A Decentralized Self-Adaptation Mechanism For Service-Based Applications in The Cloud

Vivek Nallur, Rami Bahsoon

Abstract—Cloud computing, with its promise of (almost) unlimited computation, storage and bandwidth, is increasingly becoming the infrastructure of choice for many organizations. As cloud offerings mature, service-based applications need to dynamically recompose themselves, to self-adapt to changing QoS requirements. In this paper, we present a decentralized mechanism for such self-adaptation, using market-based heuristics. We use a continuous double-auction to allow applications to decide which services to choose, amongst the many on offer. We view an application as a multi-agent system, and the cloud as a marketplace where many such applications self-adapt. We show through a simulation study that our mechanism is effective, for the individual application as well as from the collective perspective of all applications adapting at the same time.

Keywords—Self-Adaptation, Market-Based, Multi-Agent Systems

1 Introduction

Self-adaptation, as a concept, has been around for many years, in several domains like biology, chemistry, logistics, economics etc. Self-adaptivity in computer-based systems is relatively newer. Some of the first references to self-adaptive software systems are from [41], [35], [34] and [31] (where they are referred to, as autonomic systems). By self-adaptivity in software systems, we mean software that monitors itself and the operating environment, and takes appropriate actions when circumstances change. In web-applications, service-oriented architecture has often been used as a mechanism for achieving self-adaptivity [19]. Web-services allow for dynamic composition, which enables applications to switch services, without going offline. A common instance of using web-services dynamically, is applications living on the cloud, asking for computing power and bandwidth to be scaled up or down, depending on demand. However, one of the cloud's major selling points, *operational flexibility* is of little use, if applications (or organizations) have to indicate at sign-up time, the kind of services that they intend to use. On Amazon, for instance, a customer specifies during sign up whether she wants a Hi-CPU instance or a Standard On-Demand instance or a Hi-Memory instance. This assumes that an application is able to forecast its demand for computing and storage resources accurately. However, this inability to forecast is precisely what the cloud claims to address through elasticity in computing power. This is not to say that there are no flexible, demand-based pricing schemes available. Amazon's *Spot Instances* [29] is an example of how cloud providers are trying to flexibly price their services, in response to fluctuating demand over time. Applications that can adapt to fluctuating prices will be able to ensure a better return on investment. In the future, we surmise that service-pricing will depend not only on demand but also on additional attributes like performance, availability, reliability, etc.

Current implementations of public clouds mainly focus on providing easily scaled-up and scaled-down computing power and storage. We envisage a more sophisticated scenario, where federated clouds with different specialized services collaborate. These collaborations can then be leveraged by an enterprise to construct an application, that is self-adaptive by changing the specific web-service it utilizes. The notion of utilizing collaborative services to satisfy a business need, is not new in itself. The recognition of Agile Service Networks (ASN) that spring up in modern business practices, is testament to this. As ASNs mature and dynamic composition becomes the norm, we posit that applications that are composed of other applications will routinely adapt to changing QoS requirements. In this paper, we propose a decentralized mechanism to address the problem.

1.1 Problem Statement

Composite web-services are services that are composed of other web-services. Several web-applications are made by composing web-services together. The effective QoS provided by such an application is a function of the QoS provided by the individual web-services. Hence, if the application wishes to exhibit a different level of QoS, it can do so by changing its constituent web-services. However this is not an easy task. Identifying the optimal web-service, for each task that the application performs, is a hard problem. To illustrate, consider an application composed of five tasks, and its choices of web-services:

In Table 1, if the application wanted to optimize on Price, the set of selected services would be $[S1_5, S2_1, S3_2, S4_1, S5_2]$. On the other hand, if it wanted to optimize on Latency, the selected services would be $[S1_1, S2_2, S3_4, S4_2, S5_4]$. The calculations become more difficult if the optimizations need to be done on multiple QoS simultaneously, for e.g., Latency and Price

Task	Web-Service	Latency	Availability	Price
S1	$S1_1$	100	0.95	105
	$S1_2$	180	0.92	66
	$S1_3$	230	0.79	65
	$S1_4$	170	0.73	90
	$S1_5$	250	0.96	60
S2	$S2_1$	200	0.98	58
	$S2_2$	150	0.93	80
	$S2_3$	200	0.97	65
S3	$S3_1$	250	0.96	70
	$S3_2$	240	0.91	50
	$S3_3$	230	0.79	65
	$S3_4$	150	0.73	80
	$S3_5$	210	0.89	68
S4	$S4_1$	260	0.94	59
	$S4_2$	225	0.91	60
S5	$S5_1$	150	0.82	54
	$S5_2$	140	0.71	49
	$S5_3$	130	0.93	76
	$S5_4$	120	0.73	81
	$S5_5$	190	0.86	77

TABLE 1: Choice of services available

or Latency and Availability. As the number of tasks in the workflow increase, the number of combinations also increase correspondingly.

The problem arises from the fact, that for each of these tasks (AbstractService) in the application's workflow, there are several services (ConcreteService), that can be used. Each of these ConcreteServices exhibits different values of QoS, and each is priced differently. Determining the best combination of services to achieve the application's overall target QoS, is an NP-Hard problem [3]. There is a further complication. As different jobs arrive, depending on the priority and QoS demanded, the application has to either scale up or down, on each of those QoS attributes. Not only does it have to re-calculate the best value-for-money services, but the set of ConcreteServices that are available, also change with time. Since these services (possibly) belong to third-parties, they may or may not be available or, are available with different QoS or, for a different price. Thus, the application has to deal with all of these factors, while adjusting to the changed demand for QoS. Self-Adaptation, in this case, is the problem of dynamically selecting services, from the pool of services currently available. There are two conditions where an application that has started off with a satisfactory level of QoS, needs to start adapting:

1) *External Adaptation Stimulus*: When a QoS violation is detected at any of the services, the application needs to search for a new service to replace the old one.
2) *Internal Adaptation Stimulus*: When the application's QoS target changes, the application needs to trigger an adaptation, to ensure that its constituent services are able to meet the target QoS level.

From the cloud provider's perspective, any service that could potentially be used by an application, but is idle instead, represents lost revenue. According to [9], in the future, the cloud provider will have to invest in self-managing resource management models that can adaptively service new QoS demands as well as existing QoS obligations. It is this scenario, of self-adaptive applications on one side, and dynamic service provision from the cloud provider on the other, that we address in this paper. Current static models of provisioning and pricing will prove inadequate, as self-adapting applications mature.

1.2 Contributions of this paper

In [38], we had proposed a preliminary design of a market-mechanism for service-based application to self-adapt to changing QoS requirements. In this paper, we explicate on the adaptation mechanism, its design in terms of implementation units, and its efficacy. In this paper, we also detail the adaptive generation of bids, and the decentralized mechanism for selection of services. We then show that this approach scales to hundreds of applications, with thousands of services being evaluated and selected. We show that the services thus selected, meet the QoS criteria of the application, and stay within budget.

2 Our Approach

We would like to create a mechanism that allows multiple applications, constructed across a federation of clouds, to self-adapt. We chose a market-based approach to self-adaptation, not only because it is decentralized, but also due to its easy applicability to the problem domain. Services in the cloud are moving from a fixed-price package to a more flexible, auction-based approach [29]. This enables a self-adaptive application to change the QoS exhibited, by switching to a different ConcreteService.

2.1 Market-Based Control

Market-Based Control (MBC) essentially involves modelling the system as a marketplace, where self-interested agents use economic strategies to compete for resources. Self-interested competition, along with well-designed utility functions, allow for a decentralized means of decision making. These agents, via their competitive need to get resources, perform a parallel search through the space of decision points. MBC has been used in several contexts, as a mechanism for computing a good solution in a decentralized manner. Notable examples include Clearwater's bidding agents to control the temperature of a building [16], Ho's center-free resource algorithms [52] and Cheriton's extension to operating systems to allow programs to bid for memory [25]. Wellman's WALRAS system [49], which is highly distributed, reports high scalability. More examples include distributed Monte-Carlo simulations [47], distributed database design using market-methods for distributing sub-parts of queries [45] and proportional-share resource management technique [48]. All of these systems provide evidence of market-based control being a good candidate for distributed decision making.

2.2 Auctions

Auctions, specifically Double Auctions (DA), have increasingly been studied in Computer Science, as a mechanism of resource allocation. Daniel Friedman [22] reports on experiments where traders even with imperfect information, consistently achieve highly efficient allocations and prices. The rise of electronic commerce naturally creates a space for efficient exchange of goods, and services. There has been much work on the design space of market-institutions [51], [39], bidding strategies [17], [43], agent-based implementations of traders [30], [26], [27], [40], etc. Gupta et al [24] argue that network management, specifically for QoS issues, must be done using pricing and market dynamics. According to them, the flexibility offered by pricing mechanisms offers benefits of decentralization of control, dynamic load management and effective allocation of priority to different QoS attributes. The continuous-time variant of a DA, called *Continuous Double Auction* (CDA), is used in stock-markets and commodity exchanges around the world [32]. In a CDA, the market clears *continuously*. That is, instead of waiting for all bids and asks to be made, matches are made as the bids and asks come in. A new bid is evaluated against the existing asks and the first ask that matches, is immediately paired off for a transaction. A CDA is known to be highly allocatively efficient [23], *i.e.*, it achieves a very high percentage of all the possible trades, between buyers and sellers. The most important property of the work in [23], is that a CDA's efficiency results from the structure of the mechanism used, rather than intelligence of the agents involved in trading. This is a very important result, since it provides us with encouragement regarding the efficiency of our mechanism.

2.3 Use of MDA for QoS adaptation

We view an application as a directed graph of AbstractServices. A ConcreteService corresponds to a piece of functionality described by the AbstractService, along with associated QoS levels. Thus, a data-mining application can be visualised as a directed graph of abstract data-filtering service, a data-transformation service, a clustering service and visualization service. The total QoS actually exhibited by the application, is a function of the individual QoS exhibited by each of the ConcreteServices that have been used in the composition. We design a marketplace that allows individual applications to select ConcreteServices. 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 central knowledge or coordination. A market populated with self-interested trading agents is therefore suitable for our purpose. The design of the market that we chose, is the CDA. However, we are mindful of Eymann's criticism [21] with regard to implicit centralization of auctions, and hence we create multiple double auctions (MDA), for each AbstractService. That is, each AbstractService is traded in multiple markets, and trading agents can move amongst these markets to find the ConcreteService they need. Each market is homogeneous, in the sense that all ConcreteServices trading in that market, pertain to one AbstractService only. This is analogous to commodity markets in the real world. Thus, in a market for clustering web-services, each seller provides a ConcreteService that performs clustering, but offers varying performance and dependability levels. Varying levels of QoS 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 QoS levels.

2.3.1 Description of Agents

BuyerAgent: A trading agent that is responsible for fulfilling one AbstractService. The BuyerAgent bids for, and buys a ConcreteService. The amount that the BuyerAgent is prepared to pay is called the *bid price* and this is necessarily *less than or equal to* its budget. The combination of *bid price* and the QoS attributes demanded, is called the *Bid*.
SellerAgent: A trading agent that sells a ConcreteService. A SellerAgent is responsible for only one ConcreteService. Based on demand and supply, the SellerAgent adjusts the price at which it is willing to sell the ConcreteService. The amount that a SellerAgent is prepared to accept, is called the *ask price*. This is necessarily *greater than or equal to* its cost. The combination of *ask price* and the QoS attributes offered, is called the *Ask*.
MarketAgent: A trading agent that implements trading rounds for a Market. It accepts Bids from BuyerAgents, and Asks from SellerAgents. It performs matching of Bids and Asks.
ApplicationAgent: An agent responsible for ensuring that an application meets its QoS requirements. It is responsible for distributing the budget, calculating achieved QoS, and instructing BuyerAgents to start/stop adaptation.

2.3.2 Structure of the Auction

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 the existing *Asks* in the orderbook, 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 QoS attributes of the Ask are *greater than or equal to* all the QoS attributes in the *Bid*. After a transaction, the corresponding *Bid* and *Ask* are cleared from the orderbook.

Table 2 shows the state of the orderbook at some time t_0. Maximizing the number of transactions would lead to

Bids	Asks
[B1, 107, ssl=yes, framerate=24fps, latency=99ms]	[S1, 97, ssl=yes, framerate=24fps, latency=99ms]
[B2, 105, ssl=yes, framerate=32fps, latency=105ms]	[S2, 98, ssl=no, framerate=24fps, latency=99ms]
[B3, 98, ssl=yes, framerate=24fps, latency=99ms]	[S3,103,ssl=yes,framerate=32fps,latency=105ms]
[B4, 91, ssl=yes, framerate=24fps, latency=105ms]	[S4, 105, ssl=yes, framerate=24fps, latency=99ms]
[B5, 87, ssl=yes, framerate=24fps, latency=110ms]	[S5, 110, ssl=no, framerate=32fps, latency=99ms]

TABLE 2: Orderbook at time t_0

Bids	Asks
[B1, 107, ssl=yes, framerate=24fps, latency=99ms]	[S1, 97, ssl=yes, framerate=24fps, latency=99ms]
[B2, 105, ssl=yes, framerate=32fps, latency=105ms]	[S2, 98, ssl=no, framerate=24fps, latency=99ms]
[B3, 98, ssl=yes, framerate=24fps, latency=99ms]	[S3, 103, ssl=yes, framerate=32fps, latency=105ms]
[B4, 91, ssl=yes, framerate=24fps, latency=105ms]	[S4, 105, ssl=yes, framerate=24fps, latency=99ms]
[B5, 87, ssl=yes, framerate=24fps, latency=110ms]	[S5, 110, ssl=no, framerate=32fps, latency=99ms]

TABLE 3: Orderbook at time t_1

a possible set like: $[B1-S4, B2-S3, B3-S1]$. Calculating this optimal set, however, quickly becomes infeasible as the number of Bids and Asks increase, since the number of pairwise comparisons increases exponentially. With a CDA, the set of transactions is: $[B1 - S1, B2 - S3]$. This is so because a CDA evaluates Bids and Asks, as they appear, and the first possible match is set up as a transaction. Thus, $[B1 - S1]$ is immediately matched and removed from the orderbook, and then $[B2 - S3]$ is matched. 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 (figure 3) yet. This procedure is much faster, and easily parallelizable. Although counter-intuitive, 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 [23]. Zero-Intelligence refers to a strategy, where the agents involved, do not consider any historical information about trades, and nor do they possess any learning mechanism. Thus, Zero-Intelligence marks the lower limit of the efficiency of a CDA market.

There are many variations on the implementation of a CDA, and each variation introduces changes in the behaviour of both, markets as well as the trading agents. We now list the structural axes of a CDA, and our position on each of those axes:

1) **Shout Accepting Rule:** Bids and Asks are referred to, as shouts. When a shout is received, the market evaluates it for validity. If valid, a shout is inserted in the appropriate place in the orderbook. The most commonly used shout accepting rule is the *NYSE rule*. According to the *NYSE rule*, a shout is only accepted, if it makes a better offer than that trader's previous offer [46]. We modify this rule to allow BuyerAgents to submit multiple Bids. This modification allows the BuyerAgents to explore the QoS-cost search space in a more efficient manner (see section 2.4).
2) **Information Revelation Rule:** This refers to the market information that is available to the BuyerAgents and the SellerAgents. In our case, all market participants have access to the last k transactions.
3) **Clearing Rule:** The market can either clear continuously, or at periodic time intervals. A market that clears with periodic time interval of 1 time unit is equivalent to clearing continuously. A market that clears with a time interval of greater than 1, is also called a Clearing House. As soon as an Ask meets all the QoS constraints specified in a Bid, and its ask-price $\leq$ bid-price, a potential transaction is identified. For a given Bid, all the Asks that could be potential transactions are called CandidateServices. The BuyerAgent is given the option of choosing one amongst the CandidateServices, while rejecting the rest. Again, this is a departure from typical CDAs, but essential to our mechanism (see section 2.5). Once a transaction takes place, the corresponding Bid and Ask are deleted from the orderbook.
4) **Pricing Rule:** This determines the price at which a transaction takes place. The most commonly used mechanism is *k-Pricing*. The value of *k* determines which entity makes more profit, the buyer or the seller. We use *k = 0.5* (*i.e.*, $0.5 * bid\_price + (1 - 0.5) * ask\_price$) as the transaction price.

2.3.3 QoS Calculation

There are three types of QoS constraints that we consider in our services:

1) *Numeric*: These constraints pertain to those QoS attributes that are either numeric inherently (e.g., Cost) or, are easily reducible to a scale such that numeric comparisons are possible (e.g. performance, reliability, etc.)
2) *Boolean*: Refers to those QoS attributes that are required to be definitely either present or absent (e.g., secure socket layer support = *yes/no*)
3) *Categoric*: Refers to those QoS attributes that are again required to be definitively picked out of a list, but may end up being more than one (e.g.,

possible values for framerate = 16fps, 24fps, 32fps, 48fps and acceptable ones are: 32fps and 48 fps)

We also allow each constraint to be tagged as *hard* or *soft*. This allows the application to indicate that it prefers a certain QoS attribute value, but that the value is not critical. For example, an application might specify a boolean QoS attribute, say *SSL support = yes*, as a soft constraint. This means that given a choice, it would prefer a CandidateService which has SSL support, but it is not an essential criterion. On the other hand, an application in the banking domain could specify SSL as a hard constraint, which means that a ConcreteService that does not support SSL, is not a CandidateService at all.

Stochastic Workflow Reduction: Depending on the pattern of the Workflow involved in the application, the same individual services with their corresponding QoS values, could result in differing end-to-end QoS exhibited by the application. To compute the QoS metrics for the entire application, we employ *Stochastic Workflow Reduction*(SWR) [11]. Using this method, the Workflow is viewed as a graph which is reduced using a series of reduction steps. Each time a reduction rule is applied, the structure of the graph changes, and the QoS metrics for the affected nodes are aggregated. The reduction continues in an iterative manner, until only one task remains in the graph. The QoS metrics aggregated for this task represents the QoS metrics corresponding to the application Workflow. In Figure 1, we see that a set of parallel tasks $t_1, t_2, t_3 ... tn$, a *split task (t_a)* and a *join task (t_b)* can be reduced to a sequence of three tasks, t_a, t_N, t_b. The incoming transitions of task t_a and the outgoing transitions of task t_b remain the same. The reduction for the QoS of the parallel tasks is computed as follows:

$$Cost(t_N) = \sum_{1 \leq i \leq n} Cost(t_i) \quad (1)$$

$$Reliability(t_N) = \prod_{1 \leq i \leq n} Reliability(t_i) \quad (2)$$

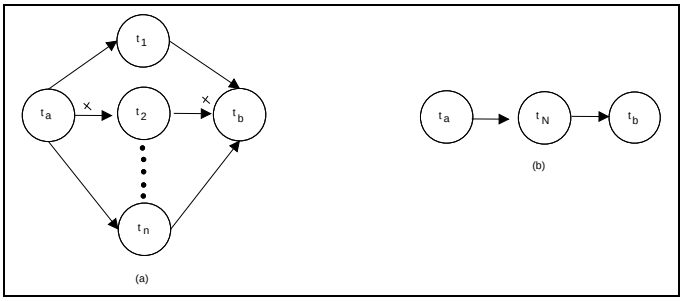

Fig. 1: Reduction of parallel tasks(reproduced from [12]). For additive QoS, the attributes are summed up, while for multiplicative QoS, the attributes are multiplied together

Thus, if the cost for each of the parallel tasks ($t_1 ... t_n$) is 5 units each, then the cost of t_N would be $5 * n$. Likewise, if the reliability for (say) three tasks ($t_1 ... t_3$) is 0.9 each, then the reliability for t_N would be 0.729. Similar reduction formulae [12] are used for other types of tasks, such as sequential tasks, loop tasks, conditional tasks, etc.

2.3.4 Decomposing End-to-End constraints

QoS constraints are usually specified on an application-wide basis. That is, performance for an entire application (end-to-end) or a cost constraint that must be met. To achieve these constraints in a decentralized manner, we decompose the end-to-end constraints, into local ones (the reverse process of SWR). Local constraints can can be handed over to individual BuyerAgents. Amongst the three type of QoS that we consider, Boolean and Categoric constraints do not need to be decomposed at all. They are given to all agents, as is. For example, if there is a Boolean constraint like: `For all services, SSL=yes`, then every agent can be given the constraint, without any processing. On the other hand, Numeric constraints like `total cost` $\leq C$ need to be decomposed on a per-Agent basis. Numeric constraints are decomposed, as shown in Algorithm 1. It may happen that none of the markets offer a service with the QoS needed to satisfy a particular application's constraints. In such a case, the mechanism terminates, and signals a failure. The ApplicationAgent does not take decisions about overriding constraints imposed by the application designer. We assume that the human designer modifies the constraints, or takes any other suitable action.

Algorithm 1: Decomposition of Numeric Constraints

Data: Numeric Constraints Limit
Result: Numeric Constraints Decomposed
begin
  foreach *Agent* a **do**
    Find list of markets (M) corresponding to a_{f_x}
    Choose $m \in M$
    Register in m
    $a \longleftarrow m_{last\_k\_transactions}$
    foreach $\omega \in NumericConstraints$ **do**
      calculate $a_{low}^{\omega\_m}, a_{median}^{\omega\_m}, a_{high}^{\omega\_m}$
  foreach $\omega \in NumericConstraints$ **do**
    Apply SWR $\longleftarrow \langle \omega^{f_1}, \omega^{f_2}, \omega^{f_3} \ldots \rangle$
    if $constraintMet$ **then**
      foreach *Agent* a **do**
        $a_{bid}^{\omega} \propto SWR^{\omega}$
    Choose different m
    if $allMarketsExhausted$ **then**
      TerminateWithFailure

2.4 Adaptation Using Bid Generation

Bids are the mechanism by which BuyerAgents explore the search space of QoS-cost combinations that are available. Depending on the *bids*, potential transactional matches are identified by the MarketAgent. Thus, there needs to be a systematic way to generate Bids. The relevant aspects in a Bid are given in Table 4.

The most important part of deciding on a value for a QoS attribute, is the constraint on each attribute. If the constraint is hard, then, in each of the generated Bids, the value inserted into the Bid will remain the same. Else, the value inserted into the Bid is varied. The varied values depend on the type of QoS attribute. The rules for generating QoS values, on the basis of type of attribute are as follows:

Boolean: In case of soft constraints, two Bids will be generated for a particular QoS attribute. For example, for QoS attribute *SSL support*, one Bid is generated with the value: *yes* and another with the value: *no*

Categoric: In case of soft constraints, the number of Bids will depend on the number of acceptable alternatives. For example, for QoS attribute framerate, the acceptable alternatives are: *24fps* and *32fps*. Therefore, two Bids will be generated.

Numeric: In case of soft constraints, the number of Bids will be three. The BuyerAgent makes one Bid at its target value, one at the median value obtained from the market, and the third Bid at the mid-point between the target and the median value.

Thus, the total number of Bids that a BuyerAgent generates is given by:

$$\begin{aligned} Num(buyer_{bids}) =& 2 * Num(BooleanQoS) \\ & * Num(AcceptableAlternatives) \\ & * 3 * Num(NumericQoS) \end{aligned} \quad (3)$$

Thus, if a BuyerAgent has the QoS and preferences as given in Table 5:

The total number of bids that it would generate would be:

$$\begin{aligned} Num(buyer_{bids}) &= 2 * Num(SSLSupport) \\ & * Num(FrameRate) \\ & * 3 * Num(Latency) \\ &= 2 * 2 * 3 \\ &= 12 \end{aligned}$$

2.5 Decentralized Decision-Making Using Ask-Selection

According to the market mechanism that we have outlined, a BuyerAgent can make multiple Bids. This is how it searches the space of possible services, for its stated budget. There might be multiple Asks in the market that match these multiple Bids. To deal with this, the market mechanism moves on to its second-stage, where the BuyerAgent has to select amongst the CandidateServices. Since the mechanism for Bid-generation precludes Bids that will violate hard constraints, none of the CandidateServices will violate the application's hard constraints either. The task facing the BuyerAgent, is to choose amongst the possible transactions, in such a way that the decision maximizes the possible benefits, while taking care to minimize the possible downsides. We use a multiple criteria decision making approach called PROMETHEE [6]

The BuyerAgent needs to rank all the CandidateServices, that have been returned by the MarketAgent as a potential transaction. Thus, for a transaction set (TS), the BuyerAgent needs to select one CandidateService that it will actually transact with. The preference of CandidateService a over CandidateService b is calculated by aggregating the preferences over each QoS attribute (and its weightage). The preference value (π) of a CandidateService a over CandidateService b is calculated as:

$$\pi(a,b) = \sum_{i=1}^{n} P_i(a,b) w_i \quad (4)$$

where P is a preference function over a QoS attribute (see example in equations 8, 9 and 10). Using this preference value, each CandidateService is ranked *vis-a-vis* the other CandidateServices in the possible transaction set. Ranking is done by calculating the positive outranking ($\emptyset^+$) and the negative outranking ($\emptyset^-$):

$$\emptyset^+(a) = \sum_{x \in TS} \pi(a,x) \quad (5)$$

$$\emptyset^-(a) = \sum_{x \in TS} \pi(x,a) \quad (6)$$

Finally, the net outranking ($\emptyset$) is created to get a complete ranking of all the CandidateServices:

$$\emptyset(a) = \emptyset^+(a) - \emptyset^-(a) \quad (7)$$

Choice of the CandidateService is based on the following heuristic:

1) Based on the net outranking flow, choose the best CandidateService.
2) If there is a tie, choose the cheaper CandidateService out of the pool of best CandidateServices
3) If using cost still leads to a tie, choose randomly amongst pool of best CandidateServices

2.6 A Worked-out Example

Suppose that the MarketAgent returns the 4 Asks (as shown in Table 7), as possible transaction matches.

Ranking amongst multiple asks is done on a per-QoS attribute basis. That is, each ask is ranked on each QoS attribute.

This effectively means that CandidateServices will be evaluated on their QoS attributes as follows:

1) In case of a boolean attribute, a CandidateService A will be preferred over CandidateService B, if

Category	Value
Types of QoS attributes	Boolean, Categoric and Numeric
Constraint on each attribute	Hard, Soft
Direction on each attribute	Maximize, Minimize, N.A.
Bid_Price	Numeric

TABLE 4: Elements in a Bid

Name	**Type**	**Median**	**Target**	**Direction**
SSL Support	Boolean	No	Yes/No	N.A.
FrameRate	Categoric	24fps	24fps, 32fps	Maximize
Latency	Numeric	99ms	95ms	Minimize
Budget	Hard Constraint	(From Market)	100	N.A.

TABLE 5: QoS preferences for a BuyerAgent

Attribute	Value
Bid-Price	80
SSL	Yes
Framerate	24fps
Latency	99

Attribute	Value
Bid-Price	80
SSL	No
Framerate	32fps
Latency	99

TABLE 6: Sample Bids

Attribute	Value
Ask-Price	74
SSL	Yes
Framerate	24fps
Latency	99

(a) CandidateService A

Attribute	Value
Ask-Price	78
SSL	Yes
Framerate	24fps
Latency	95

(b) CandidateService B

Attribute	Value
Ask-Price	80
SSL	No
Framerate	24fps
Latency	90

(c) CandidateService C

Attribute	Value
Ask-Price	76
SSL	Yes
Framerate	32fps
Latency	92

(d) CandidateService D

TABLE 7: Asks returned by MarketAgent as potential transactions

it has a desired boolean value. Else, there is no preference.

2) In case of a categoric attribute, a CandidateService A will be preferred over CandidateService B, if the categoric value of A is better by k over the categoric value of B. Else, there is no preference. The value of k is adjusted based on whether the attribute is a hard constraint or a soft constraint. For hard constraints, k is taken to be zero, *i.e.*, the CandidateService with the better categoric value is strictly preferred over the other.
3) In case of a numeric attribute, a CandidateService A will be increasingly preferred over CandidateService B, as the difference between their numeric value approaches some m. After m, A is strictly preferred. Again, the value of m is adjusted based on whether the attribute is a hard constraint or a soft constraint.

The preference functions for the three QoS attributes can be given as follows:

$$P_{ssl}(x) = \begin{cases} 0 & \text{if } x = (ssl = no), \\ 1 & \text{if } x = (ssl = yes) \end{cases} \tag{8}$$

$$P_{framerate}(x) = \begin{cases} \frac{1}{32}x & \text{if } x < 32fps, \\ 1 & \text{if } x \geq 32fps \end{cases} \tag{9}$$

$$P_{latency}(x) = \begin{cases} 0 & \text{if } x > 103ms, \\ \frac{1}{2} & \text{if } 103 > x > 95ms, \\ 1 & \text{if } x \leq 95 \end{cases} \tag{10}$$

Based on the preference functions given (in 8, 9 and 10), we can calculate the relative values of the 4 CandidateServices, as given Table 9 Once we have a relative

TABLE 8: Values of $\pi(x_i, x_j)$

	A	B	C	D
A	—	0 + 0 + (-0.5)	1 + 0 + (-0.5)	0 + (-0.25) + (-0.5)
B	0 + 0 + 0.5	—	1 + 0 + 0	0 + (-0.25) + 0
C	(-1) + 0 + 0.5	(-1) + 0 + 0	—	(-1) + (-0.25) + 0
D	0 + 0.25 + 0.5	0 + 0.25 + 0	1 + 0.25 + 0	—

per-attribute value, we can calculate the outranking values based on equations 5, 6 and 7 (see Table 9).

	A	B	C	D
ϕ^+	0.5	1.5	0	2.25
ϕ^-	1.25	0.25	2.75	0
ϕ	-0.75	1.25	-2.75	2.25

TABLE 9: Calculation of outranking values

It is clear from Table 9, that CandidateService D is the best amongst the potential transactions, and CandidateService C is the worst. The BuyerAgent now accepts the transaction with CandidateService D, and rejects all the others.

2.7 Post-Transaction

The BuyerAgent reports back to the ApplicationAgent, the cost and QoS being made available for the transaction. The ApplicationAgent then performs a SWR calculation to ensure that the QoS of the service being bought, does not violate any of the application's end-to-end constraints. Note that the ApplicationAgent needs to calculate SWR for the numeric constraints only. The other types of constraints (boolean and categoric) are specified in the bid prepared by the BuyerAgent, and therefore do not need checking. This greatly reduces the computational effort required on the part of the Application agent. The best case scenario, from a computational perspective, is when all the QoS attributes of the Application are either boolean or categoric. However, in the worst case scenario, all the QoS attributes could be numeric. In this case, it has to perform an SWR calculation for each of the QoS attributes and the distribution of computation to BuyerAgents is minimal.

In Figure 2, we show the various steps in the CDA as activities carried out by the principal agents.

2.8 Re-starting Conditions

Once an application reaches a satisfactory level of QoS, all its agents will withdraw from the market. The agents will re-enter the market, only in case of an **adaptation stimuli**. There are two kinds of adaptation stimuli:

1) External Stimuli: When a QoS violation is detected at any of the CandidateService, the ApplicationAgent informs the BuyerAgent that it needs to restart its search for a new CandidateService. This event is not propagated to all the BuyerAgent, but only to the particular one responsible for that AbstractService.
2) Internal Stimuli: When the application's end-to-end QoS target or available budget changes, the ApplicationAgent informs all of its BuyerAgents, and restarts the adaptation. The agents stay in the adaptation phase, until the stopping criteria are met.

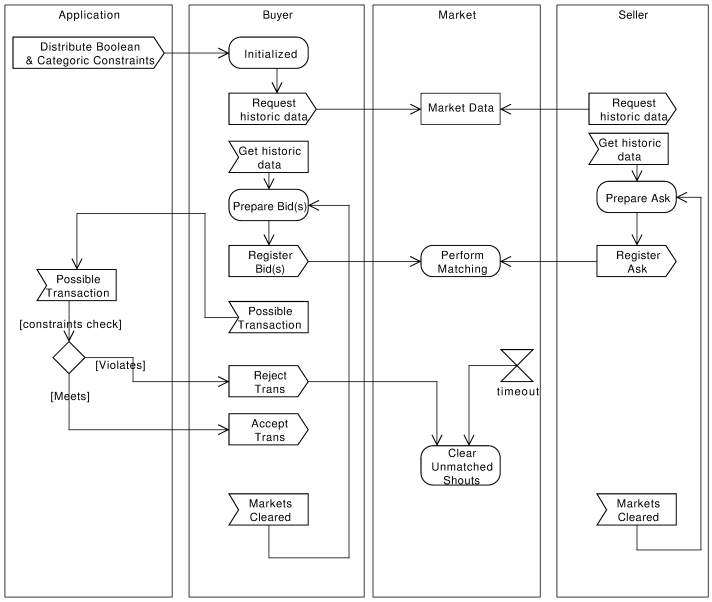

Fig. 2: Activity diagram for principal agents

Both these stimuli occur at different time scales. The external check for QoS violation is a continuous check and concerns itself with performance of the web-service across relatively shorter periods of time. The internal stimulus, on the other hand, typically happens when budgets for operation change drastically or external events cause change in performance or reliability requirements. This happens at relatively rarer intervals. Hence, typically once an application reaches close to its desired level of QoS, the BuyerAgents stop trading.

2.9 Design and Implementation

We call our implementation of the proposed mechanism, *clobmas* (Cloud-based Multi-Agent System). We document *clobmas* using the object-oriented paradigm for all of the internal components of an agent. Thus, we show the static structure of the system through package, component and class diagrams. We show agent-interaction using Agent-UML (AUML). We envision our mechanism as a middleware, between service-based applications and multiple SaaS clouds. In Figure 3, we show clobmas as a middleware between two OpenStack-based clouds and SOA-based applications. Note that OpenStack[1] is currently an IaaS cloud solution. We envision the addition of a UDDI-based Service Registry communicating with the Cloud Controller, to enable OpenStack to serve services as well.

2.9.1 Architectural Style

The design of clobmas is based on the publish-subscribe architectural style. This style was chosen for its de-

1. www.openstack.org

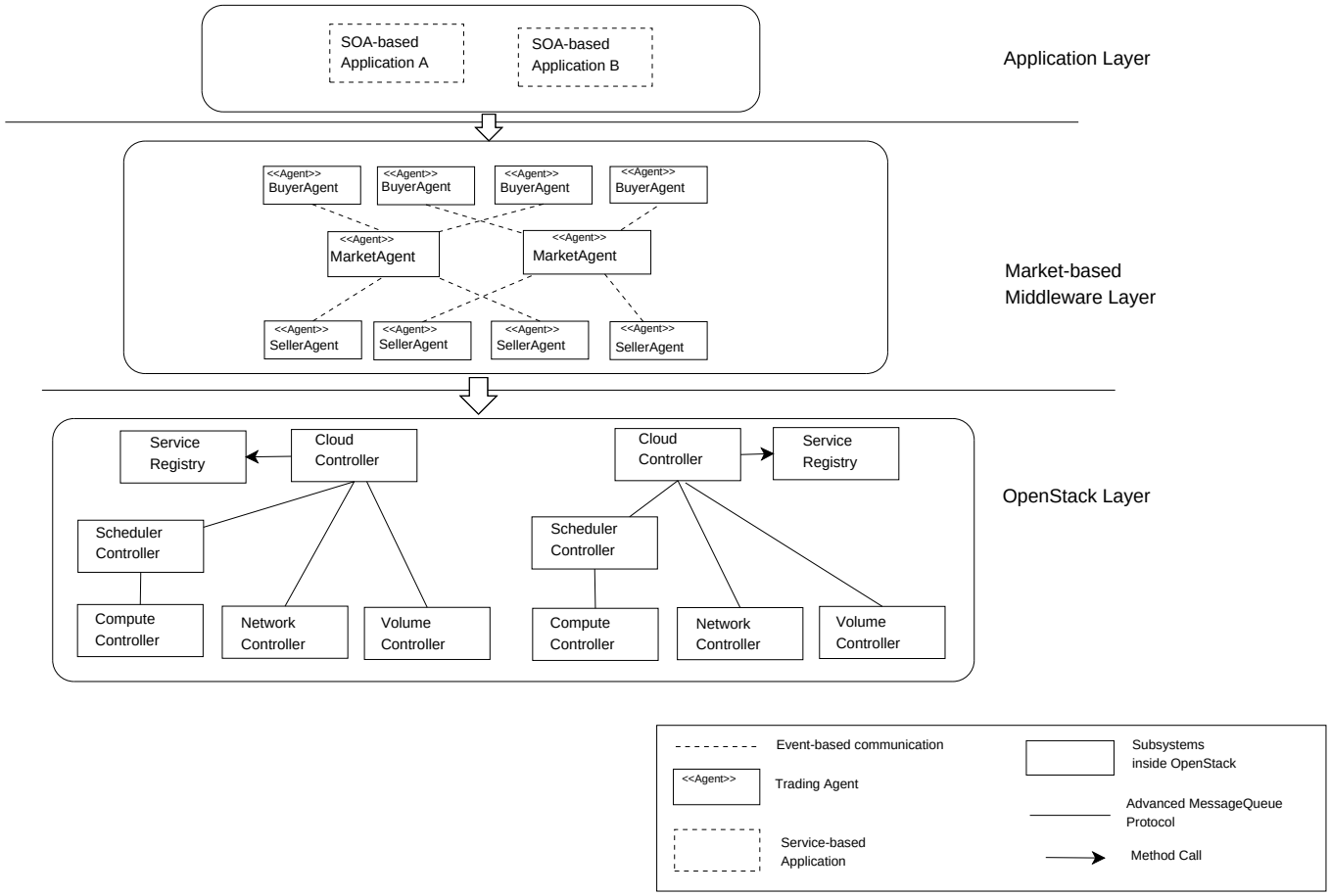

Fig. 3: Clobmas as middleware between SOA-based applications and two OpenStack-based clouds

coupling properties, such that each communicating entity in the system is independent of every other. This fits in well, with our notion of agents acting on the basis of their own goals and their environment, in a decentralized manner. Each type of agent is both an event publisher and an event subscriber. That is, each ApplicationAgent generates events that describe when the application's QoS constraints have changed, when the budget has changed, etc. These events are subscribed to, by BuyerAgents belonging to that application. The BuyerAgents, for their part, publish events relating to the market that they are registered with. Similarly, the MarketAgents publish events relating to the start of trading, transaction events, etc. The event-based paradigm fits well with the QoS monitoring schemes, as well. Both Zeng [55] and Michlmayer [37] use events to monitor and communicate SLA violations.

2.9.2 Class and Package Diagram

The class diagram shows the static relationships between the main entities in clobmas. The *Bid* and *Ask* interfaces shown in Figure 4 are implemented by the BuyerAgent and SellerAgent respectively, with the *MarketAgent* implementing both of them. The packages with the stereotype "subsystems", marked as Nova and Glance, belong to OpenStack.

2.9.3 Activity Diagram

The most difficult part of getting all the agents in an MAS to solve a problem, is the problem of communication. The design of the communication protocol determines how much communication happens, at what times and how much computational effort it takes to communicate. If the agents communicate too little, then there is a danger of the MAS failing to solve the problem it was created for. On the other hand, if too much communication takes place, then a lot of wastage occurs, not only in terms of bandwidth but also in computational cycles and time. Thus, while communication is a fundamental activity in an MAS, depicting this in UML is difficult. We document the communication steps of our mechanism through Agent-UML (AUML) diagrams.
In the following figures, we show the communication for an application with two AbstractServices, A and B, and its corresponding BuyerAgents and the MarketAgent. The *Setup Phase* (Figure 5) is composed of two calculations done by the ApplicationAgent and, two multicast messages to the application's BuyerAgents. The two calculations involve decomposing the application's end-to-end constraints into local constraints and, computing the budget that each BuyerAgent will receive.

More figures showing the various scenarios of com-

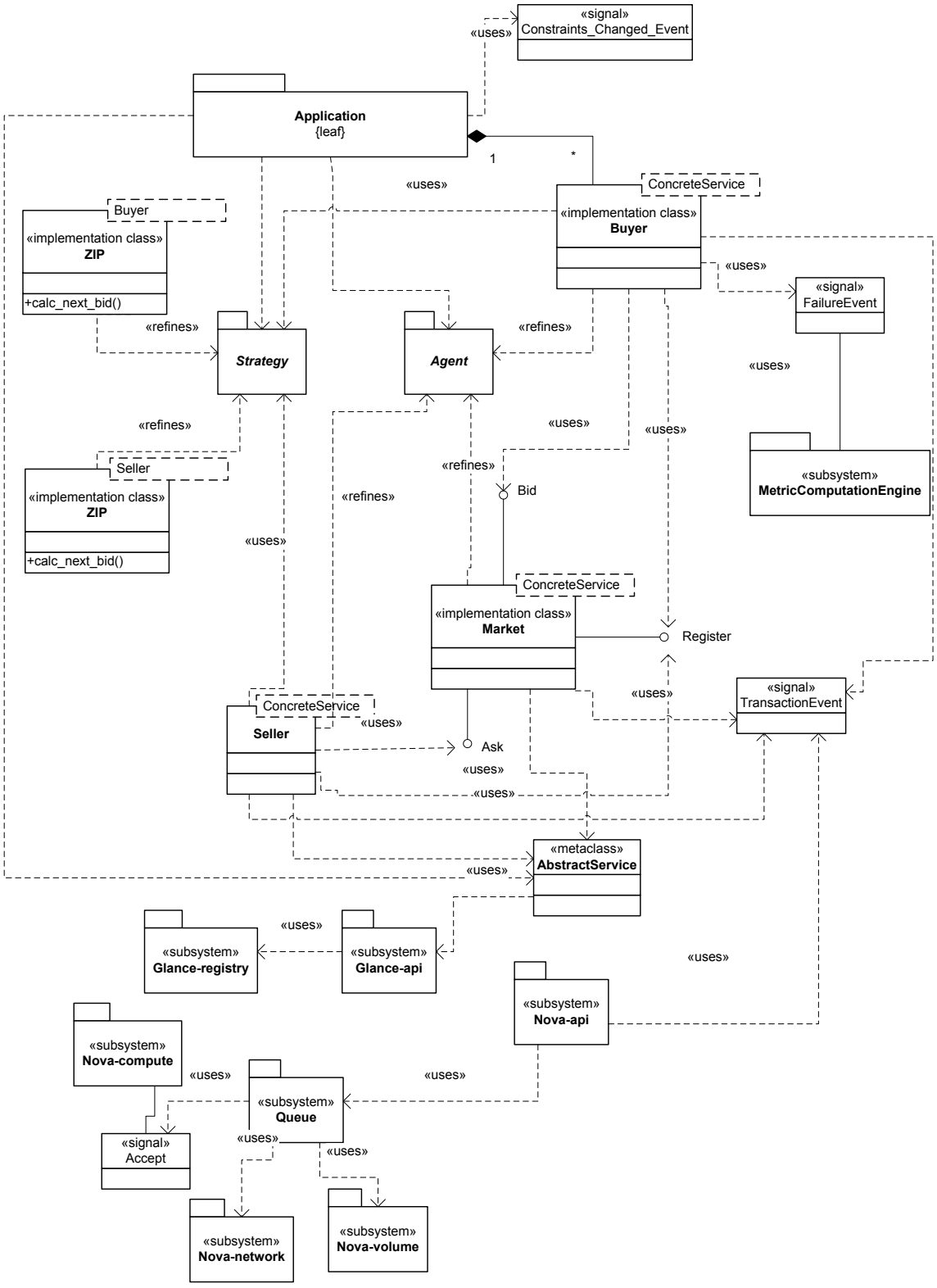

Fig. 4: Class and package diagram of clobmas and OpenStack subsystems

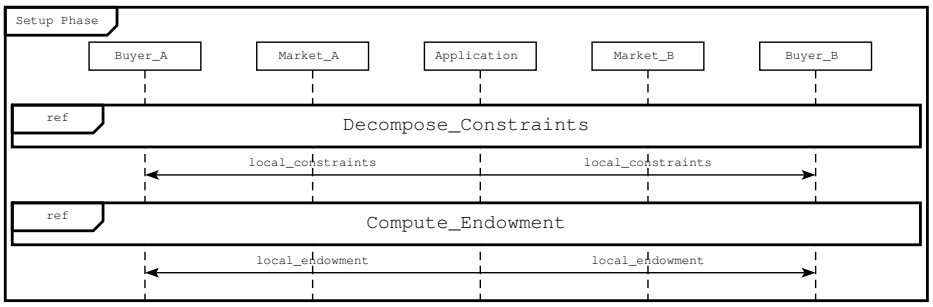

Fig. 5: Setup phase for an application with two AbstractServices A & B

munication between the agents are shown in Appendix A.

2.10 Monitoring

Monitoring the actual QoS exhibited during runtime by the CandidateService, is beyond the scope of the system. We assume that all the agents agree on the monitoring of a CandidateService, by a third-party mechanism. This could be market-specific or domain specific. The monitoring of QoS cannot be done by our mechanism, since it needs to be neutral, and trusted by both parties: BuyerAgent and SellerAgent. The BuyerAgent needs to know whether the SellerAgent's service is providing the QoS that it promised, and the SellerAgent needs to know that the BuyerAgent's application is not abusing the service.

3 Evaluation

3.1 The Current Scenario

The current form of service-selection is provider-driven. That is, in all commercial clouds, the cloud provider uses a **posted-offer** mechanism. A posted-offer is a form of market where the supplier posts a certain price on a *take-it-or-leave-it* basis. Thus, on Amazon's *elastic cloud compute* (EC2), there are several services that are functionally identical, but priced differently. This price differentiation exists due to different QoS being exhibited by these services. In Table 10, we show a slice of Amazon's pricing for its *On-Demand Instances*. Depending on the type of job envisioned, customers purchase a basket of computational power from Amazon. However, currently, there is no mechanism to automatically switch from one kind of *On-Demand Instance* to another. Customers have to forecast the type of demand for their application in advance, and appropriately chose their package from Amazon. Any application that desires to use a particular service, has to pay the posted price. There exists no mechanism to negotiate/bargain with Amazon, on pricing or QoS of the services being offered. This has the very obvious effect of customers either over-provisioning or under-provisioning for their actual demand. If an application under-provisions, then it risks losing customers due to lack of QoS. On the other hand, if it over-provisions, it loses money due to lack of productive use. In both cases, the customer faces a loss.

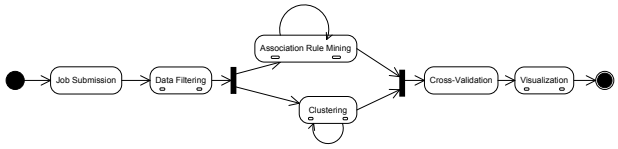

Fig. 6: BizInt's workflow constructed using composite services from the hosting cloud

3.2 Empirical Study

BizInt, a small (fictional) startup company creates a new business intelligence mining and visualization application. It combines off-the-shelf clustering algorithms with its proprietary outlier detection and visualization algorithms, to present a unique view of a company's customer and competitor ecosystem. In order to exhibit a high level of performance, it decides to host its application in the cloud. Also, instead of reinventing the wheel, it uses third-party services (for clustering, etc.) that are also hosted in the same cloud. As seen in figure 6, BizInt uses composite web services (Data Filtering, Clustering, Association Rule Mining and Cross-Validation) from the cloud, along with its own services (Job Submission, Outlier Detection and Visualization) to create a complete application. Soon BizInt discovers that different jobs emphasize different QoS. Some jobs want data to be processed as fast as possible, others require a high amount of security and reliability. In order to exhibit different QoS, BizInt needs to dynamically change its constituent services.

SkyCompute is a new (fictional) entrant to the field of Cloud Computing. It wants to compete with Amazon, 3Tera, Google, Microsoft and other established cloud-providers. In order to attract cost and QoS-conscious customers, SkyCompute will have to differentiate its cloud from the others. It plans to target the *Software-As-A-Service* market. Instead of providing specialist infrastructural services (like Amazon) or application frame-

	Linux/UNIX usage	Windows usage
Standard On-Demand Instances		
Small (default)	$0.085 per hour	$0.12 per hour
Large	$0.34 per hour	$0.48 per hour
Extra Large	$0.68 per hour	$0.96 per hour
Micro On-Demand Instances		
Micro	$0.02 per hour	$0.03 per hour
Hi-Memory On-Demand Instances		
Extra Large	$0.50 per hour	$0.62 per hour
Double Extra Large	$1.00 per hour	$1.24 per hour
Quadruple Extra Large	$2.00 per hour	$2.48 per hour

TABLE 10: On-Demand Instance Pricing on Amazon EC2

work services (like Google and Microsoft),it is planning to provide generically useful services like indexing, clustering, sorting, etc. Like most cloud providers, it plans to provide services with different QoS levels, so that multiple types of clients can be attracted to use it. To differentiate itself, SkyCompute plans to provide an adaptive framework, so that companies like BizInt can change their constituent services, dynamically.

3.3 Qualitative Criteria

Thus, *clobmas* must fulfill the following criteria:

1) Allows customers like BizInt to create adaptive applications
2) Generates a higher utilization of services than the posted-offer model currently followed (for SkyCompute)

3.4 Quantitative Criteria

Since, SkyCompute is an ultra-large collection of services, clobmas must be able to scale to large numbers of applications and ConcreteServices. Since there is no public data about the kinds of workflows hosted on commercial clouds, and their corresponding service choices, we made assumptions about the variables involved in dynamic service composition. We make these assumptions based on conversations with performance consultants at Capacitas Inc., and numbers gleaned from the literature review.

We summarize the operational ranges that clobmas is expected to deal with, in Table 11

Variable affecting performance	From Lit. Review	Target Goals
Number of AbstractServices	10	20
Number of CandidateServices	20	50
Number of QoS attributes	3	10
Number of markets	1	10

TABLE 11: Operational range for scalability goals

3.5 Experimental Setup

Although open-source cloud implementations like OpenStack[2], and Eucalyptus[3] are freely available for download, they are primarily targetted at the *Infrastructure-As-A-Service* market. Modifying these implementations for our purposes (to introduce the notion of markets, respond to transaction events, etc.), would be expensive in terms of time. In the same vein, using simulators such as CloudSim would require major modifications, to those toolkits. CloudSim is a fairly new toolkit, and does not have the ability to explore market mechanisms in a sophisticated manner [7].

Software: In this scenario, we wrote our own simulator in Python (v. 2.6.5), using a discrete-event simulation library, SimPy[4]. The operating system in use, was 64-bit Scientific Linux.

Hardware: All experiments were run on an Intel Dual-cpu Quad-Core 1.5Ghz workstation, with 4MB of level-1 cache and 2GB of RAM.

Generating Randomness: We use *Monte Carlo sampling* to draw QoS values for each of the services in the market and for the QoS values demanded by the applications. A critical factor in ensuring the goodness of sampling used for simulations, is the goodness of the pseudo-random number generator (PRNG). We use the *Mersenne Twister*, that is known to be a generator of very high-quality pseudo-random numbers [36]. We use this generator due to the fact it was designed with Monte Carlo and other statistical simulations in mind. [5]

Reporting: All results are reported as an average of a 100 simulation runs. Each simulation reported, unless otherwise specified, was run with the following parameters given in Table 12. In the section on scalability (subsection 3.8), we stress the mechanism by scaling up variables to the target goals given in Table 11.

System Parameter	Value
AbstractServices in Workflow	10
CandidateServices per AbstractService	20
QoS attributes	3
Number of markets per CandidateService	1
Tolerance Level (for numeric QoS)	0.1
Applications simultaneously adapting	300

TABLE 12: System parameters and their standard values

2. www.openstack.org
3. http://www.eucalyptus.com/
4. http://simpy.sourceforge.net/
5. For a k-bit word length, the Mersenne Twister generates an almost uniform distribution in the range $[0, 2^k - 1]$

3.6 Methodology

We simulate 300 applications, each application consisting of randomly generated workflows trying to achieve their desired QoS levels, within a given budget. The budget that each application gets, acts as a constraint and the application's constituent BuyerAgents can never bid above their budgets. Each application generates a random level of QoS that it must achieve. Once an application achieves its required QoS, it withdraws from trading until an internal or external stimulus occurs. We model the probability of a QoS service violation based on a study by Cloud Harmony [1].

3.7 Results

There are two perspectives from which to evaluate our mechanism, BizInt's and SkyCompute's perspective. We present both perspectives, with an emphasis on SkyCompute's view since it provides a better idea on the scalability of our mechanism [6]

3.7.1 BizInt's Perspective

From BizInt's perspective, a cloud provider that enabled applications to self-adapt at the QoS level would be ideal. BizInt understands that a typical adaptation process does not guarantee an optimal result. But, even so, a self-adaptive mechanism would be a step-up from the current mechanism of manually selecting the service to be used. It evaluates the efficacy of the adaptive mechanism by two measures:

1) How close to the desired QoS level does the adaptive mechanism reach?
2) How long does it take for an application to achieve its desired QoS?

Close to desired QoS: Like most research on dynamic service composition, we evaluate the goodness of a particular set of services, by means of a utility function. We use the application's targetted end-to-end QoS as the benchmark, against which the currently achieved QoS is measured. The application's target QoS is normalized and summed across all the QoS that it is interested in. This is taken to be the *ideal utility level*. The achieved QoS is fed through the same process, to attain the *achieved utility level*. Calculation of utility is done by summing up the value(s) of QoS. Hence, if w denotes the weight for a particular QoS, K be the set of QoS, and $V(x)$ be the value function, then:

$$Utility = \sum_{k=1}^{K} w_k * V(k) \tag{11}$$

where

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

6. Code for the simulation available at: http://www.cs.bham.ac.uk/~vxn851/multiple-DA-sim.tar.bz2

The difference between the *ideal utility level* and *achieved utility level* is called the *quality gap*. Each application defines a tolerance level, that specifies the magnitude of quality gap that is acceptable. **If the quality gap is within the tolerance level, the application is said to be satisfied**. We measure over a trading period, the **number of rounds that the application is satisfied**.

Time to Satisfaction: In the absence of a deterministic solution (and therefore a deterministic notion of time), we must contend ourselves with evaluating whether the adaptive algorithm is *'fast-enough'*. The natural unit of time in clobmas, is a *trading round*. We measure **how many trading rounds**, application takes to be satisfied, from the start of an adaptation event. An adaptation event occurs when an application notices that the *quality gap* is beyond the tolerance level. Thus, the start of the simulation defines an adaptation event, since all applications start off without any *achieved utility level*.

In Figure 7, we see an average application's achievement of QoS levels. The application starts off with a *quality gap* of *minus 3*. But it quickly adapts to the QoS demanded and reaches the tolerance zone, in about 20 trading rounds. From this point on, the application stays satisfied. The application is satisfied for 280 out of 300 trading rounds. We see that the achieved utility level does not stay constant. Instead, internal adaptation events cause the application to seek out different services. However, it stays within the tolerance zone. In Figure 7, we show the adaptation occurring under conditions of normal distribution of QoS demand and supply in the cloud.

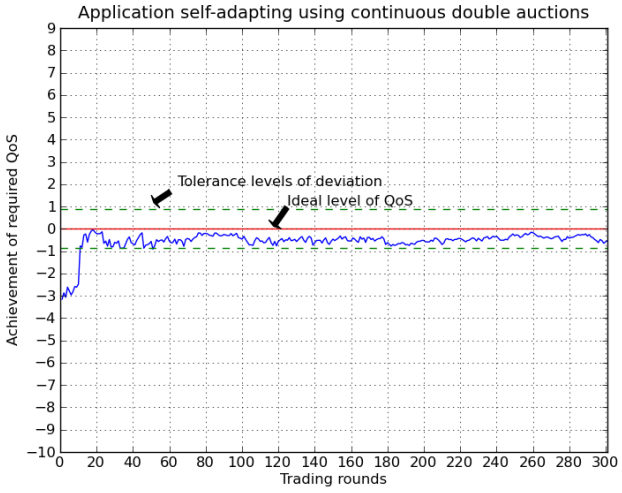

Fig. 7: Utility gained by adaptation by a single application

To compare the goodness of trading in a CDA market, we contrast its use in a posted-offer market.

Posted Offer: This type of mechanism refers to a situation where a seller posts an offer of a certain good at a certain price, but does not negotiate on either. That is,

the buyer is free to *take-it-or-leave-it*. In such a scenario, the probability that an application will find services, at the *bid-price* decreases.

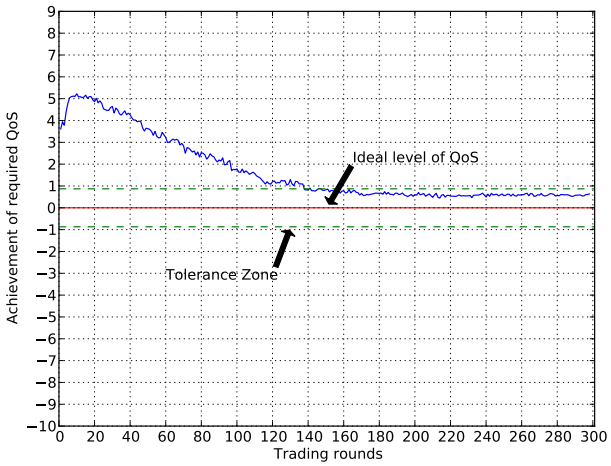

Fig. 8: Utility gained by a single application in a posted offer market

In Figure 8, we see that the application is able to adapt and acquire the QoS that it requires. It is obvious from the two figures, the application is able to reach the tolerance zone a lot quicker in Figure 7, than in Figure 8. In fact, in the *posted offer* market, we see that the application takes 140 rounds to reach the tolerance zone, while in the *CDA* market, the application is able to reach its tolerance zone in under 20 rounds of trading. This difference can be attributed to the fact that since the sellers in a *posted-offer* market do not change their price, the buying agents have to search a lot longer to find sellers that they are able to trade with. In a *CDA*, the search process is much faster, since both the buyers and the sellers adjust their *Bids* and *Asks*.

	CDA	PostedOffer
Number of rounds satisfied	282	160
Time to reach QoS	18	140

TABLE 13: Comparative performance of adaptation in CDA vis-a-vis Posted Offer

3.7.2 SkyCompute's Perspective

SkyCompute would like to ensure that any mechanism that it offers for self-adaptation results in high utilization of its services. We define **market satisfaction rate** as a measure of the efficiency of the mechanism. **Market satisfaction rate** is the percentage of applications that are satisfied with their QoS achievement. A satisfied application would withdraw from trading and use the service for a relatively long period of time. The more the number of applications that are satisfied, the more the number of services from SkyCompute that are being utilized. Thus, **market satisfaction rate** is a measure that SkyCompute would like to maximize.

As a baseline, we first implement the Zero-Intelligence mechanism, as this represents the lower limit of the effectiveness of the CDA mechanism. The Zero-Intelligence scheme consists of agents randomly making bids and asks, with no history or learning or feedback(see figure 9). As expected, it performs quite poorly, with a market satisfaction rate of about 10-20%. Clobmas, on the other hand, achieves a much higher rate of market satisfaction(see Figure 10).

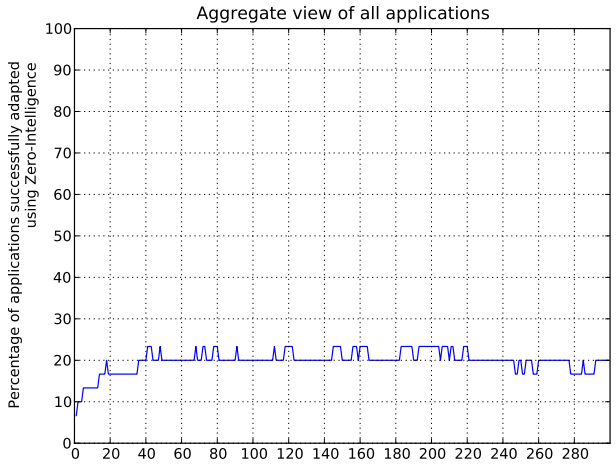

Fig. 9: Efficiency of adaptation in a CDA market with *Zero Intelligence*

On the other hand, we see (in figure 10) that with adaptive bids, the number of applications that are able to adapt rise to about 85% of the total market. This is a huge improvement with a very small improvement in the intelligence of the agents.

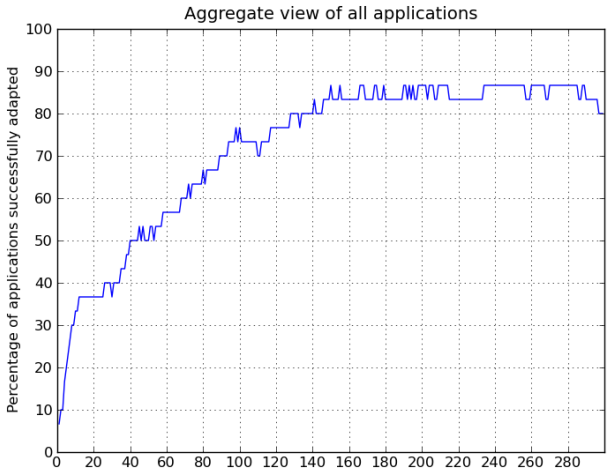

Fig. 10: Efficiency of adaptation in a CDA market

Quite similar to the CDA, the posted-offer also performs well (approx 70% market satisfaction rate). In this case, the sellers *never* adjust their prices, but the buyers do.

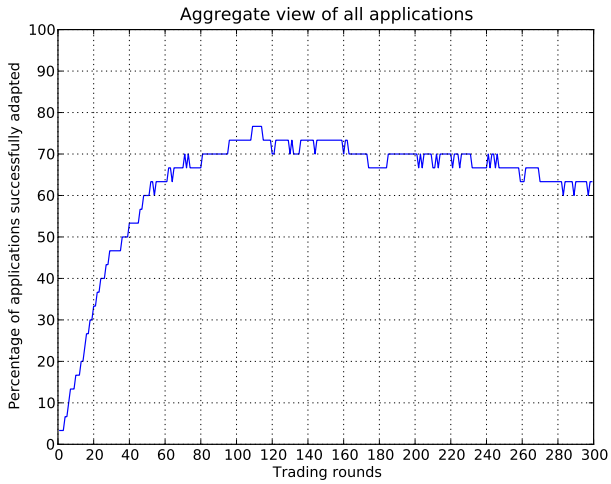

Fig. 11: Efficiency of adaptation in a Posted Offer market

3.8 Scalability Requirements for SkyCompute

There are no commonly accepted/standard definitions of scalability, in self-adaptive, multi-agent systems. To alleviate this, we use a definition of scalability from [20]: *a quality of software systems characterized by the causal impact that scaling aspects of the system environment and design have on certain measured system qualities as these aspects are varied over expected operational ranges*.

According to this definition, when aspects of the environment change, they have an impact on the system qualities. In our case, the system quality that we want to measure is performance, measured as the amount of time taken to reach a defined level of *market satisfaction rate*. Although clobmas using a double-auction is better than clobmas using a posted offer, SkyCompute would like to know if this mechanism scales well. To investigate this, we pick a level of market satisfaction rate, that is higher than posted-offer, and measure how long it takes to achieve this level. The adaptation process can be highly dependent on many variables. In this section, we tease out how each of these variables affect the time taken for adaptation. To this end, we present the time taken by the CDA market to achieve an **80% market satisfaction rate**, while varying each of these variables.

AbstractServices Vs. CandidateServices: Arguably, these are the variables that change most often from application to application. Every application has a different workflow and therefore, a different number of AbstractServices. As time passes, applications that have met their QoS will retire from the market, and new applications will come in. This changes the orderbook from the demand side. Also, the number of CandidateServices is most susceptible to change. As time passes, some CandidateServices will no longer be available, and new ones come into the market. This changes the orderbook from the supply side.

We perform curve-fitting to analyze the growth of

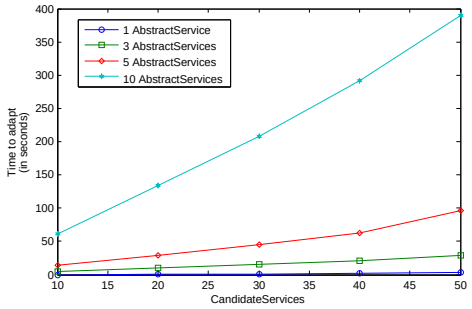

Fig. 12: CandidateServices increase

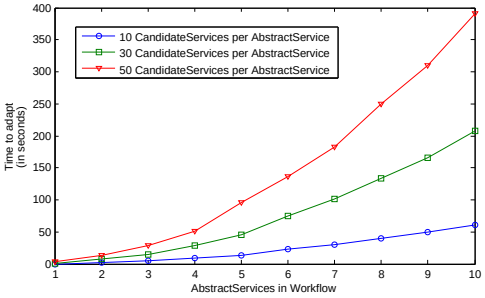

Fig. 13: AbstractServices increase in Workflow

the graphs. In the worst case, the polynomial growth of Time-to-MSR(y) when AbstractServices(x) increase, is given by

$$y = -0.0361x^3 + 4.7619x^2 - 5.4985x + 3.8978 \quad (12)$$

Again, in the worst case, the polynomial growth of Time-to-MSR (y) when CandidateServices (x) increase, is given by

$$y = 0.0012x^3 - 0.0611x^2 + 8.2678x - 16.6664 \quad (13)$$

From Equation 12 and Equation 13, we see that Clobmas is more sensitive to the number of CandidateServices than to the number of AbstractServices (see Figures

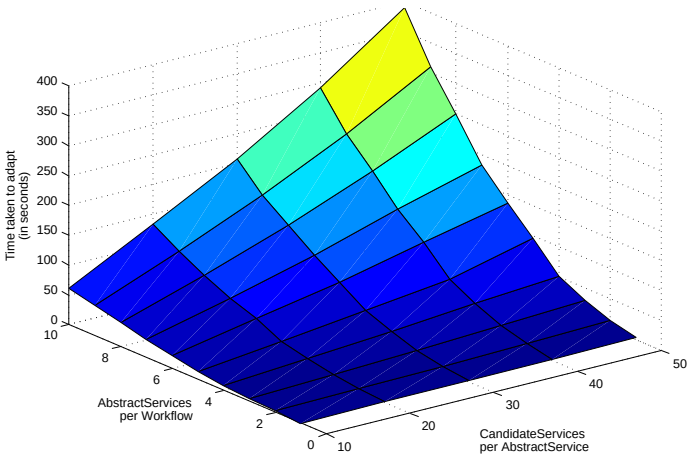

Fig. 14: Both AbstractServices and CandidateServices increase

12, 13 and 14). Although, both equations are cubic in nature, the cubic term in Equation 12 is negative. This indicates that it grows slower than Equation 13. This makes intuitive sense, since the increase in AbstractServices merely increases the number of BuyerAgents.

CandidateServices Vs. QoS attributes: In the following figures (Figure 15 and 16), we see that increasing CandidateServices and QoS attributes, affect the time-to-MSR in slightly different ways. There is a difference in the slope of the lines, when QoS attributes are increased (Figure 17).

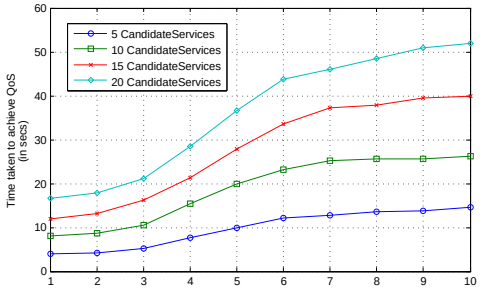

Fig. 15: QoS increase

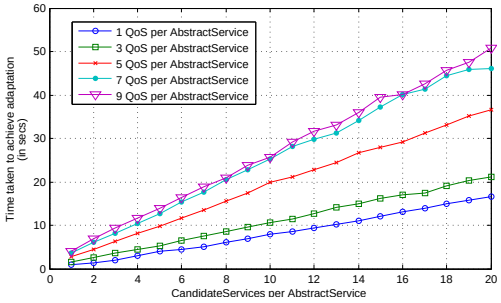

Fig. 16: CandidateServices increase

We see from Figure 17 that the slope of QoS axis is greater than that of CandidateServices. That is, time taken for adaptation increases faster when QoS attributes increase, as compared to the number of CandidateServices.

In the worst case, the polynomial describing the growth of Time-to-MSR (y) as QoS attributes (x) increase, is given by

$$y = 0.0303x^4 - 0.7769x^3 + 6.4554x^2 - 14.4945x + 25.7250 \quad (14)$$

Whereas, the polynomial describing the growth of Time-to-MSR (y) as CandidateServices (x) increase, is given by

$$y = 0.0030x^2 + 2.3721x + 2.4722 \quad (15)$$

Equations 14 and 15 indicate that it is preferable to increase CandidateServices, than QoS attributes.

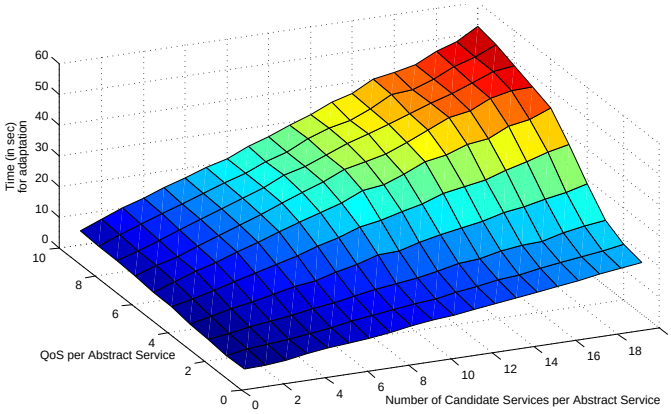

Fig. 17: Both CandidateServices and QoS increase

3.8.1 *The Cost of Decentralization*

As discussed in [21], an auction-based mechanism, by default, leads to a centralized mechanism. The auctioneer becomes a single point of failure, and thus leads to decreased robustness. This can be remedied by introducing multiple auctioneers. In clobmas, we use multiple markets for each AbstractService. This ensures that even if one market is non-functional, the others continue to function. Decentralization of markets comes at a cost. BuyerAgents have to choose which market to place their bids, to place them in multiple markets or a single market. Placing bids in multiple markets increases the chances of getting Candidate Services, but it also places a greater computational load on the BuyerAgent in terms of decision making. We currently choose markets randomly.

CandidateServices Vs. Number of Markets: Does decentralization affect time taken for adaptation or number of CandidateServices? In other words, should clobmas increase the number of CandidateServices available per market? Or would it be better to increase the number of markets?

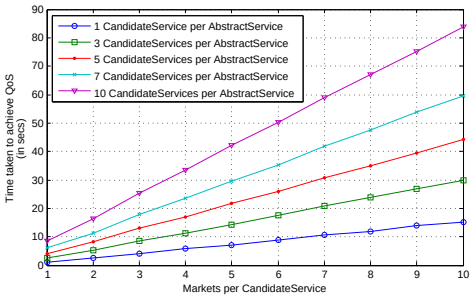

Fig. 18: Markets increase

The polynomial describing the growth of Time-to-MSR (y), as number of Markets increase (x), is given by

$$y = 8.3745x + 0.0800 \quad (16)$$

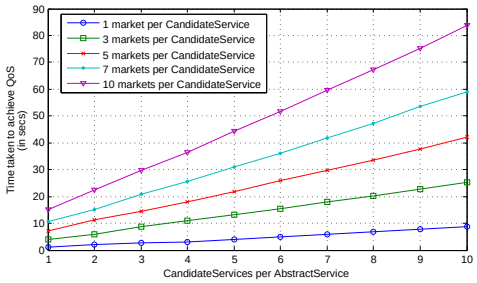

Fig. 19: CandidateServices increase

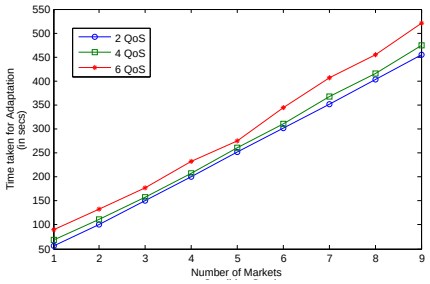

Fig. 22: Markets increase

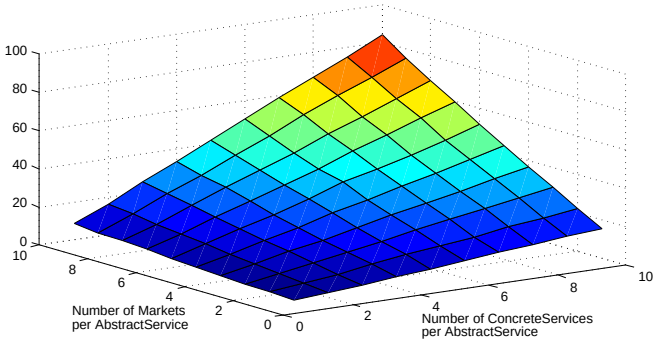

Fig. 20: Both Candidate Services and Markets increase

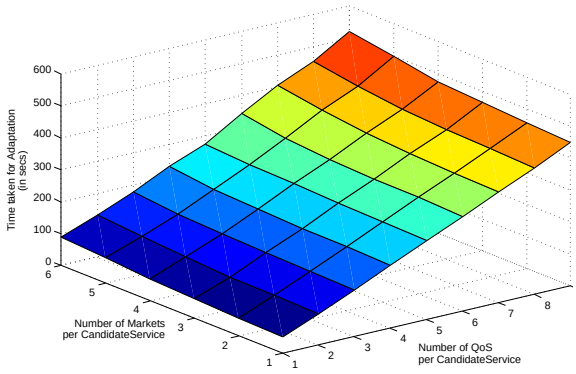

Fig. 23: Both QoS attributes and markets increase

The polynomial describing the growth of Time-to-MSR (y), as number of CandidateServices increase (x), is given by

$$y = 7.5972x + 6.8222 \tag{17}$$

We see from Equation 16 and Equation 17 that the slopes of both lines are linear. That is, increasing the number of CandidateServices *vis-a-vis* increasing the number of markets does not make a significant difference to the time-to-adapt.

QoS attributes Vs. Number of Markets: Next we look at how QoS attributes affect the time-to-adapt *vis-a-vis* the number of markets.

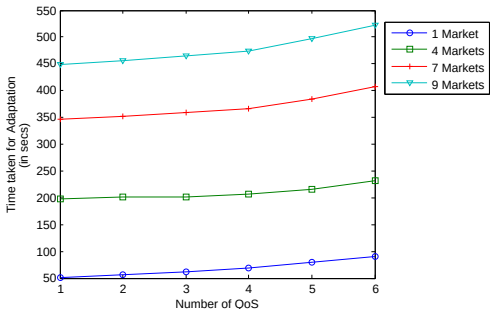

Fig. 21: QoS increase

In the worst case, the polynomial describing the growth of Time-to-MSR (y) as number of Markets (x) increase, is given by

$$y = -0.4595x^2 + 65.0048x + 26.6683 \tag{18}$$

In the worst case, the polynomial describing the growth of Time-to-MSR (y) as QoS attributes (x) increase, is given by

$$y = -0.1103x^4 + 2.3380x^3 - 14.9209x^2 + 42.0606x + 465.6389 \tag{19}$$

The increase in QoS attributes causes Time-to-MSR increase in a biquadratic way, as opposed to increase in number of Markets (Equation 18 and Equation 19).

3.9 Discussion

3.9.1 Strengths

A big strength of our mechanism is the high scalability that it offers. As service-oriented applications mature and cloud offerings become more standardized, it is easy to envision applications being composed out of several third-party services. In such a scenario, a mechanism that scales up to accommodate many concrete services is essential. None of the variables increased the time-to-MSR in an exponential manner. Rather, all of them are low-order polynomials.

Since the decision-making over which concrete service to instantiate is done in a de-centralized manner, the individual agents are simple and easy to implement.

Our mechanism implements a continuous adaptation scheme, thus leaving the system administrator free to attend to more critical tasks.

3.9.2 Weaknesses

The introduction of decentralization in the adaptivity process, while increasing robustness, also brings an increased cost in terms of time, and communication. There are two steps which are particularly affected. These are:

1) **Decomposition of Constraints**: Depending on the type of QoS constraints, the time taken to decompose them, changes. If the constraints are boolean or categoric, then the time complexity for k constraints in an application with n AbstractServices, is $O(n \cdot k)$. However, if the constraints are numeric, then the time increases substantially. Since each market is queried for its historical data of transactions. The complexity of decomposition now becomes $O((n \cdot k \cdot m) + (k \cdot SWR))$, where m is the number of markets and SWR is the complexity of applying *Stochastic Workflow Reduction* on the application structure [12].
2) **Calculation of achieved utility**: To calculate the total achieved utility, all the BuyerAgents communicate the QoS of the CandidateService that they selected, to the ApplicationAgent. The ApplicationAgent applies SWR to the numeric QoS attributes, to check if all constraints are met. This costs $O(n)$ in time, whereas the application of SWR depends on the structure of the application itself [12].

As the number of BuyerAgents for an application increase, or the number of markets increase (more likely), the time complexity of decomposition will increase.

3.9.3 Threats to Validity

Lack of Optimality: The market mechanism does not achieve optimum, either from the individual perspective or from the aggregate perspective. Since the self-adaptive mechanism merely attempts to satisfy the application's QoS targets, it does not try to achieve optimal set of concrete services, even if available. Rather, it stops adapting as soon as all QoS constraints are met.

Lack of Trust: In any market-oriented mechanism, there is the issue of trust between the buyer of a service and the seller of a service. How does the seller reliably ensure that the buyer does not make more calls to the web-service, than the number agreed upon? How is the provenance of calls established? How does the buyer ensure that the seller provides the QoS, that it has promised in the SLA? What legal/technical recourse does it have, in case of violation of contract? These are all issues that are still open research problems. Ramchurn et al [42] provide a good overview of these problems in the specific case of multi-agent systems. However, a good amount of research needs to be done, before all issues are acceptably resolved to the satisfaction of both, buyer and seller.

Simulation Effects: Any simulation model is a constrained version of reality, and as such, results from simulations should always be taken with a pinch of salt. Given this caveat, simulations help us carry out controlled experiments that would be too cumbersome or expensive to carry out in reality. Simulation toolkits are a valuable mechanism for testing out new, and sufficiently different ideas. CloudSim [8] is a toolkit that aims to make simulations of clouds easier to perform, and in the ideal case, we would have liked to implement our ideas on it. However, at its current level, it is insufficient to capture market mechanisms and multiple types of QoS attributes. In future, we aim to port our market-model to CloudSim, so as to enable richer modelling.

4 Related Work

4.1 Dynamic Composition of Web-services

There has been a plethora of work on dynamic composition of web-services. Much early work has been done in AgFlow [54] on Quality-Aware composition of web-services [5] and [53]. The authors propose a per-service-class optimisation as well as a global optimisation using integer programming.

[10] proposed a genetic algorithm based approach where the genome length is determined by the number of abstract services that require a choice to be made. Constraints on QoS form a part of the fitness function, as do cost and other QoS attributes. A big advantage of GA-based approach is that it is able to handle non-linear constraints, as opposed to integer programming. Also, it is scalable when the number of concrete services per abstract service increase.

[2] propose an interesting mechanism for cutting through the search space of candidate web-services, by using skyline queries. Skyline queries identify *non-dominated* web-services on at least one QoS criteria. A *non-dominated* web-service means, a web-service that has at least one QoS dimension in which it is strictly better than any other web-service and at least equal on all other QoS dimensions. Determining skyline services for a particular abstract service, requires pairwise comparisons amongst the QoS vectors of all the concrete services. This process can be expensive if the number of candidate concrete services is large. Alrifai et al. consider the case where the process of selecting skyline services is done offline. This would lead to an inability to adjust to changing conditions of available services and their associated QoS values. [56] propose an interesting method to achieve a good set of concrete services, using Ant Colony Optimization (ACO). ACO involves creating virtual ants that mimic the foraging behaviour of real ants. The search space of optimal concrete services is modelled as a graph, with sets of concrete services as vertices and edges being all the possible connections between different concrete service sets. The ants attempt to complete a traversal of the graph, dropping pheromones on the edge of each concrete service visited. The path through the graph that accumulates the most pheromones represents the near-optimal path of services to use. Our approach differs from the above approaches in two respects:

1) Consideration of time as a factor: In practice, the optimal set of concrete services may not be available at the time instant that an application is searching. The set of service providers changes with time, as does the set of service consumers. This means that the optimal matching of service providers to consumers changes with time. The approaches above do not take this into account.
2) Optimality not considered: Due to the infeasibility of computing the optimal set (being NP-hard), we concentrate on finding a good solution, rather than an optimal one. A good solution is one that does not violate any QoS constraints and meets the cost constraint within a certain margin.

4.2 Self-Adaptation

Applications that use dynamic service composition should be able to continuously monitor their current QoS levels and make adjustments when either the demand for QoS changes or the cost constraint changes. The application should thus be able to respond to both internal as well as external stimuli, to trigger a change in its constituent web-services. This change needs to be both timely, as well as correct, *i.e.*, the new set of services should not violate any of the application's QoS constraints, and the change should happen as fast as possible.

Self-Adaptation in software systems is the achievement of a stable, desirable configuration, in the presence of varying stimuli. These stimuli may be environmental (in the form of workload, failure of external components, etc.) or internal (failure of internal components, changed target states, etc.). Given that the range of stimuli that affect a software system is wide, Self-Adaptation has come to mean an umbrella term that covers multiple aspects of how a system reacts [44]:

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

However, most approaches to self-adaptation follow a common pattern: Monitor – Analyze – Plan – Execute, connected by a feedback loop. There are two approaches to self-adaptation: centralized and de-centralized. In a centralized self-adaptive system, the analysis and planning part are concentrated in one entity. This form of self-adaptation has the advantage of cohesiveness and low communication overhead as compared to a decentralized mechanism. The analysis and the plan can be communicated to the effectors, and feedback from obeying the plan is communicated back through the monitors (or sensors). Rainbow [14] and *The Autonomic Manager* [28] are classic examples of centralized self-adaptation.

Decentralized self-adaptation, on the other hand, distributes the analysis, planning or the feedback mechanism amongst different parts of the adapting system. This automatically implies a communication overhead, since all constituent parts must coordinate their actions. However, it also provides for robustness in the presence of node failure and scalability of application size. Cheng et al [13] have advocated that the feedback loop, which is a critical part of the adaptation, be elevated to a first-class entity in terms of modelling, design and implementation. Although, this would allow for reasoning about properties of the adaptation, there are no systems that we currently know of, that provide an explicit focus on the feedback loop. Most decentralized self-adaptation systems are typically realised as a multi-agent systems wherein the agents are autonomous in their environments and implement strategies that collectively move the entire system into a desirable state. [15] have advocated separating the functional part of the system from the adaptive part, thus allowing for independent evolution of both. Baresi et al [4] describe such a system, where adaptation is considered as a cross-cutting concern, and not a fundamental part of system computation. Baresi et al. use aspect-oriented programming to implement the Monitor and Execute part of the MAPE loop. They implement distributed analysis and planning by dividing the self-adaptive system into *supervised elements*, that perform the business logic of the application and *supervisors* that oversee how the supervised components behave and plan for adaptation. Aspect-probes form the sensors and actuators that link the supervised elements to the supervisors.

[18] describe another interesting approach to decentralized self-adaptation, through self-organization. DiMarzo et al. take a bio-inspired approach and use principles of holons (and holarchy) and stigmergy to get agents in a manufacturing department to perform coordination and control. A holon is defined by [33] to be both a part and a whole. Therefore, an agent is both autonomous as well as a part of a hierarchy, which influences it. The essential idea in their work is that with such structures, order emerges from disorder, as simple interactions build on each other, to produce progressively complex behaviour.

Weyns et al [50] study a decentralized self-healing system and a QoS-driven self-optimized deployment framework. Their approach is the nearest to ours. They suggest multiple decentralized models which feed into decentralized algorithms, which are in turn analyzed by decentralized analyzers. These analyzers then individually direct local effectors to make changes to the host system.

These approaches, while interesting, have not explicitly considered scale of adaptation. Any approach that attempts self-adaptation on the cloud, must concern itself with scaling up to hundreds and possibly even thousands of entities. Another issue that needs to be considered, is the effect of other self-adapting systems

operating in the same environment.

4.3 QoS Monitoring

Zeng [55], and Michlmayer [37] are good examples of online QoS monitoring. Zeng et al. classify QoS metrics into three categories: (a) Provider-advertised (b) Consumer-rated, and (c) Observable metrics. They provide an event-driven, rule-based model where designers can define QoS metrics and their computation logic (in terms of Event-Condition-Action rules), for observable metrics. These are then compiled into executable statecharts, which provide execution efficiency in computing QoS metrics based on service-events that are observed.
Michlmayer et al. provide their QoS monitoring as a *service runtime environment*. This service runtime environment addresses service metadata, QoS-aware service selection, mediation of services and complex event processing. The authors propose two mechanisms to monitor QoS: (a) a client-side approach using statistical sampling, and (b) a server-side approach using probes that are present on the same host as the service. The client-side approach is non-intrusive, in terms of not needing access to the service's host.
Both approaches, Zeng and Michlmayer, use an event-based mechanism to detect QoS values, and SLA violations, if any. This fits in neatly with our need for a non-intrusive, third-party based QoS Monitoring Engine. Our mechanism is agnostic to the actual QoS monitoring mechanism, that is used.

5 Conclusion and Future Work

Cloud-based service-oriented applications have the potential to self-adapt their QoS, depending on demand. Using a market-based mechanism maps nicely to the real-world situation of unpredictable change of QoS requirements, costs involved in adaptation and adaptation by competing applications. As the number of possible concrete services increase, the scalability of the self-adaptive mechanism becomes important. We see that the market-based mechanism consists of simple agents, is able to adapt well and yet scales linearly to the number of concrete services. We also see that it is robust in the presence of differences in demand and supply of QoS. Applications implemented as an ASN can thus scale and adapt to the changing business requirements of QoS.
We have not modelled complex seller-side behaviour. Specifically, actions like deliberate violation of QoS to free up resources for making Asks with higher prices or mis-reporting of QoS available. Mechanisms like penalties and reputation management can be used to prevent seller agents from behaving dishonestly. Also, we have not modelled adaptation on the part of the market. Sellers that lie about their QoS or, are generally unattractive for transactions may lower the reputation of the marketplace. Hence, the market could take steps to ensure that it is populated, only with sellers that are likely to be sold. In future work, we aim to systematically add these modifications to observe their effect on the collective adaptation.

References

[1] Do slas really matter? 1-year case study.

[2] Mohammad Alrifai, Dimitrios Skoutas, and Thomas Risse. Selecting skyline services for QoS-based web service composition. *Proceedings of the 19th international conference on World wide web - WWW '10*, page 11, 2010.

[3] Danilo Ardagna and Barbara Pernici. Global and local qos constraints guarantee in web service selection. pages 805–806, 2005.

[4] Luciano Baresi, Sam Guinea, and Giordano Tamburrelli. Towards decentralized self-adaptive component-based systems. *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems - SEAMS '08*, page 57, 2008.

[5] B. Benatallah, M. Dumas, Q.Z. Sheng, and a.H.H. Ngu. Declarative composition and peer-to-peer provisioning of dynamic Web services. *Proceedings 18th International Conference on Data Engineering*, pages 297–308, 2002.

[6] J.P. Brans and Ph. Vincke. A preference ranking organisation method: The promethee method for multiple criteria decision-making. *Management Science*, 31(6):647–656, June 1985.

[7] Ivan Breskovic, Christian Haas, Simon Caton, and Ivona Brandic. Towards self-awareness in cloud markets: A monitoring methodology. pages 81 –88, dec. 2011.

[8] R Buyya and S Pandey. Cloudbus toolkit for market-oriented cloud computing. In *Proceedings First International Conference, CloudCom 2009*, pages 22–44, 2009.

[9] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. *2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 5–13, September 2008.

[10] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for QoS-aware service composition based on genetic algorithms. *Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO '05*, page 1069, 2005.

[11] Jorge Cardoso, Amit Sheth, and John Miller. Workflow quality of service. Technical report, University of Georgia, Athens, Georgia, USA, March 2002.

[12] Jorge Cardoso, Amit Sheth, John Miller, Jonathan Arnold, and Krys Kochut. Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281–308, April 2004.

[13] B Cheng, R De Lemos, Holger Giese, and Paola Inverardi. Software engineering for self-adaptive systems: A research roadmap. *Software Engineering*, pages 1–26, 2009.

[14] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems - SEAMS '06*, page 2, 2006.

[15] SW Cheng and David Garlan. Making self-adaptation an engineering reality. *Self-star Properties in Complex Information Systems: Conceptual and Practical Foundations*, 3460:158–173, 2005.

[16] Scott H. Clearwater, Rick Costanza, Mike Dixon, and Brian Schroeder. Saving energy using market-based control. pages 253–273, 1996.

[17] D Cliff. Simple bargaining agents for decentralized market-based control. *HP Laboratories Technical Report*, 1998.

[18] Giovanna Di Marzo Serugendo, Marie-Pierre Gleizes, and Anthony Karageorgos. Self-organization in multi-agent systems. *The Knowledge Engineering Review*, 20(02):165–189, June 2005.

[19] Elisabetta DiNitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3-4):313–341, September 2008.

[20] Leticia Duboc, David Rosenblum, and Tony Wicks. A framework for characterization and analysis of software system scalability. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, page 375, 2007.

[21] Torsten Eymann, Michael Reinicke, Oscar Ardaiz, Pau Artigas, Luis Díaz de Cerio, Felix Freitag, Roc Messeguer, Leandro Navarro, Dolors Royo, and Kana Sanjeevan. Decentralized vs. centralized economic coordination of resource allocation in grids. In *European Across Grids Conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 9–16. Springer, 2003.

[22] Daniel Freidman. The double auction market institution: A survey. 1993.

[23] Dhananjay K. Gode and Shyam 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.

[24] Alok Gupta and DO Stahl. The economics of network management. *Communications of the ACM*, 42(9):57–63, 1999.

[25] Kieran Harty and David Cheriton. A market approach to operating system memory allocation. pages 126–155, 1996.

[26] Minghua He and Nicholas R. Jennings. Southamptontac: An adaptive autonomous trading agent. *ACM Trans. Internet Technol.*, 3:218–235, August 2003.

[27] Minghua He, N.R. Jennings, and Ho-Fung Leung. On agent-mediated electronic commerce. *Knowledge and Data Engineering, IEEE Transactions on*, 15(4):985 – 1003, july-aug. 2003.

[28] IBM. An architectural blueprint for autonomic computing. June 2006.

[29] Amazon Inc. Amazon spot-instances. December 2009. http://aws.amazon.com/ec2/spot-instances/.

[30] Nick Jennings. Automated haggling: building artificial negotiators. pages 1–1, 2000.

[31] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[32] Paul Klemperer. Auction theory: A guide to the literature. *JOURNAL OF ECONOMIC SURVEYS*, 13(3), 1999.

[33] Arthur Koestler. The ghost in the machine. 1989. ISBN 0-14-019192-5.

[34] MM Kokar and K Baclawski. Control theory-based foundations of self-controlling software. *Self-Adaptive Software and their Applications, IEEE Intelligent Systems*, 1999.

[35] Robert Laddaga. Creating robust software through self-adaptation. *IEEE Intelligent Systems*, 14:26–29, May 1999.

[36] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998.

[37] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Comprehensive qos monitoring of web services and event-based sla violation detection. pages 1–6, 2009.

[38] Vivek Nallur and Rami Bahsoon. Design of a Market-Based Mechanism for Quality Attribute Tradeoff of Services in the Cloud . In *Proceedings of the 25th Symposium of Applied Computing(ACM SAC)*. ACM, 2010.

[39] Jinzhong Niu, Kai Cai, Simon Parsons, Enrico Gerding, and Peter McBurney. Characterizing effective auction mechanisms: insights from the 2007 tac market design competition. pages 1079–1086, 2008.

[40] Jinzhong Niu, Kai Cai, Simon Parsons, Peter McBurney, and Enrico Gerding. What the 2007 tac market design game tells us about effective auction mechanisms. *Autonomous Agents and Multi-Agent Systems*, 21:172–203, 2010. 10.1007/s10458-009-9110-0.

[41] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. *Proceedings of the 20th International Conference on Software Engineering*, pages 177–186, 1998.

[42] Sarvapali D Ramchurn, Dong Huynh, and Nicholas R Jennings. Trust in multi-agent systems. *The Knowledge Engineering Review*, 19(01):1–25, 2005.

[43] a 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.

[44] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, May 2009.

[45] Michael Stonebraker, Robert Devine, Marcel Kornacker, Witold Litwin, Avi Pfeffer, Adam Sah, and Carl Staelin. An economic paradigm for query processing and data migration in mariposa. pages 58–67, 1994.

[46] Perukrishnen Vytelingum. *The Structure and Behaviour of the Continuous Double Auction*. PhD thesis, 2006.

[47] C.A. Waldspurger, T. Hogg, B.A. Huberman, J.O. Kephart, and W.S. Stornetta. Spawn: a distributed computational economy. *Software Engineering, IEEE Transactions on*, 18(2):103–117, feb. 1992.

[48] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: flexible proportional-share resource management. page 1, 1994.

[49] Michael P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *J. Artif. Int. Res.*, 1(1):1–23, 1993.

[50] Danny Weyns, Sam Malek, and J. Andersson. On decentralized self-adaptation: lessons from the trenches and challenges for the future. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 84–93. ACM, 2010.

[51] P Wurman. A Parametrization of the Auction Design Space. *Games and Economic Behavior*, 35(1-2):304–338, April 2001.

[52] L. Servi Y. C. Ho and R. Suri. A class of center-free resource allocation algorithms. *Large Scale Systems*, 1:51, 1980.

[53] Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, and Quan Z. Sheng. Quality driven web services composition. *Proceedings of the twelfth international conference on World Wide Web - WWW '03*, page 411, 2003.

[54] Liangzhao Zeng, Boualem Benatallah, Phuong Nguyen, and Anne H. H. Ngu. Agflow: Agent-based cross-enterprise workflow management system. pages 697–698, 2001.

[55] Liangzhao Zeng, Hui Lei, and Henry Chang. Monitoring the qos for web services. pages 132–144, 2007.

[56] Wei Zhang, Carl K. Chang, Taiming Feng, and Hsin-yi Jiang. QoS-Based Dynamic Web Service Composition with Ant Colony Optimization. *2010 IEEE 34th Annual Computer Software and Applications Conference*, pages 493–502, July 2010.

Vivek Nallur Dr. Vivek Nallur is a postdoc at the University of Birmingham. Prior to doing research, he used to work with VMware Inc. His research interests are complex systems, decentralized self-adaptation, ultra-large scale systems, and communication and decision-making in multi-agent systems.

Rami Bahsoon Dr. Rami Bahsoon is a Lecturer in Software Engineering at the School of Computer Science, the University of Birmingham, United Kingdom. He did his PhD from University College London (UCL), on evaluating software architectures for stability using real options theory. His research interests include Cloud Architectures, Security Software Engineering, Relating software requirements (non-functional requirements) to software architectures, testing and regression testing, software maintenance and evolution, software metrics, empirical evaluation, and economics-driven software engineering research.

APPENDIX

Decomposing Constraints: Decomposing constraints (Figure 24) involves communication between the BuyerAgents, their respective markets and the ApplicationAgent. The ApplicationAgent waits for the BuyerAgents to get data about previous transactions in the market, applies SWR [11] and checks whether the combination of QoS available in the market meets its end-to-end constraints. Based on the combinations that meet the end-to-end constraints, the ApplicationAgent creates local constraints for the individual BuyerAgents. These are then propagated to the BuyerAgents.

Computing Endowments: The budget for the individual agents is split in the ratio of the transaction prices that are prevalent in the individual markets (see Figure 25). Given historical price information in a market, the prices in the next trading rounds are likely to be around the same figure. Splitting the application's budget evenly across the BuyerAgents could possibly result in some agents getting excess endowment, and some agents too less. The ratio of their transaction prices allows the agents with expensive services to get a naturally higher share of the budget.

Trading Phase: In Figure 26, we show the communication that takes place during the trading phase. This phase is essentially a loop. It periodically evaluates the *bids* and *asks* in the orderbook, and tries to match them. If it is successful, the transaction moves into the second stage (see Figure 27). Based on whether a trade occurs or not, the application evaluates whether it needs to redistribute the endowments of its agents. It is possible that an agent is unable to find any potential transactions, simply because it does not have the budget to bid high enough.

CDA Protocol: The trading proceeds in two stages. In the first stage, the MarketAgent matches the bids and asks based on their individual QoS values and shout prices. After matching, a provisional transaction is created. This provisional transaction enters the second stage. In the second stage, the BuyerAgent compares all the Asks returned as provisional transactions. The top-ranked Ask is selected and the other Asks are rejected. The BuyerAgent enters into a transaction with the SellerAgent of the selected Ask.

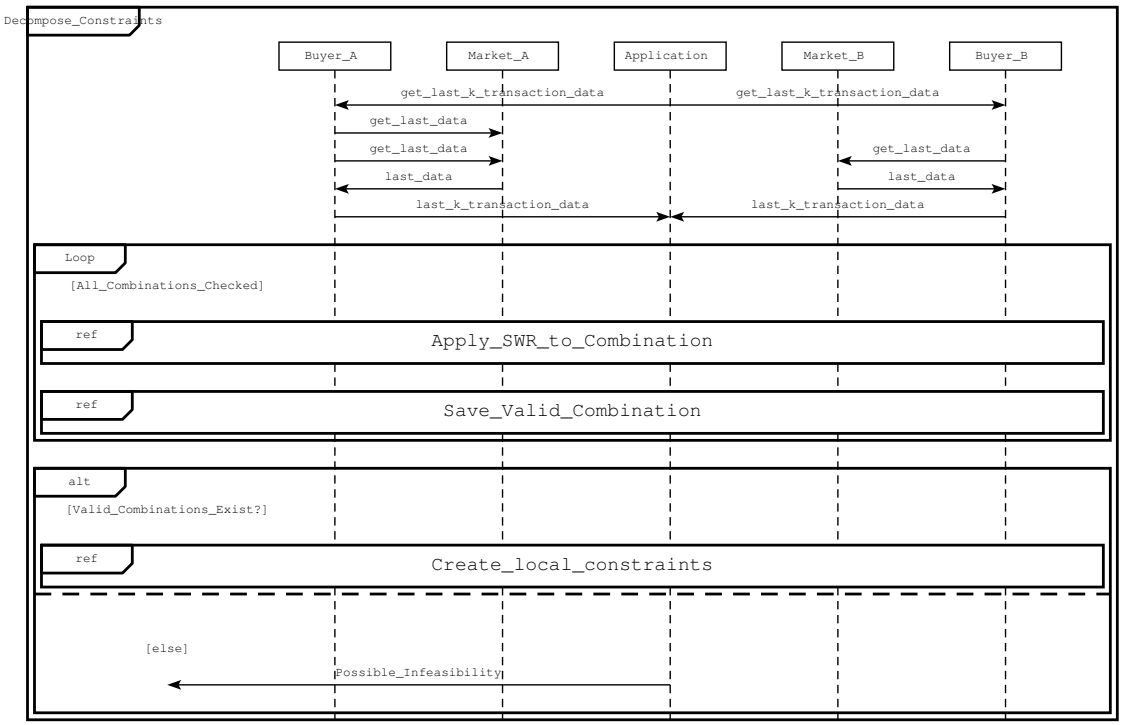

Fig. 24: Application decomposes its constraints into local constraints

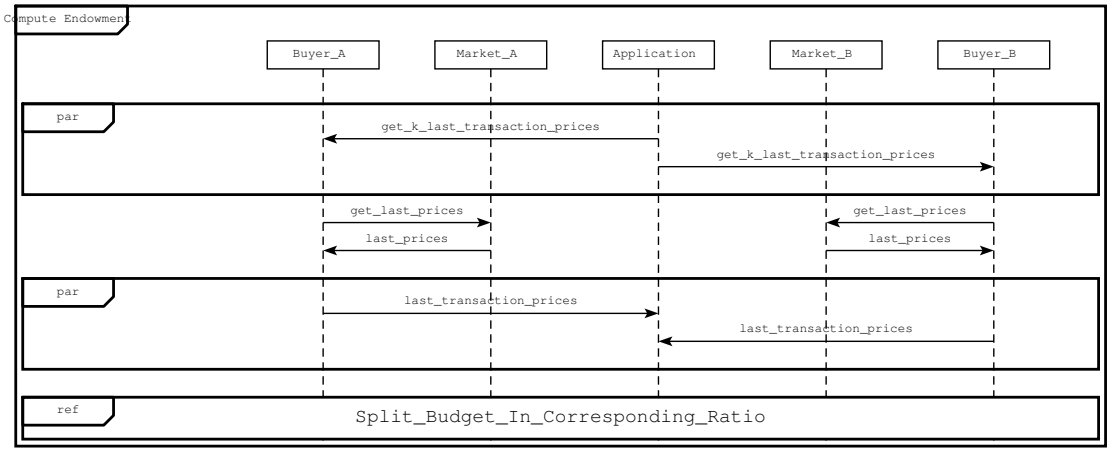

Fig. 25: Application computes endowment for its BuyerAgents

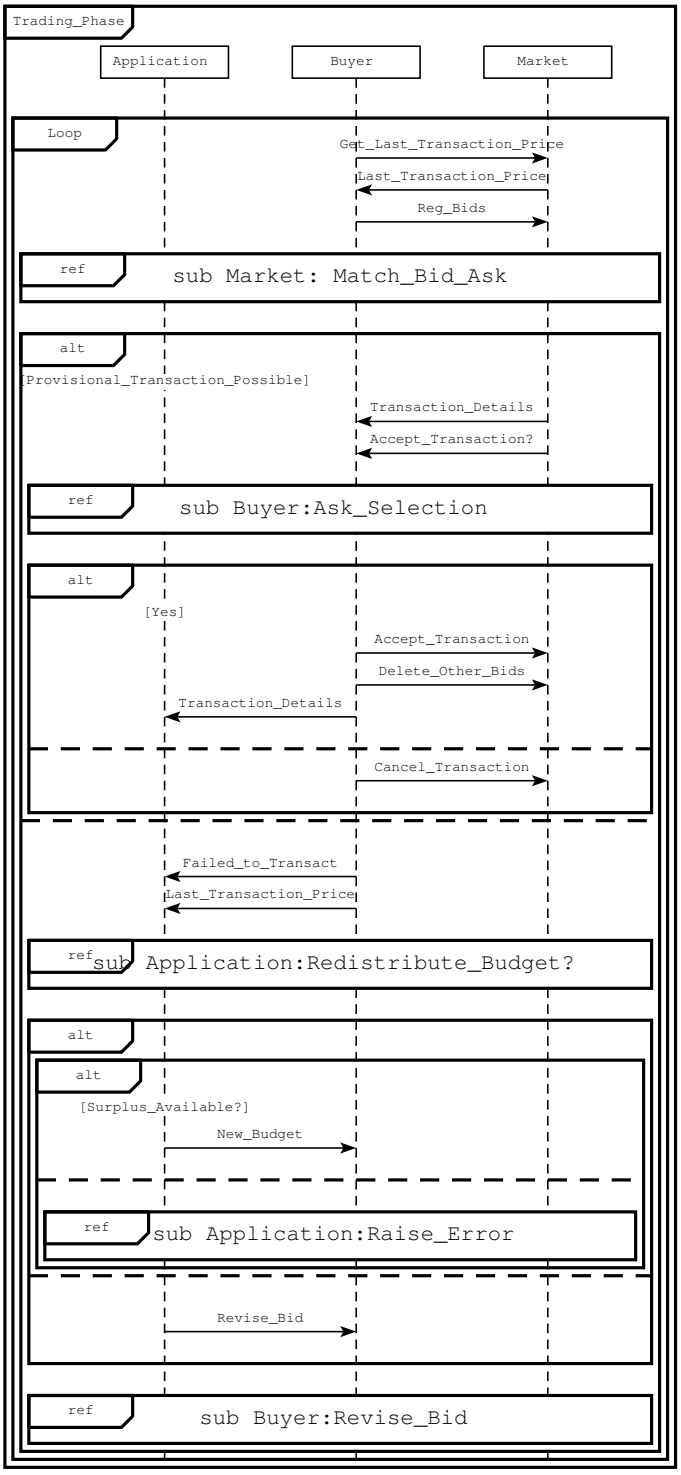

Fig. 26: The trading phase of buying a service

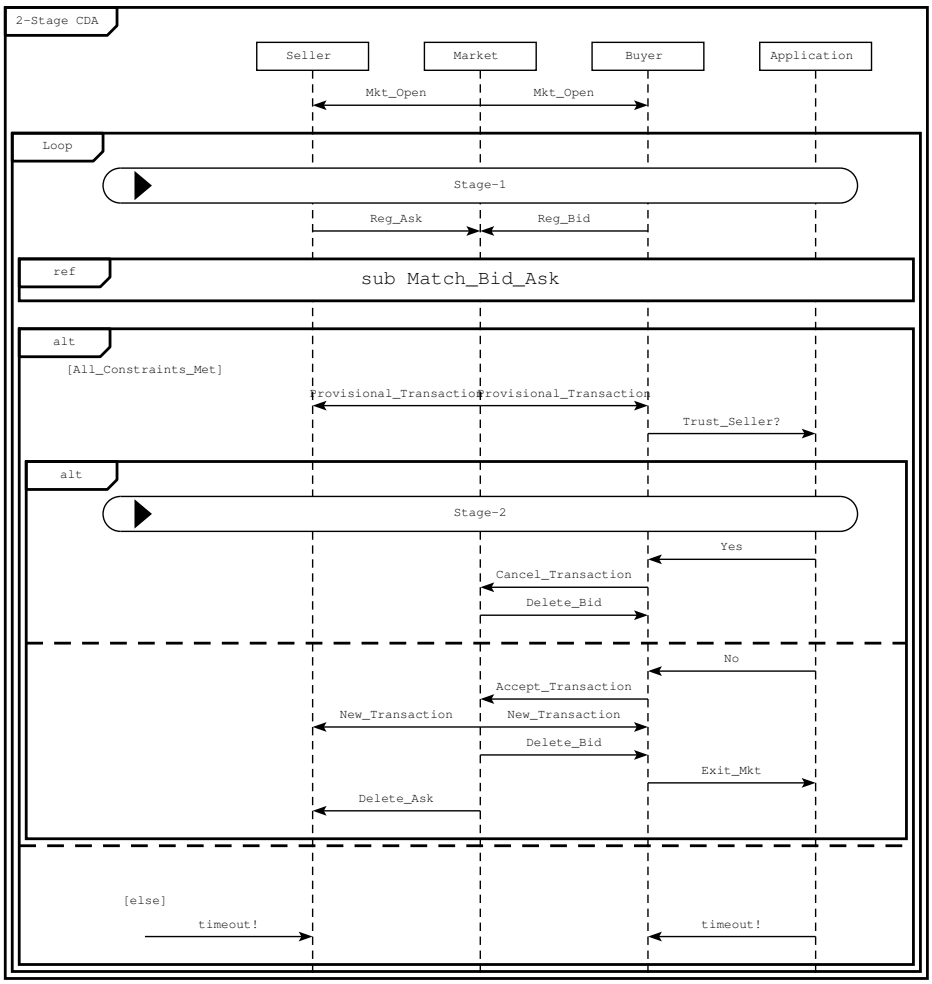

Fig. 27: Two-stage CDA protocol