

Design of a Market-Based Mechanism for Quality Attribute Tradeoff of Services in the Cloud

Vivek Nallur, Rami Bahsoon
School of Computer Science
University of Birmingham
Birmingham B15 2TT, United Kingdom
{V.Nallur, R.Bahsoon}@cs.bham.ac.uk

ABSTRACT

Cloud computing, with its promise of (almost) unlimited computation, storage and bandwidth, is increasingly becoming the infrastructure of choice for many organizations. As applications gain in popularity and mature, the quality attributes demanded of them change significantly. Applications that manage themselves and exhibit different quality attributes, based on demand, are the ideal that we would like to have. Creating self-managing applications for the cloud present significant problems, since the cloud infrastructure is not under the application architect's control. We propose an initial design of novel market-based mechanism to allow web-applications living on the cloud to self-manage with regard to their quality attributes. We use a scenario to exemplify and evaluate the approach.

Categories and Subject Descriptors

D.2.10 [Methodology]: Emergence

Keywords

Self-Organization, Market-Based Control, Cloud Computing

1. INTRODUCTION

There are lots of web-applications today, increasingly used for ever-more sophisticated tasks. From procurement auctions for businesses to email (à la GMail) to social networking (à la Facebook), web-applications are pervasive in our daily computing. The increasing prevalence of these web applications leads to an interesting conundrum. With increasing demands on it, how does a web application provide the relevant Quality of Service (QoS), dynamically and on demand? Initially when GMail entered the web-based email world, it's USP was the AJAX-powered UI that made composing and sending email a high-speed task. However, this very same high-performance that has led to its becoming the one of the most popular email systems around, has also resulted in a changed requirement. With such a large number

of people using GMail, it becomes increasingly critical for GMail to be highly stable, secure and available. Downtime of email is unacceptable, since many people and companies rely on it for communication. Again, since a large number of individuals and enterprises use it, security of the data being transferred and stored is a key concern. From performance, GMail's focus has changed to be availability and security. How does a web-application that was not initially designed with these Quality Attributes in mind, re-configure its architecture to reflect the changing runtime requirements?

Most web-applications are hosted on datacenters, and more recently on clouds. The rise of cloud computing has freed the application provider from the logistical problems of provisioning and maintaining multiple machines; instead leasing as many as are required at that particular instance in time. This type of arrangement has the advantage of 'cost associativity' offered by the cloud [1]. This flexibility in cost, however, has a downside. Although the cloud providers make available large amounts of computing power and storage, they make no guarantees about the QoS being provided by them. By QoS, we refer to qualities like reliability, availability, performance, security and other non-functional requirements which need to be provided, maintained, evolved and monitored at runtime. As mentioned previously, these qualities are fundamental to the user's satisfaction with the application.

While issues like performance can ostensibly be solved by increasing the amount of computational power available through the cloud, the real problem of change in the cloud does not get solved. The current approach to solving this problem is, to look at all the possible quality attributes that the application must have and design for them *a priori*. However, it is obvious that, from a long-term point of view, this is not a feasible solution. Not only is it difficult to predict what future requirements will be, it is near impossible to predict how changes in user-profiles and usage will affect those requirements. We posit that the solution to this problem lies in self-management. Self-Management for an arbitrary application is difficult, however web-applications have some properties, which can be utilized to implement self-management. Web-applications can easily be constructed out of web-services which are themselves dynamically composable. The environment of the cloud, however, is vastly different from anything that we have designed for, up to now. The unrestricted scale alone makes it difficult. Adding properties such as reliability and autonomic adjustment makes it even more difficult. We propose a novel, market-based mechanism in which web-services can determine which other

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'10 March 22-26, 2010, Sierre, Switzerland.

Copyright 2010 ACM 978-1-60558-638-0/10/03 ...\$10.00.

web-service to connect to, and what the advantage of connecting to that web-service would be.

The above problem has been discussed in more detail by *Nallur et al.*[6] and they posited that the solution lies in self-management. In this work, we present the outline of such a solution and an initial design of the same. We propose to use the economic principles of a market to incentivise and control change in the architecture. We set up a market where components interact as trading agents, and act selfishly according to the utilities that they derive. Self-Management of an application’s architecture therefore becomes the emergent property of this market. We report on the design of such a marketplace, and its essential elements. This paper is structured in the following manner: In section 2 we discuss the mechanism; in section 3 we apply and evaluate the mechanism through the means of a scenario; in section 4 we discuss the related work and finally, in section 5 we present our concluding thoughts.

2. THE MARKET MECHANISM

Market Oriented Programming has been used by many researchers to study and control distributed systems. It is essentially an attempt to blend principles of economics with computational problems. It is not merely used as a metaphor for a distributed system. It is also a literal creation of a marketplace, that is set up in such a manner that, with the application of economic principles of selfish, rational behaviour, multiple agents reach an equilibrium point. In our case, by equilibrium, we mean a ‘good-enough’ state of tradeoffs made between different quality attributes and the cost of achieving those quality attributes. We can view applications as either a buyer of web-services with certain quality-attributes, or a seller that is capable of delivering those quality attributes at a certain cost. The resource allocation problem can be set up as an optimisation problem, where the buyers need to maximize their QA, given that they have a limited budget while sellers have a limited capacity to sell.

We start off with a minimal market that exhibits interesting behaviour. We treat the universe of web-services as an economy, consisting of several marketplaces, several buyers, several sellers. All of these exhibit *ex-post Individual Rationality* (IR). This means that none of the entities in the economy exist for altruistic reasons. All their actions are rational and will result in a non-negative utility for them. The marketplace operates a *continuous double auction* (CDA) which brings buyers and sellers together, and decides when a transaction should take place and at what price. We will now elucidate each of the entities in our system:

1. **The Buyer:** This is the application that we are primarily concerned about. This is the application that re-configures its architecture through the process of buying web-services. The application receives a relative weighting amongst the QAs that it is concerned about. So if ω denotes the weight for a particular QA and K be the set of QAs, then:

$$\Omega = \langle \omega_1, \omega_2, \omega_3, \dots, \omega_K \rangle \quad (1)$$

and

$$\sum_{k=1}^K \omega_k = 1 \quad (2)$$

where

$$\omega_k \in \mathbb{R}_0^1$$

Crucial to the notion of a market-based approach is a constraint on the buyer, in the form of price. If ℓ is the limit price of the buyer, and \cap be the number of API calls that it is bidding for, then the **bid** from buyer i (where $i \in B$, the set of all buyers) is given by the vector:

$$b_i = \langle \Omega_i, \ell_i, \cap_i \rangle \quad (3)$$

Based on the limit price (ℓ_i) and the relative weights (ω) assigned to each QA, the application can work out the *expected value* (EV), in monetary terms, of each QA. When a buyer buys a contract for a web-service for \cap calls and starts using it, a monitoring component assigns an *actual value* (AV) to each QA exhibited by the web-service. If the difference between the EV and the AV crosses a threshold (when AV is less than EV), the buyer breaks the contract and goes back to the market with a **bid** for a better web-service. Thus, for a buyer, the sequence of steps involved in self-management are as follows:

- (a) A human assigns a budget to the web-application
- (b) A human assigns relative weights to each quality attribute that she is concerned about
- (c) The application partitions the budget amongst the functionalities that it provides
- (d) For each distinct functionality, the application enters a different marketplace and performs the following steps
 - i. Creates a bid consisting of the maximum price it is willing to pay and the level of QA(s) that it needs
 - ii. Post bid to the market
 - iii. If matched within t time, connect to seller’s web-service, compose functionality, proceed to (1(d)v)
 - iv. Else, adjust bid (changing price or any QA). Proceed to (1(d)ii)
 - v. Monitor composed web-service to ensure that AV is atleast as good as EV
 - vi. If AV is significantly lesser than EV, modify bid and proceed to (1(d)ii)
2. **The Seller:** This is the application that sells web-services to the highest bidder. This application has a minimum ‘ask’ price, below which it is not economical for the seller to sell. This is so due to the fact that computation, storage and data transfer all have a cost in the cloud. These are all paid by the seller’s web-service. Also in a realistic setting, for any individual seller, the actual amount of computation, storage and networking bandwidth on offer will be finite. This can be denoted by cap_s , where $s \in S$, the set of all sellers. The limit price (ℓ_s) for a seller will be dependant on

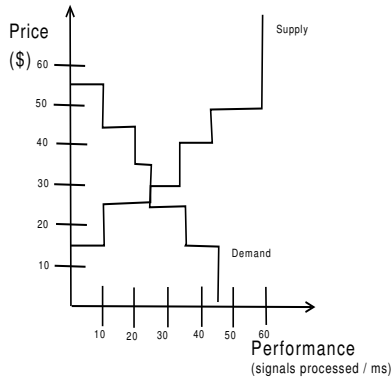


Figure 1: The supply and demand in Marketplace

the cost that she incurs, for providing the web-service. The cost can be given by:

$$C_s = \begin{cases} 0 & \text{No cost if web service is not sold} \\ \cap_s * u_s & \text{Number of API calls * unit cost} \\ & \text{where } \cap_s \leq cap_s \end{cases} \quad (4)$$

The seller’s ask is given by the vector:

$$a_s = \langle \Omega_s, \ell_s, \cap_s \rangle \quad (5)$$

3. **The Marketplace:** This is an application that resides in the cloud, and acts as the meeting point for buyers and sellers. Our condition of IR means that this application does not exist for altruistic purposes. That is, it gains some amount of money by virtue of bringing buyers and sellers together. The more the number of transactions that occur, the more it earns. The marketplace can be characterized by the following vectors, from equations (3) and (5):

$$Bids = \langle b_1, b_2, b_3, \dots, b_B \rangle$$

$$Asks = \langle a_1, a_2, a_3, \dots, a_S \rangle$$

The market is said to be optimally efficient, when the all the possible bids and asks that can be matched, are matched for a transaction. Simplistically speaking, the greater the volume of trade, the greater the efficiency of the market.

We now elucidate the properties of the marketplace, since it is these properties that influence the kind of emergent solution computed. The marketplace operates a **continuous double auction (CDA)**. A double auction is an auction where both buyers and sellers post quotes about the quantity being traded and the price they’re willing to pay/ask. Deciding when a transaction has happened is called clearing the market. A double auction can be cleared on *one-shot* or a *repeated* basis. In a repeated auction, clearing of the auction occurs either at discrete, periodic times or on a continuous basis. A CDA operates a continuous clearing mechanism. It has been suggested by *Friedman et al* [5] that continuous clearing results in a greater volume of trade.

If the application’s QA focus changes, a human adjusts the weights for each QA thus causing the *expected value* (EV) to change. This causes a discrepancy between the EV and the AV, which could potentially cause the application to revisit the marketplace. The threshold at which the buyer revisits the marketplace is a function, that depends on its budget (M), last price paid (TC) and the expected value.

We see immediately from the steps outlined above, that there’s a lot of potential for sophistication in the way that each of the steps are carried out, in terms of learning from past behaviour, creating bids to maximize benefit to self, etc. However, we seek to see if the simplest possible implementation of the process above, would lead to successful adaptation.

3. EVALUATION

3.1 Scenario

We take a hypothetical scenario and follow the steps outlined in (1) to show that a web-application, is able to self-manage its QA responsibilities. Consider a new social networking startup, called myChaseBook that provides the following innovative services:

- Track friends geographically via cellphone, overlaid on a city map
- Tagging of streaming radio
- Cellphone based access to friends’ music, photographs, restaurants based on location on a map

myChaseBook, being a startup, doesn’t have enough resources to invest in a huge datacenter. Also, it knows that the services that it offers will be attractive to users only, if the tracking and tagging are done fast enough to ensure a good user experience. myChaseBook decides to host its application in the cloud where computing power can be scaled up automatically. It also decides to outsource some cellphone signal processing components that it needs, but has no expertise in building. This has the dual advantage of adding functionality and cutting time-to-market, both of which are critical for a startup. It decides to build its application using web-services and enter our web-service economy. The steps that it follows are:

1. The architect assigns \$40 (ℓ_b) to myChaseBook
2. The architect assigns weights of $\Omega = \langle 0.5, 0.5 \rangle$ each to the QAs: performance and reliability (from 1 and 2)
3. myChaseBook partitions the \$40 into two lots of \$20 each and assigns them to performance and reliability
4. myChaseBook enters the marketplace for radio signal processing web-services and performs the following market-specific steps:
 - (a) It creates a bid.
 - (b) The marketplace evaluates all the bids and asks that it has.
 - (c) Once myChaseBook gets a web-service that is willing to provide it the functionality at \$36, it connects to the seller, composes the functionality and monitors the performance delivered by seller’s web-service.

Limit Price	A limit price is the maximum price that a buyer is willing to pay and the minimum price that a seller is willing to accept
Bid	The buyer's bundle of requirements. Includes the maximum price, it is willing to pay
Ask	The seller's bundle of offer. Includes the minimum price, it is willing to accept
Actual Value (AV)	The value that the application gets on a particular QA attribute. This is on a per-webservice basis
Expected Value (EV)	The value that the application expects on a particular QA attribute. This is on a per-webservice basis. A significant difference between EV and AV causes the application to enter the marketplace
Web Service Contract (WSC)	The finalised terms of contract between the buyer and the seller. The value of the contract is average between the minimum price expected by the matched seller and the maximum price expected by the matched buyer. The matching must fulfill or exceed the QA criteria specified by the buyer
Above Market Minimum (AMM)	A real number that quantifies how much better than the benchmark, this webservice is. Each marketplace will publish a benchmark (Market Minimum) for each QA For e.g. A marketplace for sorting web-services publishes a MM as: 1GB of data in 10 seconds
Transaction Cost (TC)	A percentage of the value of the WSC. This allows the marketplace app to gain value from the transactions it is able to generate

Table 1: Terms used in the Marketplace

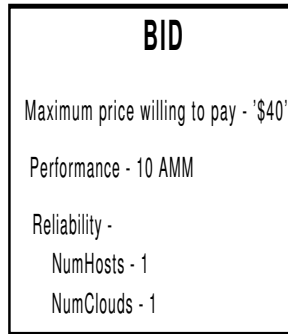


Figure 2: A sample bid that myChaseBook makes

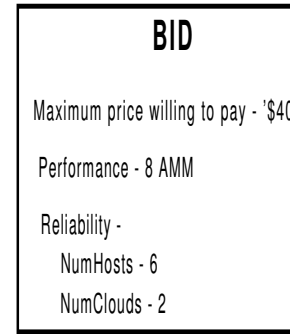


Figure 3: A modified bid that myChaseBook makes

- (d) As long as the *Actual Value* of performance(\$18) and reliability(\$18) provided by the seller's signal processing web-service is more than the *Expected Value* of \$20 each, the contract continues.

Suppose now that myChaseBook becomes hugely popular and many users sign up. Suppose again, that as more people start to rely on its services, one of the clouds that was hosting the seller's web-service suffers an outage. Since this causes myChaseBook's reliability to suffer, the architect modifies the relative weights of the QAs to $\langle \omega_1 = 0.4$ (performance), $\omega_2 = 0.6$ (reliability)). This causes the Expected Value of reliability to rise, causing (EV-AV) to rise above the threshold. myChaseBook takes the following steps:

1. It creates a modified bid and re-enters the marketplace (Note the modified values of performance and reliability)
2. The marketplace evaluates the asks and bids, and myChaseBook possibly finds a different seller.
3. If myChaseBook does not get matched within t time, then it modifies its bid by increasing the price that it is willing to pay

Thus, through a process of adjusting the price and minimum value of the Quality Attribute, myChaseBook self-manages to keep pace with the organization's objectives.

3.2 Discussion

The novelty of this approach is not in doubt. There has been no work reported on using economic principles to self-manage software in the cloud. The use of market-based techniques in itself is not new in Computer Science, however they have mostly been used for problems in load balancing and resource allocation. *Buyya et al.* [2] propose a market-oriented approach for services in the cloud, but their concern is resource-reservation, and not architectural configuration. *Wolksi et al.* [9] investigate auction-based methods vis-a-vis commodities markets for Grids. Again, however, their concern is with resource-allocation. Using market-based techniques for architectural adaptation has not been reported, to the best of our knowledge. Combining it with the peculiar need for adaptation in the cloud, that of quality attributes rather than functionality, has not been attempted. This approach does not solve all problems of architectural adaptation in the cloud. It can only be used to manage the so-called runtime quality attributes like performance and reliability. Other statically determined QAs like modifiability and testability cannot be influenced easily, by this approach. A shortcoming of this approach is that the time taken for self-management is non-deterministic. Once the application determines that it needs to approach the market, there is no time-bound on how quickly it finds a suitable web-service to buy. This will limit its applicability in domains, like medical services, where the deadlines are hard. In domains like e-commerce, social networking etc., this is not a major liability and can be used to good effect, as a tactic against environmental change. As the needs of the users change, the

QA demanded of the application changes and our approach is adaptive in such scenarios. Since markets are inherently decentralized, our approach is very scalable in principle.

4. 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 [4].
- Create a framework for specifying types of adaptation possible using constraints and tactics thus ensuring ‘good adaptation’[3].
- Using formalized methods like SAM [8] and middle-ware optimizations [7].

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.

We feel that these are limiting assumptions because of the following reasons:

1. In a system of systems, each software component will have a different objective to attain and hence need not cooperate. Also, since each web-service is potentially owned or managed by a different entity, its optimization objective could be different. In our approach, each sub-system of an application could be self-managing with a different objective.
2. In a truly large scale system, communication between centralised management components and distributed lower-level components may not be feasible/robust. The Market-Oriented approach is decentralized and requires no coordinating nodes or planner.
3. In a large scale system, global application knowledge is infeasible. If there are thousands of object instances, each with its own performance/memory issues, it is infeasible to expect a central optimizer to take into account each of those instances’ states or metrics. Our approach does not assume any knowledge of the topology of the application, nor does it care about any other sub-system. The self-management is an emergent effect of each component trying to meet its economic objective.

5. CONCLUSION

The scenario outlined assumes a simplistic trading agent, which simply allocates all its budget and spends it all naïvely. In the real world, we expect trading agents to have memory

of previous transactions, strategies to bid the minimum possible price etc. Also, as selling agents become more sophisticated they might try to break their contracts, to re-enter the market and gain a better price. Evolution of these agents will lead to notions of penalties for contract-breakage, reputations etc. That is, over time we expect the autonomically traded web-services to achieve the same kind of complexity that real economies have.

We intend to simulate the behaviour of these simple and sophisticated trading agents and report on the effect of this kind of novel self-management on the attained Quality Attributes. Measuring these will involve creation of sophisticated utility functions to measure Actual Value gained from using a particular web-service. We will also report on the time taken for self-management to achieve a particular QA, and its determining factors.

6. REFERENCES

- [1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.
- [2] R. Buyya, C. S. Yeo, and S. Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 5–13, Washington, DC, USA, 2008. IEEE Computer Society.
- [3] S. Cheng, D. Garlan, and B. Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *Proceedings of the International workshop on Self-adaptation and Self-managing Systems*, pages 2–8, Shanghai, China, 2006. ACM.
- [4] 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.
- [5] Friedman and Rust, editors. *The Double Auction Market: Institutions, Theories and Evidence*, chapter 6, pages 199–219. Santa Fe Studies in the Sciences of Complexity. Perseus Publishing, Cambridge, USA.
- [6] V. Nallur and R. Bahsoon. Self-optimizing architecture for ensuring quality attributes in the cloud. In *Proc. 8th Working IEEE/IFIP Conference on Software Architecture*, Cambridge, UK, September 2009.
- [7] M. Trofin and J. Murphy. A self-optimizing container design for enterprise java beans applications. In *Proceedings of the Second International Workshop on Dynamic Analysis*, 2003.
- [8] 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.
- [9] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. Analyzing market-based resource allocation strategies for the computational grid. *International Journal of High Performance Computing Applications*, 15:258–281, 2000.