and each ask contains the maximum QA levels that the seller is willing to provide. Thus, a transaction between a buyer and a seller is now an agreement to provide a certain functionality, at a certain price, at pre-defined QA levels. After every round of trading, the application calculates the total utility that it obtained after the trades and compares it to the previous utility it had. Depending on whether the utility in the current round is better or worse, the application provides an appropriate feedback to all the buying agents. This allows an agent to realize the effect of its action on the overall utility. It then modifies its bid, accordingly.

In Figure 4, we see the functionality being composed at the top and the agents below calculating the utility achieved by each trade. The agents are also able to observe the last known transaction price and QA levels in their markets. This allows them to create bids, that are most likely to result in a trade.

4.1 Assumptions

One of the assumptions in this simulation, is that supply of a web-service with the desired Ω_{f_x} always exists; the buyer only needs to pay a high-enough price to be able to procure it.

5 Evaluation

5.1 Methodology

We use a simulation to test our market structure. We simulated multiple markets, applications composing each of their functionality, by buying services from these markets. An application possesses a buying agent for each market. This

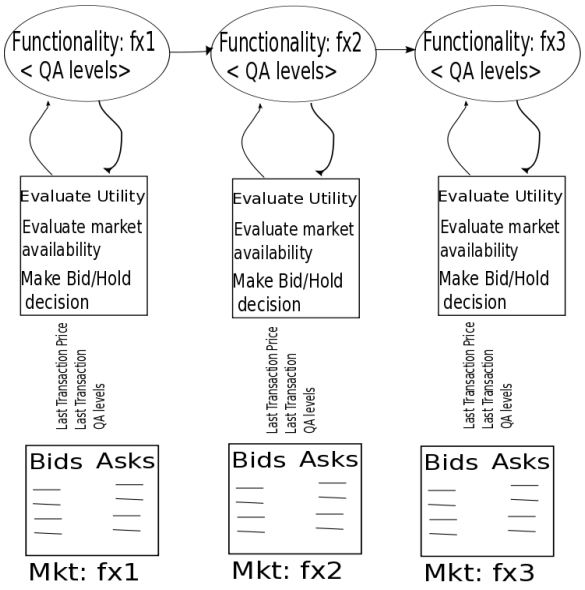

Fig. 3. Web-services with agents that decide when to approach the market

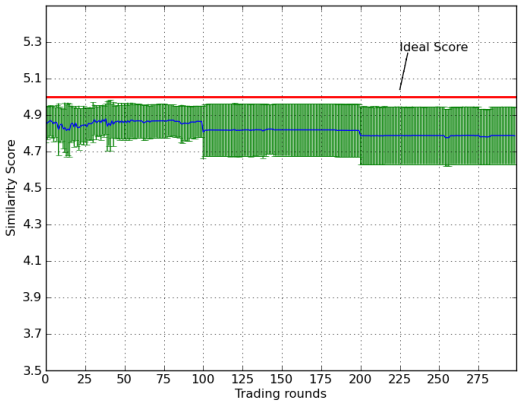

Fig. 4. Self-adaptation in the presence of equal demand and supply

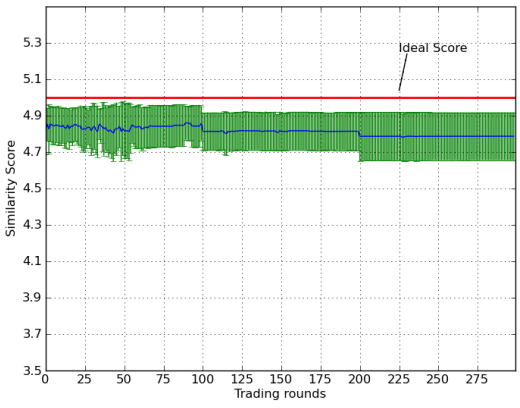

Fig. 5. A market with excess supply of services

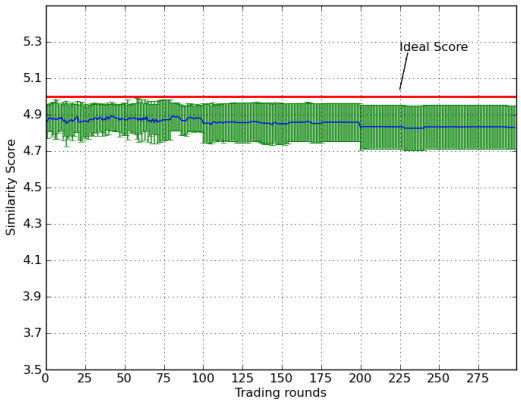

Fig. 6. A market with excess demand of services

buying agent is responsible for understanding the market protocol, observing the prices in the market and bidding for a web-service. The application composes the web-services and gives feedback on the total utility derived. The buying agent is thus able to work out how its action affected the entire application's utility. We want to test whether the collective set of buyers are able to use the feedback given by their application, and the ZIP strategy to negotiate on price, to attain a high level of global utility.

We calculate the *ideal score*, which is the equilibrium level of utility that that set of applications should attain. The results shown are the averaging of 30 simulations, with each simulation running for 300 trading rounds. After every 100 rounds, the applications would dynamically change their desired QA levels, which would require the buying agents to re-adapt their bids. Ideally, the global utility attained by the market should be equal to the *ideal score*.

The simulations were done on a quad-core CPU machine with 2GB of RAM. The absence of network traffic between the simulation entities meant that the simulations performed faster than if it were across a network. However, as the number of traders increased, the amount of memory swapping also increased, thus more than counter-acting the performance effects of a single-machine simulation.

5.2 Analysis

In proposing a market-based approach, we're aware that the performance of the simulation depends critically on the market conditions. Hence, we tested the approach against three market conditions:

1. Markets with balanced supply and demand
2. Markets with excesss supply
3. Markets with excess demand

As we see from the figures (Fig 4 5 6), we see that the combined utilities of the applications (global utility) is able to reach quite close to the ideal score.

Observe that in Fig 4, after the initial perturbation where the agents work out what the application desires, the rest of the peturbations happen only at QA-change-boundaries, *i.e.,* at trading rounds which are multiples of 100, where the applications used to change their desired QA levels. Excess demand and Excess supply were simulated by halving the number of suppliers and doubling the number of suppliers, respectively. Also observe, in Fig 6, that the peturbation never really settles down. But this is to be expected, since each buyer tries to make a trade and not everyone is able to do so, due to a paucity of suppliers.
The reason that a double auction performs so well in searching through the search space of possible application-to-webservice matchings is due to all the agents performing search in parallel. In our simulation, all the agents use *Zero Intelligence Plus* (ZIP) [6] strategy, which uses the Widrow-Hoff learning rule to guide buyers and sellers to prices that allow for a high amount of trade. ZIP is a fairly simple and intuitive strategy, however other sophisticated strategies like "Gjerstad-Dickhaut" [10], "Roth" [18] and their modifications would prove to be much better and faster at reaching market equilibrium.

5.3 Strengths

We see in all cases, that the mechanism allows applications to reach very close to their 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. The mechanism scales fairly well in our experiments with increasing the number of buyers and sellers in each market. The novelty of this method of collective intelligence is that it is the structure and rules of the mechanism that result in a high level of world utility, rather than the intelligence in the behaviour of the individual agents.

5.4 Weaknesses

With regard to performance of the mechanism, our simulation was on a single machine and hence the scaling results (Figure 5.3) do not appear to be linear. As the number of traders increased beyond 1600, the amount of memory swapping that took place, started to slow down the time taken to finish a round of trading. Also, while the allocative efficiency of a continuous double auction is very high and there are a multitude of trading strategies, it is very difficult to analyze formally. For this reason, it is difficult to know whether a particular trading strategy or a market parameter is optimal. Also, the indifference curves assume that the search space is smooth, but this may not be necessarily true in all situations.

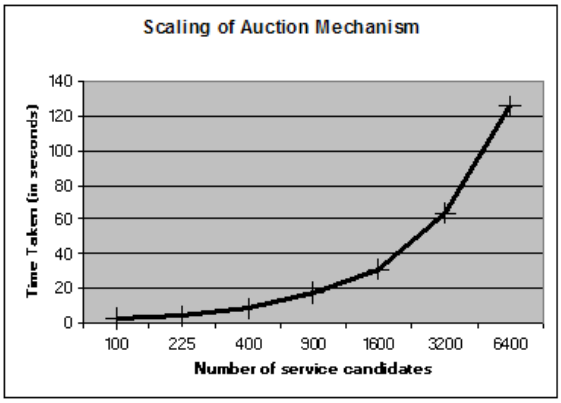

Fig. 7. Performance scaling with increase in traders

6 Related Work

Wolpert et al. [30] have done a lot of work in formalising the requirements for engineering a system that exhibits collective intelligence (COIN). They show through experiments on packet routing in data networks [31, 23] that a system designed as COIN exhibits better performance than Ideal Shortest Path Algorithm. They use Reinforcement Learning algorithms, with utility functions of each agent designed not decrease the global utility.

Using market-based techniques for solving distributed resource allocation problems, is not a new idea. The field of computational economics is concerned with economics-inspired techniques to guide interactions between components of a distributed system. *Tucker et al* [22] have a good survey of market-based techniques used in the software domain. Notable examples include Clearwater's bidding agents to control the temperature of a building [5], Ho's center-free resource algorithms [33] and Cheriton's extension to operating sytems to allow programs to bid for memory [12]. In all of these works, distributability of the system is not a major concern. Indeed, the auctioneer is a centralized agent, and thus runs into the usual problems of communication and scaling. An exception is Wellman's WALRAS system [28], which is explicitly distributed and reports high scalability. More examples include distributed Monte-Carlo simulations [25], distributed database design using market-methods for distributing sub-parts of queries [20] and proportional-share resource management technique [26]

Self-Adaptive systems were first proposed by *Oreizy et al* [16] and since then, there has been extensive work done on engineering systems that exhibit the autonomic properties discussed in section 2 and, as reported by Kramer and Magee [15]. Current research pursues software architectures as the appropriate level of abstraction for evaluating, reasoning about, managing and facilitating the dynamic change and evolution of complex software systems. 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 [7].
– Create a framework for specifying types of adaptation possible using constraints and tactics thus ensuring 'good adaptation ' [9, 4].
– Using formalized methods like SAM [27], control theory [13] and middleware optimizations [21].

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, and is always timely.
3. The management component gets full feedback or is able to view the results of changes made, on the entire system.

We feel that these are limiting assumptions, and hence we approach the problem without assuming a closed-loop. In the domain of service-based applications, due to the late-binding approach enabled by web-services, there has been an increasing interest in dynamic web-service composition. *Zeng et al.* [35] 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.* [34] 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 application 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.

7 Conclusion and Future Work

We consider that the Market-based approach to dynamic web-service assignment 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. There is no 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 to asymmetric assignment. The market structure is inherently distributed, however our simulation was on a single machine. We aim to run distributed simulations to validate our hypothesis that the market mechanism is inherently parallelizable and will scale linearly.
We shall also be looking at tuning of various parameters of the market and buyers' learning strategies, to improve its performance. Depending on the QAs that we're interested in, the seller's prices and QA availability might be starkly different. We intend to model QAs that have different relationships (highly correlated, anti-correlated, un-correlated) amongst themselves, to observe their effect on the achievement of global utility.

References

1. J. Anselmi, D. Ardagna, and P. Cremonesi. A qos-based selection approach of autonomic grid services. In *Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, SOCP '07, pages 1–8, New York, NY, USA, 2007. ACM.
2. B. W. Arthur. Inductive reasoning and bounded rationality. *American Economic Review (Papers and Proceedings)*, (84):406–411, 1994.
3. G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. Qos-aware replanning of composite web services. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, pages 121–129, Washington, DC, USA, 2005. IEEE Computer Society.
4. S. Cheng, D. Garlan, and B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, Shanghai, China, 2006. ACM.
5. S. H. Clearwater, R. Costanza, M. Dixon, and B. Schroeder. Saving energy using market-based control. pages 253–273, 1996.
6. D. Cliff and J. Bruten. Less than human: Simple adaptive trading agents for cda markets. Technical report, Hewlett-Packard, 1997.
7. E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.
8. E. DiNitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engg.*, 15(3-4):313–341, 2008.
9. I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA, 2002. ACM Press.

10. S. Gjerstad and J. Dickhaut. Price formation in double auctions. Microeconomics 0302001, Minnesota - Center for Economic Research, 1995.
11. D. K. Gode and S. Sunder. 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.
12. K. Harty and D. Cheriton. *A market approach to operating system memory allocation*, pages 126–155. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.
13. J. Hellerstein. *Engineering Self-Organizing Systems*, page 1. 2007.
14. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.
15. J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.
16. P. Oreizy, Gorlick, Tayor, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.
17. D. C. Parkes. *Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, May 2001.
18. A. E. Roth and I. Erev. Learning in extensive-form games: Experimental data and simple dynamic models in the intermediate term. *Games and Economic Behavior*, 8(1):164 – 212, 1995.
19. M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.
20. M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An economic paradigm for query processing and data migration in mariposa. In *PDIS '94: Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 58–67, Washington, DC, USA, 1994. IEEE Computer Society.
21. M. Trofin and J. Murphy. A Self-Optimizing container design for enterprise java beans applications. In *In Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004*, 2003.
22. P. Tucker and F. Berman. On market mechanisms as a software technique, 1996.
23. K. Tumer, A. K. Agogino, and D. H. Wolpert. Learning sequences of actions in collectives of autonomous agents. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 378–385, New York, NY, USA, 2002. ACM.
24. P. Vytelingum. *The Structure and Behaviour of Continuous Double Auctions*. PhD thesis, School of Electronics and Computer Science, University of Southampton, 2006.
25. C. Waldspurger, T. Hogg, B. Huberman, J. Kephart, and W. Stornetta. Spawn: a distributed computational economy. *Software Engineering, IEEE Transactions on*, 18(2):103–117, feb. 1992.
26. C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 1, Berkeley, CA, USA, 1994. USENIX Association.
27. J. Wang, C. Guo, and F. Liu. Self-healing based software architecture modeling and analysis through a case study. In *Networking, Sensing and Control, 2005. Proceedings. 2005 IEEE*, pages 873–877, 2005.

28. M. P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *J. Artif. Int. Res.*, 1(1):1–23, 1993.
29. Wikipedia. Von neumann-morgenstern utility theorem — wikipedia, the free encyclopedia, 2010. [Online; accessed 22-July-2010].
30. D. H. Wolpert and K. Tumer. *An Introduction to Collective Intelligence*. Aug. 1999.
31. D. H. Wolpert and K. Tumer. Collective intelligence, data routing and braess' paradox. *J. Artif. Int. Res.*, 16(1):359–387, 2002.
32. D. H. Wolpert, K. R. Wheeler, and K. Tumer. Collective intelligence for control of distributed dynamical systems. *Europhysics Letters*, 49:708–714, 2000.
33. L. S. Y. C. Ho and R. Suri. A class of center-free resource allocation algorithms. *Large Scale Systems*, 1:51, 1980.
34. T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Trans. Web*, 1(1):6, 2007.
35. L. Zeng, B. Benatallah, A. H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5):311–327, 2004.