A DECENTRALIZED SELF-ADAPTATION MECHANISM FOR SERVICE-BASED APPLICATIONS IN THE CLOUD

by

VIVEK NALLUR

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
College of Engineering and Physical Sciences
The University of Birmingham
31st May 2012

Abstract

This thesis presents a Cloud-based-Multi-Agent System (*Clobmas)*, that uses multiple double auctions, to enable applications to self-adapt, based on their QoS requirements and budgetary constraints. We design a marketplace that allows applications to select services, in a decentralized manner. We marry the marketplace with a decentralized service evaluation-and-selection mechanism, and a price adjustment technique to allow for QoS constraint satisfaction.

Applications in the cloud using the *Software-As-A-Service* paradigm will soon be commonplace. In this context, long-lived applications will need to adapt their QoS, based on various parameters. Current service-selection mechanisms fall short on the dimensions that service-based applications vary on. Clobmas is shown to be an effective mechanism, to allow both applications (service consumers) and clouds (service providers) to self-adapt to dynamically changing QoS requirements. Furthermore, we identify the various axes on which service-applications vary, and the median values on those axes. We measure Clobmas on all of these axes, and then stress-test it to show that it meets all of our goals for scalability.

Acknowledgements

At the very outset, I would like to acknowledge my gratitude to my supervisor, Dr. Rami Bahsoon. Without his patience, and guiding hand, this thesis would not be, what it is today. I would also like to thank my RSMG committee,Prof. Xin Yao and Dr. Behzad Bordbar. Your comments and suggestions have informed my research, in more ways than one.

One never forgets one's first boss, and I would like to record my appreciation and admiration for Dr. M. Sasikumar. Sasi, my introduction to research could not have come from a better person. I hope that I shall do good work, and not lose my integrity even in the face of great disappointment.

Daily life in Computer Science has been a pleasure, because of all the friends that I made during the course of the PhD. The heated discussions, passionate arguments, and random conversations that I've had with Peter Lewis, Seyyed Shah, Loretta Mancini, Benjamin Woolford-Lim, Edward Robinson, Catherine Harris, Gurchetan Grewal, Gurpreet Grewal-Kang, and many others have contributed to a rich, and fulfilling period of stay, in this institution.

For rich, and engrossing discussions about varied cultures, and words of encouragement when I was down, I would like to thank Amaria Zidouk and Shiwei Xu. You guys have been great office-mates, and it has been a pleasure to share an office with you.

If there has been one person who has been with me, quite literally, every step of the way, it has to be my 'twin', Christopher Staite. From procrastinating, to playing pranks, to cogitating about research, to relaxing in Coniston, we've done it all together. For being there, as a person to talk to, a support to lean on, a chum to share adventures with, and being an all-round good egg, thank you my friend. My journey through the PhD would have been lonelier, and tortuous, if it weren't for your cheery countenance (which I sometimes wanted to punch). It's been a good ride. I wish you a fulfilling career, and all happiness in the world, as you embark on a different journey with Becky.

Raji and Ramesh, it makes me sad that you are on the other side of the world, just when I finish this adventure. You participated in the beginnings of it, and it would have been nice to have you here, at this moment. Here's hoping that we land up in the same part of the world

again, sometime.

Elder brothers usually never know how much their sibling admires them. At least mine has never heard it from me. Deepu, you have been the epitome of cool, and someone to look up to, ever since I could remember. For knowing just when to give me space, when to beat things into me, and for always supporting me amidst thick and thin, thank you. You're ace.

Devikutty, even in my imagination, I could not have conceived of a better sister-in-law. I'm so glad that you are a part of our family.

A debt of gratitude, more than I can ever repay, is owed to Rajesh and Mala. From the very first day that I landed in England, you guys have been there for me. You have opened your hearts, your home, and made me feel welcome. You did not know it, but your love, affection and warmth, have pulled me through some really difficult times. So much so, that in the School of Computer Science, when I speak of 'going home', my friends know that I'm talking about Milton Keynes. Today, if I even consider making England home, it is because of you. Many, many, many thanks.

And finally, to the most important people in my life, Amma and Achan. Every year that I get older, I realize just how much I owe you, and how much of what I am, is due to you. You have been patient, indulgent and loving, to a fault. You've supported me through, what must've seemed like, crazy dreams and ambitions. And I'm afraid that it's probably going to continue to be that way. Whipper-snapper isn't ready to come home, yet. I shall not attempt to thank you, because words will not be enough. I love you, and to you both, I dedicate this thesis.

※ ※ ※ ※ ※

Publications arising from this Thesis

- **Conference**

1. V.Nallur and R.Bahsoon, *Self-Adapting Applications Based on QA Requirements in the Cloud Using Market-Based Heuristics* in Proceedings of ServiceWave 2010. Lecture Notes in Computer Science - 6481
2. V.Nallur and R.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 2010)
3. V.Nallur, R.Bahsoon and X.Yao, *Self-Optimizing Architecture for Ensuring Quality Attributes in the Cloud* in Proceedings of 2009 IEEE/IFIP WICSA/ECSA, 281-284

- **Journal**

1. V.Nallur and R.Bahsoon, *Market-Based Multi-Agent Systems as a Self-Adaptation Mechanism in the Cloud* in IEEE Transactions on Software Engineering. (In revision cycle. Submitted March 2012)
2. V.Nallur, *Self-Optimizing Quality Attributes in Cloud Architectures*: International Journal of Software Architecture, Vol.1, No. 1, Jul-Aug 2010

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

In my beginning is my end

East Coker, *T.S. Eliot*

In the short history of computing, there have been a few ideas that have changed the focus of software engineering. The shift from mainframe-based computing to desktop-based computing was one such idea. The computer began to be seen as an individual device, storing personal data and work, rather than an exclusively workplace-tool. Programming started to focus on user-interfaces, smaller disks, less cpu power and personalized tools for individuals, and families. The move towards mobile-based applications is, to some extent, an extension of the personalization idea. However, the idea of *cloud computing*, also known as utility-based computing, is prompting a move back towards enterprise-scale computing. Combining ideas of virtualization and data centers, cloud computing promises a near-infinite amount of computing power, storage and bandwidth. All of this at costs that are within the reach of small enterprises, and even individuals. One of the major selling-points of the cloud is the *pay-per-use* model of revenue generation. In this model, much like public utilities of electricity and water (and hence the name), customers pay for the amount of computing services they actually use. This allows customers, especially small and medium-scale enterprises, to cut back on *capital expenditure*, and focus only on *operational expenditure*. From the computational perspective, a big benefit of such an infrastructure is that an application can scale up, to serve millions of customers around the world, without significant effort on the application's part. On the other hand, when demand goes down, the application can release the computational power that it no longer needs. This flexibility in cost, however, has a downside. Although the cloud providers make available large amounts of computing power and storage, they make no guarantees about the *quality-of-service*(QoS) attributes of the services being provided by them. By QoS attributes, we refer to qualities like reliability, availability, performance, security and other non-functional requirements which need to be provided, maintained, evolved and monitored at runtime. These qualities are fundamental to the user's satisfaction with the application. In fact, even if an application performs its function correctly, the absence of the requisite QoS can lead to significant contractual losses. Consider a news-ticker service, that is used by banks and other financial institutions. If the news-ticker service is required by its contract to provide updated data every $10ms$, then regardless of the

correctness of the data, the service would be liable to pay a penalty to its customers, if it is unable to meet its performance target.

1.1 Problem

Service-based applications have the ability to change their constituent services at runtime. This implies that they have the ability to change both, their functionality and their QoS attributes dynamically. In this thesis, we are concerned with the problem of self-adaptation of cloud-based applications, with regard to changes in QoS requirements. A self-adaptive service-based application should, in theory, be able to change one of its services for another service, that is functionally the same, but delivers better QoS. This would lead to the application being able to exhibit higher performance, higher throughput or lower latency, as the situation warrants. Thus, the news-ticker application previously mentioned, could switch its current price-extraction service (in peak times, for instance), for another that has a higher throughput rating. It could switch back to a lower rated (and lower-priced) price-extraction service, in non-peak times. Obviously, this kind of adaptation would entail a cost. But not meeting customer expectation could entail a higher cost, or even loss of market-share. Therefore, an application that is able to adapt to changing QoS demand is better, than an application that cannot adapt. However, self-adaptation pre-supposes that the application is:

1. Able to evaluate multiple services for QoS, and

2. Able to select and use a suitable service within its budget

However, the current model of working on the cloud does not enable this. Services cannot be flexibly utilized and released. Also, selecting from a bundle of services that are functionally identical, but differ on QoS is a difficult problem. We now introduce a running example, that exemplifies the problem in more detail.

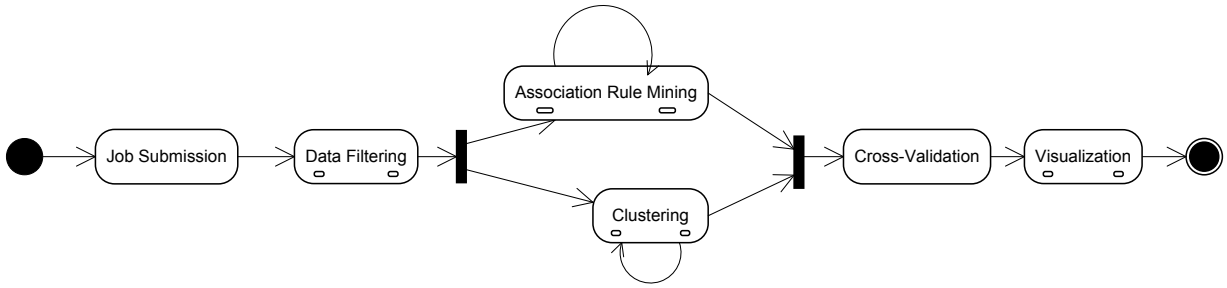


Figure 1.1: BizInt's Workflow constructed using composite services from the hosting cloud

1.1.1 The case of BizInt

BizInt, a small 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. It is confident that this application would be a huge success. However, BizInt also knows that, in order to attract and retain customers, it must offer a higher level of performance than its competitors. In order to do this, 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 1.1, 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. Although services with the required QoS are available on the cloud, selecting the right service is a problem. 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

	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 1.1: On-Demand Instance Pricing on Amazon EC2

QoS being exhibited by these services. In Table 1.1, 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 excessive overheads. In both cases, the customer faces a loss. To make matters worse, searching for new services, even ones that are functionally identical, but exhibit different QoS levels, is a difficult process. The number of available services to choose from is typically large, and the number of parameters on which to match, adds to the complexity of choosing.

In this situation, Amazon also, is unable to get a high utilization of its services. Due to a fear of over-provisioning or selecting the wrong service many companies, like BizInt, stay away from a cloud and choose to use a bespoke service-provider instead. If an adaptive infrastructure were available that allowed the application to flexibly choose different services based on its

requirements, more applications would be willing to join the cloud.

1.1.2 The Case of SkyCompute

SkyCompute is a new 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. Instead of providing specialist infrastructural services (like Amazon) or application framework services (like Google and Microsoft), it wants to provide a mixed-bag of generic services. It is planning to target the *Software-As-A-Service* (Saas) market, with 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 may 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. However, it wants to ensure that there is a high level of utilization of its services. So, it is looking for a mechanism that meets two criteria:

1. Allows customers like BizInt to create adaptive applications

2. Generates a higher utilization of services than the posted-offer model currently followed

1.2 Solution

We posit that a different type of market-based solution would allow both parties to be satisfied. We call our solution, Clobmas (Cloud-based Multi-Agent System). Also, Clobmas being principally designed for the cloud, should be able to scale to thousands of services. The over-arching research questions that this thesis would investigate, via Clobmas, are:

1. In a system of systems, each of which is a self-adapting application, what kind of a mechanism will allow satisfaction of QoS constraints?

2. How do we systematically measure the goodness of this mechanism? Is it good for BizInt? For SkyCompute?

3. How do we systematically measure the scalability of this mechanism?

We introduce a multi-agent based approach to tackle the problem of self-adaptation. Clobmas is a multi-agent based market, that performs service-matching based on QoS and budget constraints. In a nutshell, Clobmas:

1. Is targetted at service-based applications that use the cloud, from the SaaS perspective,

2. Allows for self-adaptation by such service-based applications,

3. Results in more services being used, than the current posted-offer mechanism,

4. Is scalable to thousands of services.

1.3 Contributions of This Thesis

This thesis makes the following contributions:

1. **Existing Cloud Resource Allocation**: We review existing work on cloud resource allocation. The objective of the review is to draw from existing approaches, and techniques, new insights that can assist the problem of QoS-aware dynamic selection for cloud-based applications. These insights inform the design of our novel self-adapting mechanism.

2. **Systematic Literature Review of Dynamic Service Selection**: The fundamental characteristics that we demand of our solution, is that it should be *scalable* and *decentralized*. That is, the self-managing mechanism should cater for the wide variation in QoS attributes, and associated constraints, for a large number of applications, that are composed of services offered by one or more clouds. A systematic literature review was conducted to establish the evidence that existing approaches, which look at the

problem of dynamic service selection, have limitations when they need to scale to the case of the cloud. Results of the literature review identified the axes, that self-adaptive cloud-based applications need to consider, to achieve scalability.

3. **Goal-Oriented Requirements and Design**: The axes identified in the previous step, also informed the quality goals of our proposed mechanism. More specifically, these axes have informed our reference model for systematically specifying, analyzing, and evaluating scalability requirements, which decentralized and self-adaptive architectures for service-based applications should consider.

4. **Novel Market-Based Mechanism**: This thesis views the cloud as a marketplace, and we propose a novel, market-based mechanism to enable decentralized self-adaptation, by service-based applications. The mechanism uses multiple double auctions, and multiple-criteria decision-making to allow applications to acheive their required QoS, while keeping within budgetary constraints. The use of multiple-double-auctions introduces robustness, which is an essential characteristic while dealing with large systems.

5. **Systematic scalability testing**: Using non-trivial simulations, we systematically test our mechanism for thousands of services, along all of the axes identified previously. The results show that our mechanism is scalable and robust. This form of testing can also be considered for other multi-agent systems, that need to verify their scalability properties.

1.4 Structure of the Thesis

This thesis is structured in the same order as the contributions listed previously.

1. In chapter 2, we start with looking at the various techniques used by cloud-providers to ensure that tasks are completed, while minimizing their QoS violations. These techniques are used mainly on IaaS clouds. To the best of our knowledge, there are no papers detailing techniques used by SaaS clouds.

2. In chapter 3, we perform a systematic literature review (SLR) of current dynamic service selection (DSS) techniques. An SLR is an important step in mapping out the research landscape, and identifying gaps in current research. Through the SLR, we discover that there are no generally agreed principles on which the scalability of DSS methods is currently evaluated. We propose four axes on which these techniques should be measured.

3. In chapter 4, we review the literature regarding self-adaptive architectures, specifically the use of Market-Based Control (MBC) techniques in decentralized self-adaptation.

4. In chapter 5, we explicate our mechanism (called Clobmas), which is a multi-agent system based on multiple double auctions. We combine ideas from micro-economic theory, management science and machine learning, to create a set of protocols for the multiple agents to interact, and achieve our goal.

5. In chapter 6, building on work in goal-oriented-requirements-engineering, we systematically arrive at the quality goals of our mechanism. We present the candidate architectural styles and the behavioural patterns amongst the agents, that allow us to meet these goals.

6. In chapter 7, we test Clobmas on all the axes that are identified through the requirements engineering process.

7. In chapter 8, we reflect on the journey of the thesis, and present concluding thoughts about directions that this research can take, in the future.

1.4.1 Publications

On page iv, we provided a list of publications that were generated, during work on this thesis. This thesis contains, and must be considered, the definitive reference of details and ideas, present in those publications.

Now, we begin the journey, with a foray into the current techniques used by IaaS cloud-providers, to fulfill the QoS levels promised in their *Service Level Agreements* (SLA).

CHAPTER 2

Resource Allocation in the Cloud

Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way. Science is supposed to be cumulative, not almost endless duplication of the same kind of things.

Richard Hamming

2.1 Cloud Computing

Advances in networking, storage and processing technologies have given us software that is mind-boggling in size and complexity of structure. Enterprises routinely deploy applications that span continents (e.g. through Grids, WANs, Internet), are composed of computing elements that are heterogeneous, and connected in complex topologies. In the wake of organizational change, economic downturn and a demand for tightening the belt on IT costs, there is a trend toward moving large applications to the cloud [10]. Organizations such as IBM [48] and Gartner [11] advocate cloud computing as a potential cost-saver as well as provider of higher service quality. We use the following definition of Cloud Computing, from [33]

> A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet

Clouds, as made available by the major players like Microsoft, Amazon, Google and 3Tera, use the following three different types of models:

1. **Infrastructure-As-A-Service (IaaS):** This is the most basic form of service that is provided. Infrastructure is provided in the form of virtual machines, virtual storage and bandwidth. Users of the cloud have to configure their virtual machines, install operating systems and create filesystems before they can start running their programs. Users also have to write meta-level programs to automatically increase the number of machines available to the running programs. Payment is made by the users on the basis of (virtual) CPU cycles used, (virtual) storage used and bandwidth taken or amount of data transferred. Amazon's *Elastic Compute* along with *Simple Storage* is the representative example of such a service.

2. **Software-As-A-Service (SaaS):** In this form of the cloud, the cloud provider provides software libraries and application frameworks into which a user plugs in her application. The number of machines, storage, bandwidth being used is transparent to the user at

the time of deployment. The application framework automatically detects when the load on a particular server reaches a threshold limit and provisions more machines, subject to the contractual limit. Payment is made on the basis of data transferred plus a certain amount of fixed rent.

3. **Platform-As-A-Service (PaaS):** Here the cloud provider makes available a particular platform on which the user develops the application. This has the advantage that the user is freed from the maintaining the platform and all upgrades etc. are taken care of, by the cloud provider. Again, computational resources keep increasing to keep up with the load being placed on the application. Microsoft's *Azure* and Google's *App Engine* are representative examples of such a service. Azure provides the users with the .NET platform, while Google provides its users with Python and Java-based platform.

. The differentiating factors between Software-As-A-Service and Platform-As-A-Service seem very small. This is attributable to the nascent stage that the cloud industry is in, with different companies coming up with different jargon for their offerings.
Usually, applications in the cloud are of the following types:

1. Web-applications that cater to diverse users across the Internet and demand fast response times.

2. Enterprise applications that cater to different business units across the world and require large amounts of secure, reliable data transfer and high availability (99.999%).

3. Scientific applications that need raw CPU or enterprises that perform batch processing.

The third type of application has the natural advantage of '*cost associativity*' offered by the cloud [4]. That is, on the cloud there is no difference between running a program on 10,000 machines for one hour and a program on one machine for 10,000 hours. This enables an organization to be extremely flexible in its IT expenditure. It is with the first two categories of applications that we are concerned with. From the perspective of these applications, cloud computing still has to prove itself on various fronts, before it can be considered a beneficial solution. These list of concerns include:

- Availability — When mission-critical applications are hosted in the cloud, the organisation becomes critically dependant on the cloud provider not having an outage. Any downtime will not only be embarrassing, but also possibly result in a high economic cost.

- Security — Cloud providers are able to sustain their economies of scale by statistically multiplexing virtual machines on physical machines. Any security hole in the virtualization technology that allows co-resident virtual machines to make unauthorised reads/writes could compromise information assets of a firm.

- Performance — Organisations that depend on a certain level of performance will need performance guarantees of their applications in the cloud. The typical mechanism for ensuring some kind of quality is a *Service Level Agreement* (SLA). An SLA is an agreement between the service provider and the consumer, that specifies the Quality of Service (QoS) that the service will provide.

2.2 Resource Allocation

The promise of nearly infinite computing power and concomitant bandwidth/storage, is implemented by cloud providers using technologies like *virtualization*, and *data centers*. A *data center* is a huge collection of commoditized computers, that are networked together in one physical location. This so-called 'warehouse' allows the owner to benefit from economies of scale. Thus, the owner is able to buy many thousands of computers and networking equipment at a cheap rate, and use common power and cooling equipment to run them all. *Virtualization* is a technology that allows a physical machine to run several virtual machines, each of which is akin to a real machine, from a program's perspective. But it can be switched on, booted up, and turned off without affecting the physical machine. This allows cloud providers to leverage their high-performance physical servers and lease out standardized machine images to consumers' applications. This form of lease-based computing enables it

to use (say) 100 physical machines and provide (say) 300 virtual machines. It also allows the provider to amortize the physical cost of computing, networking, power supply and cooling, amongst many customers. From a consumer's point of view, it is able to lease computers that exist only virtually and therefore pay a smaller cost on a per-machine basis. It can lease more computing power when its applications need it, and release them when they are no longer needed. Thus, a cloud provider combines the economies of scale created by the data center, and the flexibility created by virtualization, to provide a flexible infrastructure to its clients.
It is now fairly intuitive that a cloud provider would like to have as many of its resources leased out to consumers, as possible. The more resources that are leased out, at any given time, the more the cloud provider increases its revenue. On the other hand, cloud consumers would like as much flexibility as possible in leasing out resources, since they would not like to pay for more than what is absolutely essential. Also, if the cloud provider agrees to a certain SLA, which specifies the QoS, and does not deliver resources according to the SLA, it would lose customers. Thus, the cloud provider cannot overcommit its resources, since that would leave customers unsatisfied. It cannot undercommit its resources, since that would also result in a loss of revenue. Thus, the allocation of resources to customers is a critical part of cloud management. We now look at various resource allocation mechanisms that have been proposed in the context of the cloud. Although these mechanisms deal with IaaS clouds, we review them because they provide us with a perspective on how resources are demanded and allocated, in a cloud. A SaaS cloud will have to deal with more QoS constraints, than an IaaS cloud.

2.3 Current Approaches

A resource allocation policy is a mechanism by which a cloud provider allocates resources to a service request. By allocating resources, we mean that the cloud provider assures the service consumer that the required resources will be available at the required time, for the required duration. In addition to providing resources, the resource allocation policy must

ensure that the QoS being promised in the SLA will also be fulfilled. For example, if an SLA specifies that encryption will be done using 1024-bit encryption, it is not enough to provide a service that will perform 512-bit encryption. This would be a violation of the SLA and the service provider could possibly be penalized. There are various approaches proposed towards efficiently allocating resources in the cloud. First, we explicate the approach used by Amazon and other open-source cloud toolkits, and then we look at other approaches proposed by researchers.

2.3.1 Policy-Based Approach

Amazon's *Elastic Compute Cloud* (EC2) is a public cloud that exposes all the computing power, storage, bandwidth, etc. as services to its consumers. EC[49] is the canonical example of *Infrastructure-As-A-Service* cloud. Nimbus[76], Eucalyptus[74] and OpenNebula[87] are open-source cloud management toolkits. The resource allocation policies followed by these toolkits are summarized in Table 2.1:

Cloud Toolkit	**Allocation Policies**
Amazon (EC)	Best Effort
Nimbus	Immediate
Eucalyptus	Immediate
OpenNebula	Best Effort

Table 2.1: Resource allocation policies. Excerpted from [69]

An *immediate allocation policy* only accepts service requests, if it has the resources to immediately deal with the request. This has the obvious drawback that even if resources become available in the very near future, the service request will be denied, thus leading to a loss of revenue. A *best-effort allocation policy* is much like an immediate allocation policy, but if the request cannot be serviced immediately, it is put into a FIFO queue. This means that the service provider does not guarantee that a service request will be honoured at a particular time, or even in a particular duration. Thus, a service that is granted access to computing re-

sources can be pre-empted at any time, and resumed whenever there are sufficient resources available.

Haizea [75] is an open-source resource lease manager that integrates with OpenNebula, and implements more complex allocation policies. In addition to *immediate* (IM) and *best-effort* (BE), it supports two more allocation policies:

1. Deadline Sensitive (DS): A kind of a BE lease, but with a certain time limit. Thus, if accepted, the resource lease manager guarantees that the task will be completed before a deadline. The services requested are still preemptable (like BE), but it assures consumers that their request will be completed within a time-limit.

2. Advance Reservation (AR): This is like an IM lease (non-preemptable), but deals with resource allocation at some future time.

Swapping and Backfilling: Nathani et al. propose an improvement to Haizea[69]. The DS policy is used for deadline sensitive tasks. However, Haizea's current implementation of DS tries to find a single time slot to accommodate the service request. In doing so, it resorts to re-scheduling other leases, to create space for the new one. It displays this behaviour because it does not consider the amount of resources demanded by the request. Rather, it only sorts the DS leases based on the amount of slack between them and their preemptability. To improve upon this, Nathani et al. propose that the amount of resources also be taken into consideration. This allows Haizea to swap two leases, as long as their deadlines are met, thus potentially freeing up resources for a new, incoming lease. They also propose splitting up resource chunks, to enable swapping more easily. Thus, a lease that requires 10 units of resource X may not be satisfiable, but if split into two requests of 5 units each, that lease may be accommodated. This process of splitting up a resource-lease into smaller chunks and then allocating them, out of order (Haizea processes lease requests in order of submission time), is called *Backfilling*. Implementing swapping and backfilling improves Haizea's acceptance rate of DS leases and also improves the system's utilization rate. The

authors, however, do not measure how long it takes for the algorithm to run, and how this runtime grows as the number of leases increase. Run-time complexity is a key characteristic of cloud-based scheduling systems, and needs to be considered while looking at improved resource-allocation algorithms.

2.3.2 Deadline-Driven

Vecchiola et al. describe Aneka, a *cloud application platform* that is capable of provisioning resources from multiple sources, clusters, private and public clouds[96]. Aneka implements a *PaaS* model, with an API that allows developers to create their application in a variety of programming models (Map-Reduce, Bag-of-Tasks, Threaded, etc.). The resource provisioning algorithm used by Aneka is a deadline-driven, best-effort algorithm. For every task that the scheduler decides to run, the resource provisioner calculates the predicted number of resources required by the task. It compares the predicted resource usage with current availability to estimate whether the task's deadline can be met. If so, the task is run, or the provisioning service requests for more resources from clusters, private cloud and (finally) from public clouds. When the task is complete, these resources are released. This approach makes computing resources truly elastic, by providing middleware that abstracts away from the origin of the resources.

2.3.3 Machine-Learning Algorithms

The machine-learning approach views resource provisioning as a demand-prediction function, and applies various machine learning algorithms to estimate how the *Workflow* changes, over time. Workflow refers the set of services that an application is composed of. Sadeka et al. use multiple linear regression and feedforward neural networks to predict resource demand for a cloud[51]. In the context of PaaS, they enable a hosted application to make autonomic scaling decisions using intelligent resource prediction techniques. All of these techniques, however, require historical data to learn effectively. To simulate historical data, Sadeka et al.

run a standard client-server benchmark application, TPC-W [28], on Amazon's EC2 cloud. This data is then divided into training sets and validation sets, using a variety of statistical techniques.

Linear regression attempts to fit a curve to the given data points, while minimizing the error between the curve and the observed data. It effectively yields a function that, when successful, approximates the real process that gave rise to the observed data. *Neural Networks* are another machine-learning technique, that approximate a real-world process. A neural network consists of an input layer, an output layer and one or more hidden layers. Each layer consists of neurons that have a certain value, and are connected to the next layer's neurons by way of synapses. These synapses initially start with random weights, but get adjusted as training goes on. The network is trained by presenting a known input to the input layer, and observing the output at the output layer. The difference between the output produced by the network, and the actual output is the error. This error is fed backwards into the network, to allow the synapses to change their weights. This is called training the network. The *Sliding Window* technique [30] is a sampling technique to allow the learning algorithm to view the same dataset from different sample perspectives.

After training, both Linear Regression and Neural Networks were evaluated using unseen data and compared on several statistical measures. Neural Networks were found to be able to generalize well, with the prediction of resource usage closely matching the actual data. However, the only parameter used by the authors was the resource load placed on the cloud provider. In reality, each resource has several attributes that are each decision parameters in their own right. In such a scenario, it is difficult to train such machine learning methods. Increasing the number of hidden layers in a neural network does not necessarily increase its predictive power. Also, the need for historical data presents a problem in the case of dynamically changing importance of QoS attributes, for any particular service.

2.3.4 Cost-Optimization

Byun et al. propose a cost-optimization method for task scheduling[17]. They attempt to find the minimum number of resources, given a set of tasks with deadlines. Like swapping and backfilling proposed in [69], Byun et al. attempt to move tasks that are not in the critical path of the application Workflow, such that the total cost of the resources used for the Workflow is minimized. Their algorithm, called Partitioned Balanced Time Scheduling (PBTS), assumes that tasks are non-preemptable and executed on a single resource or set of resources. Based on the minimum time charge unit of the provisioning system (say 1 hour on Amazon's EC2), the algorithm divides the application's estimated work-time into time-partitions. It then iterates over all the tasks that are yet to be executed, and estimates the next set of tasks that can be fully scheduled in the next time-partition and their required resources. Having done this, it schedules these tasks for execution, and repeats the cycle for the remaining tasks, until all tasks are completed. While this results in minimum cost of resources, it does not take into account other qualities of resources that might be required by a certain task. For instance, a task might request 1 hour of processing on a node that has a certain kind of graphics chip or level-1 cache. Since PBTS assumes that all resource units are homogeneous, these type of requests cannot be accommodated.

2.3.5 Market-Based Methods

There is an increasing amount of work that champions the use of market-based methods for resource allocation in either a cloud or a grid setting. This has been due to the realization that all tasks are not of equal priority, and assigning economic value to tasks is a natural (and decentralized) way to factor their relative priorities into the resource allocation procedure. Fu et al.[35], Auyoung et al.[5], Lai et al.[61] and Irwin et al.[50] have all considered utility-based markets for sharing computing resources. Lai et al.[61] describes an implementation of an auction-based resource allocation, with compute resources distributed across a geographical area. However this approach ignores the notion of QoS per resource, as well. It assumes that

every different point on the QoS scale can be considered a different resource, and makes it the user's responsibility to find and bid for time, on these resources. Buyya et al. advocate a market-oriented mechanism to allocate resources. "For Cloud computing to mature, it is required that the services follow standard interfaces. This would enable services to be commoditized and thus, would pave the way for the creation of a market infrastructure for trading in services." [16].

There is little consensus however, amongst the market-based advocates, on how the market should be structured, and what trading and contracts should be based upon? Wu et al.[104] propose using *Service Level Agreements* (SLA) as the legal contract which specifies the qualities that the service provider is expected to provide. They provide heuristics to increase the amount of profit, made by the service provider by minimizing costs incurred in setting up resources. This is done via minimizing the penalty associated with the response time violation, in an SLA-based environment. Sun et al. propose using a *continuous double auction* (CDA) with an algorithm to determine what the Nash Equilibrium allocation would be[90]. Nash Equilibrium[9] is a solution concept in game theory, where given a state of the game, each player has a strategy that maximizes his utility for that state. The player cannot unilaterally change his strategy to get a better outcome. Sun et al. model the cloud as a game, with the resources and tasks inside it as players. They propose an algorithm called NECDA that continually runs a double auction, until the players' bids and asks are in a state of Nash Equilibrium. This however assumes that each task (while bidding for a resource) has full knowledge of the mean reserve prices of all the other resources, and numbers of remaining resources. Also, each resource while creating its ask, knows the maximum price it can achieve for that resource by asking the auctioneer. These are clearly very strong assumptions and, will not scale when the number of resources and tasks rise, in the cloud.

2.3.6 Ranking-Based Method

Machine learning methods, utility functions, etc. try to search the computational space of resource to consumer allocation, so as to optimize some parameter or another. However, in

many cases, there are multiple criteria that a cloud provider would like to optimize. In such a situation, searching for an optimal solution might not be computationally feasible. In such cases, fast, but sub-optimal, solutions are better than optimal, but slow solutions. In this scenario, Yazir et al.[108] use Distributed Multi Criteria Decision Analysis in a distributed fashion to achieve resource allocation. The essential idea is to create an outranking method, so that the decision-maker can choose amongst alternatives. Yazir et al. use an agent-based method to assign each physical machine in the cloud with a Node Agent, which makes the decision to select, retain or migrate virtual machines, based on some ranking criteria. This ranking mechanism is based on PROMETHEE[12], which is a multi-criteria decision analysis method. PROMETHEE aims to model each criteria in a decision problem according to a function, that the decision-maker feels best models that criteria. It then systematically allocates a score to each option, according to the value of each of the criterion presented by the option. Based on these scores, the decision-maker is able to rank multiple options. Yazir et al. use this mechanism to decide which physical machine, a particular virtual machine should migrate to. This mechanism of transferring virtual machines amongst the physical machines in the datacenter, allows the decision-maker to continuously ensure that the SLAs it has committed to, are not being broken.

2.4 Conclusion

In this section, we looked at various approaches being taken towards resource allocation in the cloud. Almost all of the approaches have looked at the cloud in terms of *Infrastructure-As-A-Service*. By far, the most popular idea has been to view resources in the cloud, in terms of simple descriptors like cost, CPU cycles, memory availability, etc. Even when viewed through such a lens, the problem of dynamically allocating resources to applications, is a difficult one. Additional attributes like security, reliability, throughput, etc. have not been a part of SLAs. There have been no systematic studies on the cloud in its other avatars, viz., *Software-As-A-Service* and *Platform-As-A-Service*. Some approaches like [16, 104] attempt

to deal with higher-level attributes of resources in the cloud, but have not considered the mechanism for systematically evaluating the QoS attributes, while allocating resources.

Since our research question involves service-based applications residing in the cloud, we need to look at mechanisms used for enabling *Software-As-A-Service*. In the absence of literature dealing with service-selection in the cloud, we look at the related domain of dynamic service composition, instead. Specifically, we look at techniques in dynamic service composition, from the point of scalability. The cloud is composed of thousands of services, and so any technique for service selection must evaluate whether it is able to scale to the level of the cloud. To this end, in the next chapter, we conduct a systematic literature review of scalability in dynamic service composition literature.

CHAPTER 3

Scalability in Dynamic Service Selection

3.1 Introduction

There are a plethora of techniques to perform dynamic web-service selection, each with its own advantages and disadvantages. Some are centralized techniques, others are decentralized, some vary the size of the Workflow, some vary the number of QoS attributes, etc. In such a scenario, it is difficult to analyse whether a particular technique is applicable to a particular application or not? Is there a yardstick against which all techniques of dynamic web-service composition can be measured? How do we know whether a particular approach will scale to the levels required by a software architect? In this chapter, we compare the various techniques proposed in literature, derive the four axes that affect these techniques' scalability and attempt to compare each of the salient techniques.

3.2 Dynamic Web-Service Selection

Service-Oriented Architecture has brought about a paradigm shift in the way we think about creating an application. Instead of linking programs and libraries at compile-time, we are now able to merely specify the functionality that the component parts should have, and the application can be dynamically composed using web-services. A web-service[7] is a self-describing computational entity that can be used to perform various kinds of functions. These can be composed together, in a specific order, to deliver some functionality. A web-service is so-called because it uses web-based standards like XML, SOAP, etc to achieve its communication and data exchange, thus allowing the application location and platform independence. This allows an application to search a service repository for service that it wants, and then bind to it. This dynamic binding allows for the notion of an application changing the QoS properties that it exhibits, at runtime. Depending on the task at hand, or the budgetary resources or any other QoS constraints that the architect imposes, the application can potentially pick an appropriate service and achieve its functional and non-functional targets. Fundamental to the notion of a service-oriented architecture is the concept of a *service consumer*, a *service provider* and a *service registry*.

Service Consumer: A service consumer is the entity requesting the service,

Service Provider: A service provider is the entity responsible for describing and providing an implementation of the service.

Service Registry: The service registry provides the mechanism for publishing, finding, selecting and binding to a service.

These roles are independent of each other, and thus an entity can take on multiple roles simultaneously. Working in concert, they allow an application to be composed of services that have no prior knowledge of each other.

The service registry publishes information about the service's interface, its conceptual description and the QoS attributes, like reliability, availability, security attributes, performance

metering, etc. A service consumer uses this information to select a particular service from the many that are available in the registry. In this quest, it faces two problems: matching the interface and conceptual description to its need and ensuring that the QoS advertised, are acceptable to its own application's needs. Dynamic service selection while optimizing the QoS is known to be an NP-hard problem[110]. This immediately implies that any solution that tries to find the optimal solution, would find itself confronted with an exponentially increasing search space. Hence, most solutions focus on obtaining good solutions, that meet the QoS constraints that an application specifies. Given this scenario, it would be reasonable to expect that solutions proposed in literature identify how scalable they are, at least with regard to each other. Different variables, Workflow size, number of QoS attributes, etc., affect the scalability of a technique in different ways. However, we find that there is no such systematic comparison amongst QoS-based approaches to dynamic service selection.

In order for our survey of literature to be useful, we perform a systematic literature review, as outlined by Kitchenham[55]. Kitchenham's guidelines lay out the process of performing a literature review, with an end to ensuring that, it can be analyzed by a third party for completeness and fitness for purpose. To our mind, the question of how an application living on the cloud, should adapt itself to varying QoS demands breaks down into two questions: how does an application change its exhibited QoS levels dynamically, and how does a cloud efficiently allocate services to applications? To that end, in this chapter we perform a systematic literature review of dynamic web-service composition, and resource allocation in the cloud. Based on Kitchenham's guidelines, we chose the following process for doing the literature review and the following sections explicate each part of the process.

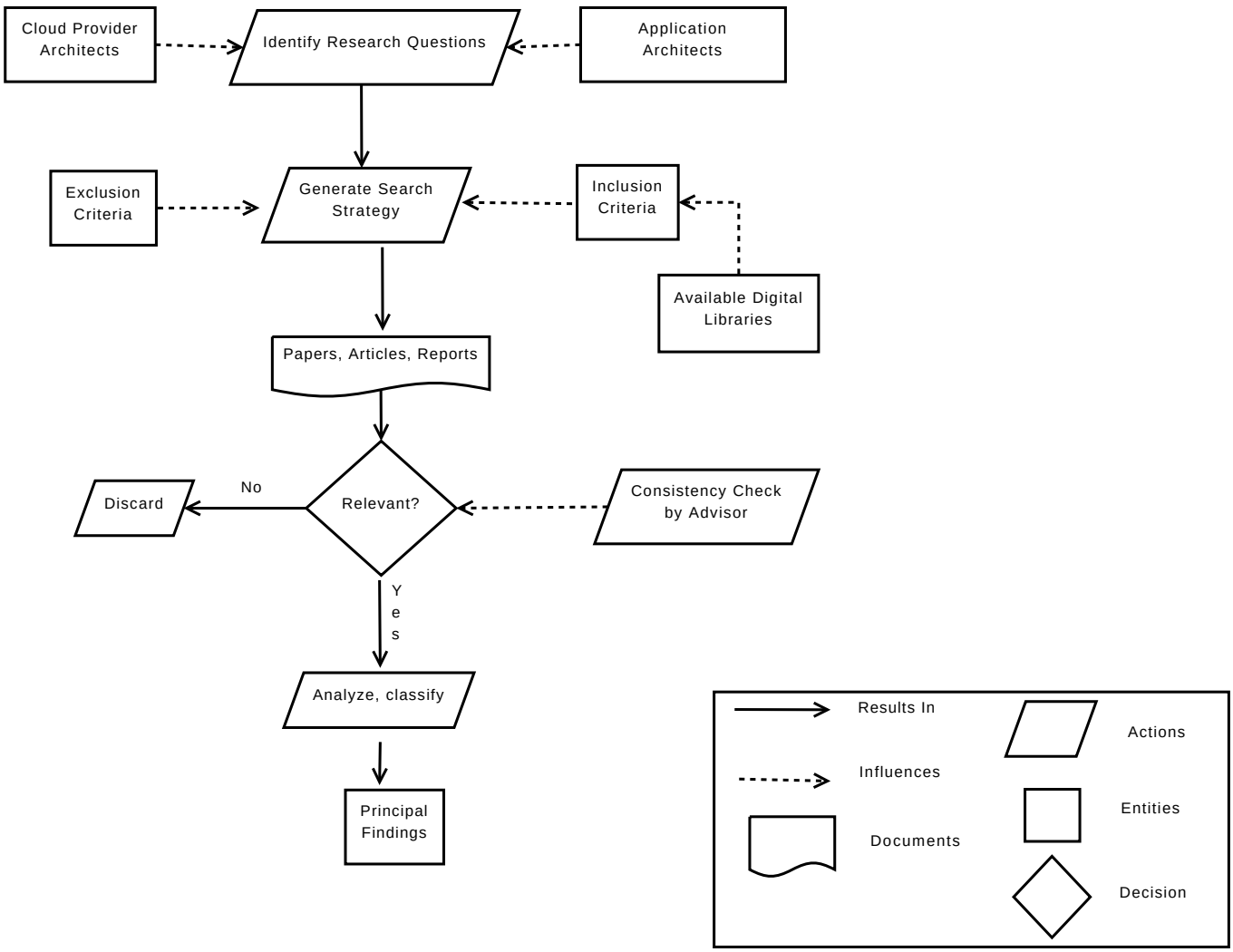


Figure 3.1: The literature review process

3.3 The Literature Review Process

Identifying the Research Questions: The most critical research question, from our perspective, is the following:

- What is the most scalable mechanism for QoS-based dynamic service composition?

  This question can be broken down into the following sub-questions:

  1. What are the variables that affect dynamic service composition?
  2. In testing for scalability, what ranges are typically chosen for these variables?

3.3.1 Systematic Review

We have already discussed the various mechanisms used by cloud-providers to dynamically allocate services to applications. Although, it would seem logical that techniques used by cloud-providers would have some overlap with answers to our current question, that is not the case. Indeed, going by current literature, the two areas seem to have no overlap at all, and authors of papers in both domains have no intersection. This is what makes our research valuable.

3.3.2 Research Questions

Service Composition as a mechanism could be considered to have started from the year 2000, since SOAP (the underlying protocol for exchanging structured information about Web-Services) was submitted to the W3C (World Wide Web Consortium) in that year. Service-Oriented Architecture as a style, arises from the ability of a service to be dynamically found, and bound into an application. This essentially allows an application, at runtime, to find a service that fulfills its needs, and bind to it. This technical possibility has opened up immense business possibilities, in that, an enterprise can now tailor-make an application to suit its

circumstances.

One of the opportunities is that an enterprise can now focus on improving its core strength, while still serving its customers fully. For example, a company that specializes in providing custom data mining algorithms can still bind, *at runtime*, to external storage, bandwidth, data-filtering, indexing and graphing services to provide a full data-mining solution, whilst concentrating on its own implementation of faster and more accurate clustering or rule-mining algorithms. This phenomenon is what we refer to as ***dynamic service composition***. The key word here is, *dynamic*. This means, that at design-time the application merely knows the functional characteristics of the services that it is going to bind to. The actual runtime attributes of the application like performance, reliability, availability are all determined by the specific service that it binds to.

There are several aspects to consider, while composing a service. First, the primary concern of the application is the semantic match between the application's requirement and the service's provision. Second, the interface expected and published must be the same as the one that the service actually implements. Third, the qualities promised by the service, such as performance, availability, throughput, reliability (hereinafter referred to, in this thesis, as QoS attributes) must satisfy the application's desire to exhibit a certain quality level. We are concerned with the third problem, in this thesis.

3.4 Research Method

Kitchenham[56] created guidelines for the use of a systematic method for literature review in Software Engineering. A systematic review is a powerful mechanism for establishing the state-of-the-art in any research area. It helps to: (1) collate main ideas in existing research (2) identify gaps in current research, and (3) map new research activities into an existing framework. We feel that our research question is best answered by adopting this method. According to the guidelines in [56], a systematic review consists of the following three phases:

(a) planning the review (b) conducting the review (c) reporting the review.

3.4.1 Review Protocol

The review protocol is a specified plan-of-action that is created *before* the review is conducted. This enables the researcher to look for flaws in the procedure and ascertain if the procedure is capable of answering the research question. The main components of a review protocol are: (a) specifying the data sources (b) laying down a search strategy to obtain literature (c) specify a selection strategy to filter relevant results (d) use a specific data extraction methodology, and (e) perform data synthesis. The importance of the review protocol lies in laying bare the precise steps followed by the researcher, which can then be studied, criticized and/or replicated. Now, we explicate the main components of our review protocol.

3.4.1.1 Data Sources

In determining the relevant papers for our study, we followed the following process:

1. Make a list of data sources
2. Perform a keyword search on each of these sources
3. Remove duplicates, presentations, work-in-progress
4. Divide remaining papers into: “relevant" and “not relevant"
    (a) Since titles and abstracts can sometimes be misleading, read title, abstract, introduction and conclusion to filter
5. For all relevant papers, read methodology, experiment design and analysis to filter papers that address scalability
6. Analyse all papers that address scalability

In the realm of software engineering, both [56] and [39] agree that electronic sources are sufficient in the domain of software engineering. Hence, we used the following search engines:

1. IEEE Explore
2. ACM Digital Library
3. Science Direct
4. Engineering Village (which searches INSPEC as well as EI Compendex)
5. ISI Web of Knowledge
6. Google Scholar
7. CiteSeer

3.4.1.2 Search Strategy

To start with the largest possible pool of papers, we searched all of the data sources for the following keywords:

1. Quality of Service – variants: QoS, QoS-aware, QoS-enabled,
2. Web Service Composition – variants: WSC, Service Composition, Service-based, Service-Oriented, Service-based Architecture, Service-oriented Architecture, Service-selection
3. Dynamic – variants: Adaptive, Adaptation, Self-adaptive, Self-optimizing, Self-healing, Self-managing

A sample search string used is as follows: `((((“dynamic web service composition") OR “automatic service selection") OR “service composition") AND QoS)`

Inclusion Criteria To form the initial pool, we accepted conference papers, journal articles, workshop papers, and technical reports.

Exclusion Criteria We had to carefully craft the exclusion criteria to ensure that ideas that were relevant were not filtered out, and yet papers that did not address our central question, did not end up in the final pool. To this end, our exclusion criteria was as follows:

- **E1**: Papers that did not deal with dynamic service selection
- **E2**: Papers that were domain-specific
- **E3**: Papers that had extensions in journals
- **E4**: Papers that were published before the year 2000
- **E5**: Papers that were not published in English
- **E6**: Duplicate references
- **E7**: Papers that could not be obtained / behind paywalls / inaccessible

At this point, we also had to unify vocabulary used inside the papers. Sometimes authors use different terms to refer to the same thing. For example, some authors use *tasks*, while others use *service class* and yet others use *abstract service* to refer to the service-functionality inside an application. The unifying terms that we used are as follows:

AbstractService: The functional specification of a certain task. This is also sometimes referred to, as a *task* or *service class* or *abstract service* in a Workflow.

CandidateService: An implementation of an AbstractService. Each CandidateService has a QoS that it advertises through its SLA. This is also referred to as *service candidate* or *concrete service*. This forms the pool of services that an architect selects from, for the application.

Workflow: An architect composes several AbstractServices into a Workflow, to create an application.

3.4.1.3 Study Selection

Relevance

- **E8**: Papers dealing with technical improvements to the underlying infrastructure like SOAP, WSDL, BPEL, DAML-S, OWL-S (No particular QoS mentioned in the paper)
- **E9**: No mechanism for QoS evaluation mentioned
- **E10**: No mention of Workflow or AbstractServices in Workflow
- **E11**: No mention of CandidateServices for an AbstractService

Consistency Check The classification of papers into “Relevant" and “Not Relevant" is a vital part of the review. By discarding papers that are relevant, one might miss out on novel ideas. On the other hand, considering papers that are concerned with other topics like BPEL (for instance) is potentially misleading. To ensure that papers were being classified correctly, random samples were extracted from the initial pool and given to other researchers for blind classification. If there was a disagreement in classification of a paper, as being relevant or not relevant, that paper was discussed in person, to arrive at a consensual decision.

3.4.1.4 Justification for Exclusion Criteria

Most search-engines use keywords or categorization terms to return as many results as possible. The creation of exclusion criteria is essential, so that only papers that are concerned with the same topic are compared. Also, without an exclusion criteria, the number of papers that merely match keywords is likely to be so huge as to defy sensible comparison. We believe that the first set of exclusion criteria (E1 – E7) are self-evident. They help in filtering out results that are returned due to mis-categorization, or are only tangentially related to the topic of dynamic web-service composition. The second set of exclusion criteria are more subtle. Papers that do not get filtered out through E1–E7, are mostly related to our topic. However,

there are still papers amongst them that cannot be used in comparing techniques against each other.

1. **E8:** Papers that discuss improvements to SOAP, WSDL, etc. are mostly related to the problem of efficiency in service-description, and bandwidth usage in transferring service ontologies. While advances in these technologies would definitely increase the speed of transferring service-descriptions, they would not address the actual problem of matching services based on their QoS. Improvements in description languages themselves, like DAML-S and OWL-S, could contribute to matching services. Amongst these, we only eliminated those that did not mention QoS matching at all.

2. **E9:** For determining which CandidateService is a match, and then determining the best one out of the matched set requires some mechanism of QoS evaluation. Merely using cost as the determiner is not enough, to be applicable to the problem that we're solving. Therefore, papers that did not propose a mechanism to evaluate a service on the basis of its advertised QoS, were eliminated.

3. **E10:** Service-based applications are composed together in a Workflow. It is the structure of this Workflow, that gives rise to the end-to-end QoS exhibited by the application. The same set of services, composed differently, would result in a different end-to-end QoS. We are concerned with the QoS exhibited by the entire application, and not the individual service. This implies that a mechanism that does not take into account, the structure of the Workflow, will not solve our problem. Hence, papers that did not mention any notion of a Workflow or structure of tasks in a Workflow, were removed from consideration.

4. **E11:** Our problem involves choosing the right service from amongst many CandidateServices. The problem lies in the fact that there are many CandidateServices, and that making the wrong choice would result in possibly either over-shooting the application's budget or getting the wrong QoS. Papers that did not address the issue of multiple CandidateServices for one AbstractService, were therefore removed from the study.

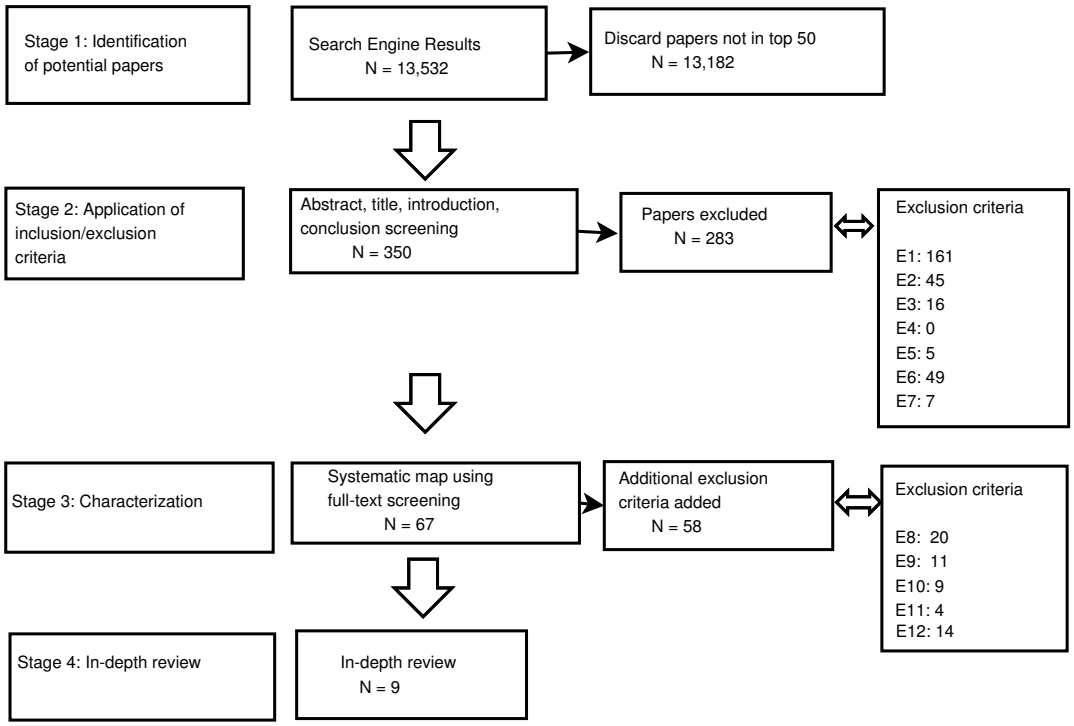


Figure 3.2: Selection of papers according to inclusion and exclusion criteria

3.4.1.5 Data Extraction

The reference details of each of the papers selected for systematic full-text screening, after the first stage of exclusion criteria was applied, was stored using Mendeley (www.mendeley.com). Kitchenham and Charters[55] suggest using *reciprocal translation* to arrive at a summary about similar studies, “by translating each case into each of the other cases". This allows us to compare method descriptions across papers and arrive at common axes. For instance, “AbstractService", “service class" and “service task", all refer to a task in an application’s Workflow. Similarly, “Quality Attribute", “Quality of Service attribute" both refer to the same concept. We can apply reciprocal translation to unify all of these terms. After applying reciprocal translation, we determined that the common axes were:

1. Number and type of QoS attributes considered,

2. Technique used for QoS evaluation,

3. Size of the Workflow,

4. Number of CandidateServices per AbstractService in Workflow,

These were recorded in a spreadsheet, along with their year of publication and abstract.

3.4.1.6 Data Synthesis

After identifying the pool of papers for full-text screening, we applied an additional set of exclusion criteria (E8 – E11), to determine whether the paper was a candidate to be considered for evaluating scalability. There were many papers that referred to a specific case study, and did not make an attempt to evaluate their solution with different configurations of Workflow size, number of QoS attributes, etc. While this does not necessarily mean that their method is unscalable, lack of any mention on the authors' part led us to believe that this had not been a major consideration in that paper. We must emphasize that, we are not engaged in evaluating each solution and coming to a conclusion about its particular scalability. Rather, we are interested in surveying the research landscape for techniques that have been used to claim scalability of approach.

3.5 Overview of Included Papers

We now explicate on the surveyed papers and the process and rationale of applying the exclusion criteria. The initial keyword search from the search engines yielded 13,532 papers. This number would clearly be infeasible to read and digest. We arbitrarily selected the top 50 papers from each of the search engines' results and created the initial pool. This obviously runs the risk of missing out papers that are relevant but are positioned lower in the search results. Then we studied whether papers appearing in the 40-50 range of results were being classified as “Relevant" or “Not Relevant" . We observed that in only one case (with ScienceDirect) did we actually classify a paper ranked at 41 as “Relevant". On this data source, we checked the next 20 paper results, to ensure that no more relevant papers were missed. We also randomly

sampled 3 papers each, from each of the data source's list of results, beyond the 50-paper mark to check if we had missed a relevant paper. While the risk of missing a paper can never totally be eliminated, we believe that our sample is fairly representative of the main ideas in the field.

After collecting all the papers yielded from the data sources, we ran a duplication check and a extension check. A lot of papers were reported in multiple sources and hence were removed. There were also journal articles that were extended versions of papers that had appeared in conferences (exclusion criteria E3). In these cases, we considered only the longer, journal article since it had more details about the method used. After applying exclusion criteria E1, E3, E4, E5, E6 and E7, we were left with 122 papers. We then read through introduction and conclusion to apply exclusion criteria E2. This narrowed the pool down to 67 papers.

Some papers amongst the 67 selected for full-text screening did not mention QoS in the service selection process. The distribution of papers that mentioned QoS in their service selection process is shown in Figure 3.3. Eliminating the ones that did not mention any QoS at all (exclusion criteria E8), left 47 papers. We then filtered those papers that mentioned QoS attributes in the paper, but did not provide a method for evaluating a service based on the QoS values described by a service (exclusion criteria E9). The results are shown in Figure 3.4. Application of this criteria left 36 papers. Application of exclusion criteria E10, resulted in eliminating all the papers that did not use the idea of composition of an application as a Workflow, with AbstractServices. This left 27 papers in the pool (Figure 3.5). Applying the final criteria, E11, resulted in the elimination of papers that did not mention the notion of choosing CandidateServices from a pool, for an AbstractService. This left 23 papers in the pool, as shown in Figure 3.6.

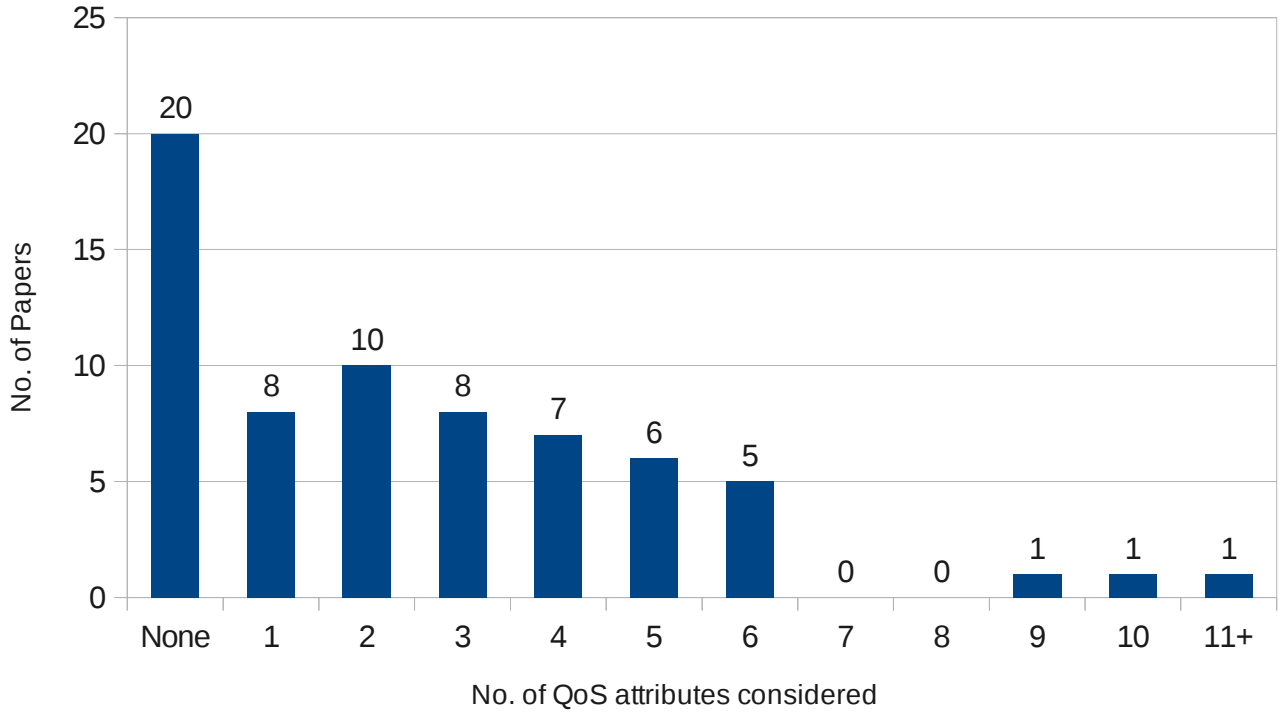


Figure 3.3: Number of QoS used

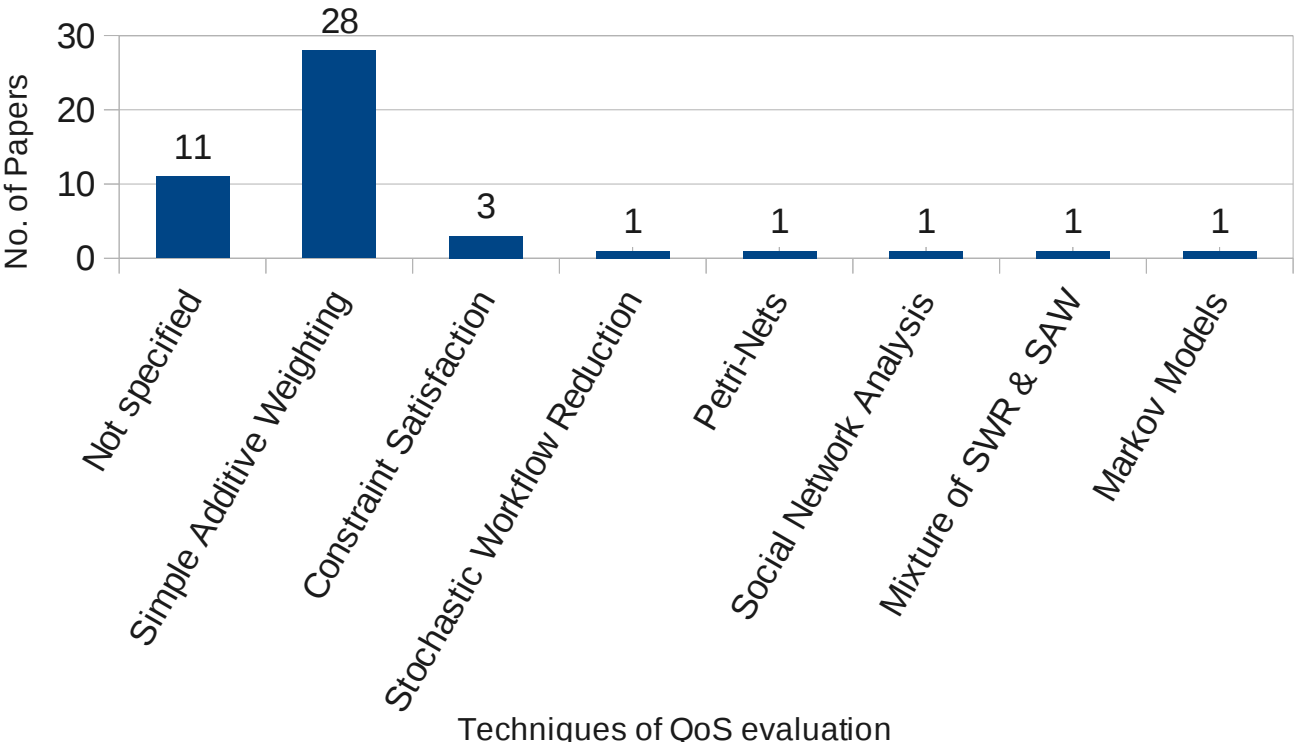


Figure 3.4: Number of ways of measuring QoS

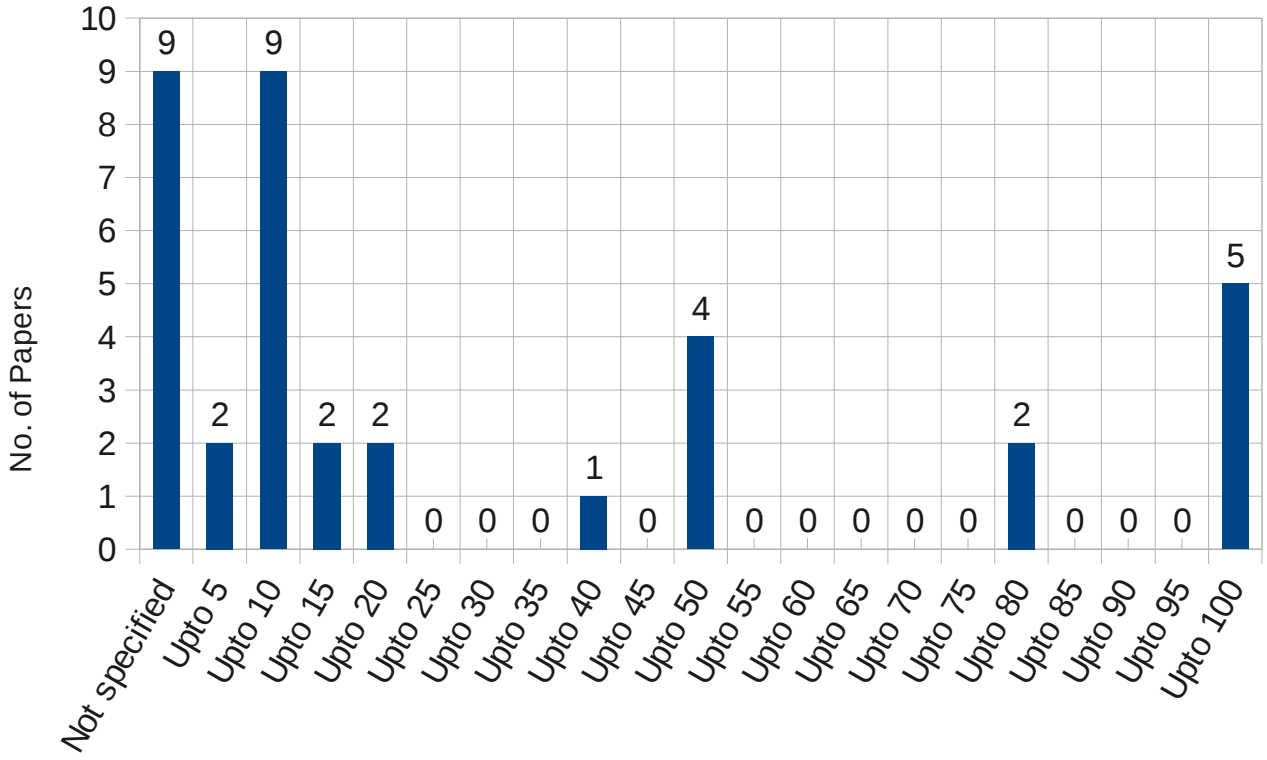


Figure 3.5: Number of AbstractServices per Workflow

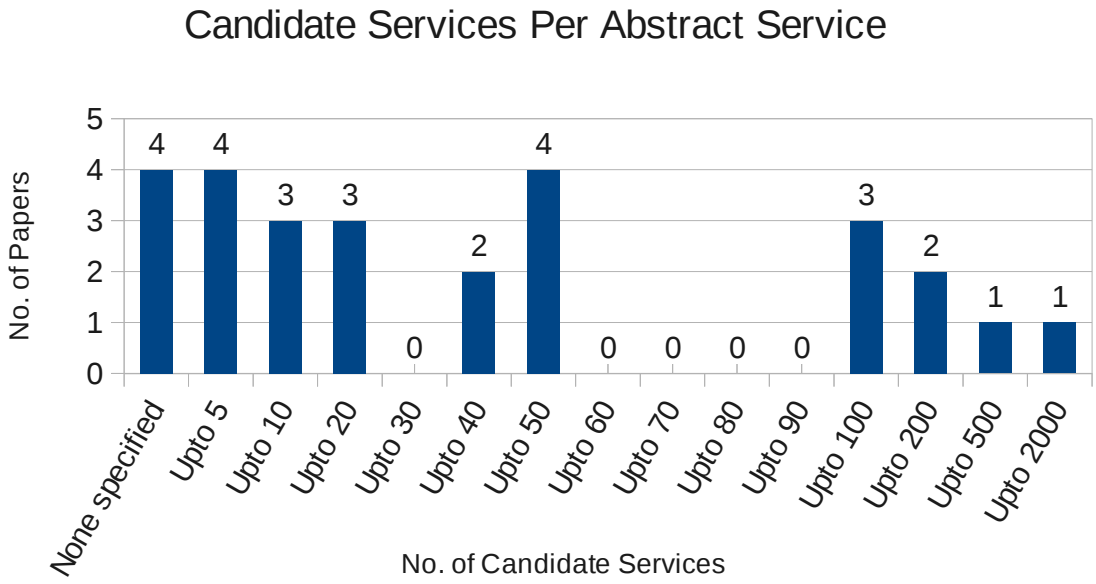


Figure 3.6: Number of CandidateServices per AbstractService

3.6 Results of Systematic Review

> *I examined aspects of scalability, but did not find a useful, rigorous definition of it. Without such a definition, I assert that calling a system "scalable" is about as useful as calling it "modern". I encourage the technical community to either rigorously define scalability or stop using it to describe systems* – Mark D. Hill [44]

Hill's quote, though more than 2 decades old is still as significant as ever. In none of the papers that we surveyed, did we find a single rigorous definition of what it would mean for a dynamic service composition technique to be scalable. In the absence of such a definition, it is difficult to judge whether a technique is really scalable, or worse if authors have considered scalability at all in their analysis. This would render a review on scalability, quite open to criticisms of capriciousness. To alleviate this, we use a definition of scalability from Duboc et al: *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*[31].

According to the definition in [31], when certain (system or environment) variables change, they have an impact on the system's qualities. In our case, the system quality that we want to measure is performance, measured as the amount of time taken to select services that meet all the cost and QoS constraints. The performance of a technique could vary, as the variables change over some range. In the ideal case, all the techniques being compared would have a good implementation, be measured on a comparable platform, with the same dataset. However, this is almost never achievable in reality. The notion of complexity class allows us to do away with implementation-dependent issues, however not all papers report the complexity of their techniques.

In such a situation, the best we can do is compare the *reported* change in performance, as the relevant variables change. Wherever possible, we also provide information about the complexity of the technique used. To compare papers against the definition in [31], we added another exclusion criteria (E12), which considered whether the authors had carried out experiments testing their technique against various values of Workflow size, size of CandidateService set, etc. Papers that had not considered varying any of the variables were deemed

not to have *tested* for scalability. Application of E12 reduced the set of papers for in-depth review to 9. **Note: All values are inferred from graphs published in their respective papers.** A comparison of their QoS and core ideas is shown in Table 3.1

Author	QoS evaluation	QoS Considered	Core Idea	Centralized	Complexity Reported
[8]	SAW	Cost, Execution time, Reputation, Reliability, Availability	Discusses both global planning as well as local optimization algorithm. Most cited amongst all others for QoS evaluation.	Yes	No
[36]	SAW	Cost, Response time, Reliability, Availability	Represent as 0-1 knapsack problem and then solve using integer programming	Yes	No
[107]	SAW	Execution time, Price, Reliability, Availability, Reputation	Create a convex hull of multi-dimensional QoS points; sorting the frontier segments according to their gradients, allows one to pick near optimal points.	Yes	$O(nmL+nLlgL+nLlgn)$
[110]	SAW	Randomly generated QoS values	Uses both MMKP and graph representation. Presents WS_HEU, an extension of HEU	Yes	$O(n^2(l-1)^2m)$
[24]	SAW	2 to 5 (randomly generated QoS values)	Use a rule library to cut down on candidate solutions; iteratively pick the best service that does not violate constraint or decrease utility	Yes	$O(n^2(l_r-1)^2m)$ where $l_r < l$, after using rule library
[58]	SWR-like formula	Cost, Execution time, Availability, Reliability, Reputation, Frequency-of-use	Use tabu search and simulated annealing to generate candidate composition plans	Yes	No
[1]	SAW	Only consider negative QoS criteria, so only upper bound constraints	Decompose global QoS into local values, use mixed integer programming to select best mapping of global constraints to local levels	No	MIP's: $O(2^{n \cdot l})$; local utility: $O(l)$. Decision variables in MIP: $n \cdot m \cdot d$
[62]	SAW	Cost, Availability, Response time, Reputation, Throughput, Transaction_support, Fault_tolerant_support, Encryption_support, Location, Service_provider	Decompose global QoS into local values; select few from distributed QoS registries; perform central optimization at the final stage	No	Communication: $7 \cdot e$. Data sharing cost is $n \cdot (\alpha+1)$. Central optimization done using MIP, decision variables is $n \cdot e \cdot (\alpha+1)$
[2]	SAW	Cost, Execution time, Reputation	Represent as a DAG; individual brokers select a concrete service for each flow; each flow modelled as M/G/1 arrivals of service requests; pick concrete service using non-linear optimization	No	$O(max\{\mid K \mid \cdot lg \mid K \mid ,\mid K \mid \cdot max_{k \in K} \mid \vee^k \mid \cdot max_{k \in K, i \in \vee^k} \mid J_i^k \mid^2\})$

Table 3.1: Comparison of techniques

Benatallah et al.[8] were amongst the early papers on dynamic web-service composition. Their usage of Simple Additive Weighting(SAW) from Yoon et al.[109], for calculating the utility value of each QoS attribute, has been adopted by many subsequent papers. They represent the Workflow of the application, as a statechart (which can be unfolded to remove

loops) and compare the use of local optimization vis-a-vis global planning using integer programming (IP). They consider 5 QoS attributes (cost, execution time, reputation, reliability and availability) and 1 global QoS constraint. Although, in their experiment description they vary both the Workflow size (WS) as well as the number of CandidateServices, the graphs shown in the paper only show figures for 10 & 40 CandidateServices. In table 3.2, we show the figures reported for global planning using IP in a dynamic environment.

Candidates / Workflow size	WS–10	WS–20	WS–30	WS–40	WS–50	WS–60	WS–70	WS–80
10 CandidateServices	<1s	3s	6s	11s	22s	32s	54s	89s
40 CandidateServices	5s	10s	15s	30s	50s	90s	140s	190s

Table 3.2: Performance of global planning using IP in a dynamic environment

Gao et al. [36] model the dynamic web-service selection problem as a zero–one integer programming problem and report results with 4 QoS attributes (cost, response, reliability and availability), along with capacity and load.

Yang et al. [107] propose modelling QoS scores of each candidate web-service (along with its resource consumption) in two-dimensional space and then construct a convex-hull to identify a near-optimal solution to the service-selection problem. With 5 QoS attributes (execution time, price, reliability, availability, reputation), they provide experimental data with zero, single and multiple QoS constraints. They also identify the worst-case complexity of their technique as $O(nmL + nLlgL + nLlgn)$ where $L = max(l_i)$ and l_i is a CandidateService, n is the Workflow size and m is the number of QoS constraints.

Yang et al. [107] and Gao et al. [36] come the closest amongst all the papers, to comparable experimental data. Although in Figure 3.7, we see that even they don't have an exact overlap of data points provided. Figure 3.7 is based on data provided in their respective papers, *without* being normalized for dataset, memory size, computational power, etc. The figure is only intended as an indicator of the range of variables being used.

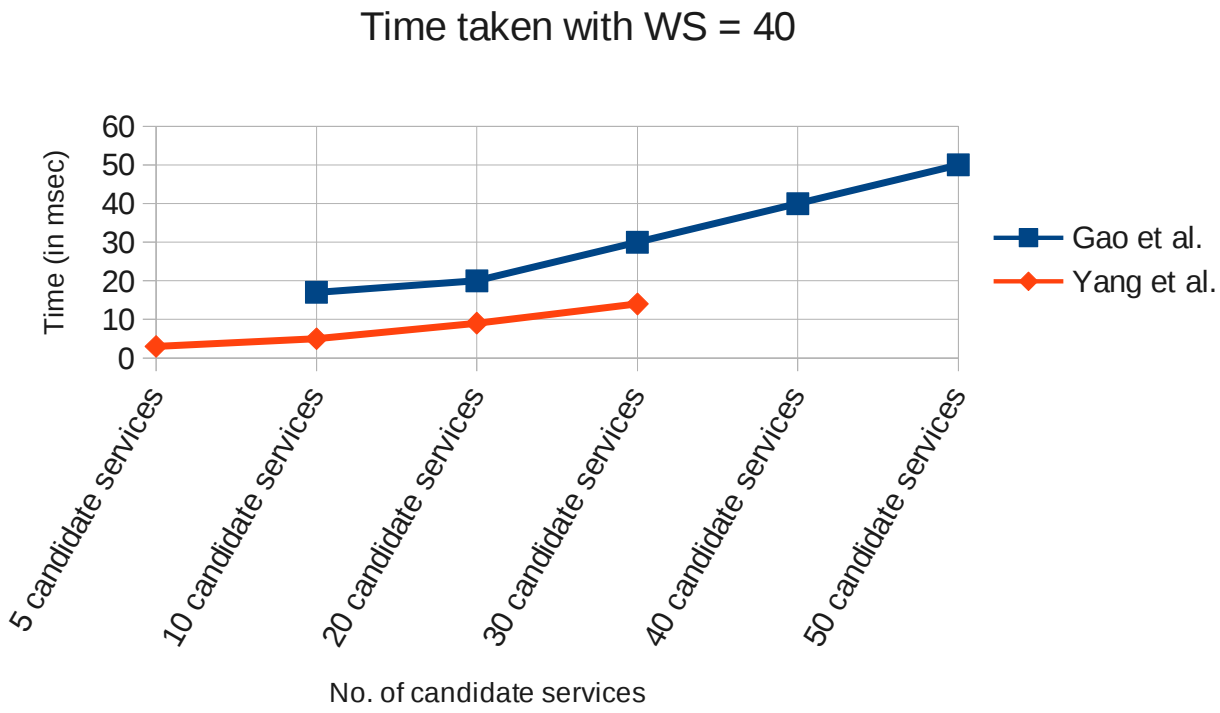


Figure 3.7: Workflow Size = 40

Yu et al.[110] model the problem in two different ways: (a) as a multichoice, multidimension knapsack (MMKP) problem (b) as a graph (solving for a multiconstrained optimal path). The first approach uses a *branch and bound* algorithm, BBLP[53], to find the optimal solution. The second approach (WS_HEU) is an extension of HEU[54], which is a heuristic algorithm for solving the knapsack problem. WS_HEU prunes out more infeasible services from the candidate set, and always selects a feasible solution (if one exists), thus is faster than HEU by about 50%. Yu et al.[110] do not provide timing data of their experiments. Instead they provide comparative statistics *vis-a-vis* HEU and BBLP. While this is a quite useful way to benchmark against existing algorithms, evaluating future techniques or newer approaches against [110] becomes difficult. WS_HEU is identified as having a complexity of $O(n^2(l-1)^2m)$

Chen and Yao[24] also model service selection as a MMKP, and their approach is influenced greatly by [110]. However, Chen and Yao advocate a rule-based approach. In this method, iterative application of rules constrain services that could not possibly meet QoS constraints,

from being selected. This cuts down on the search space of CandidateServices and iterative improvements are made to feasible set, to reach a near-optimal solution. Chen and Yao's approach to evaluation is the same as [110]. In fact, all of their experimental evidence is benchmarked against WS_HEU. The authors identify the time complexity of their approach as being $O(n^2(l_r-1)^2m)$ where $l_r < l$, after using rule library, n is the Workflow size and m is the QoS constraints.

Ko et al.[58] propose a hybrid algorithm, combining tabu search and simulated annealing techniques, to find a composition plan that does not violate any given QoS constraint. However, they only consider constraints on individual QoS attributes and not on a global basis. They generate an initial plan by considering reputation and frequency as the most important of all QoS attributes, and assigning the candidate with the highest reputation score to the abstract task. Neighbour plans are generated by sorting unsatisfied QoS constraints and finding a candidate that can satisfy those constraints, in order of priority of constraint. In the experimental evidence, only coarse-grained time information is given, in seconds (from the graph published in the paper). In Table 3.3, we show the approximate timings, gleaned from their paper.

Candidates / Workflow size	WS–10	WS–50	WS–100
10 CandidateServices	< 0.5s	< 0.5s	<1s
20 CandidateServices	<0.5s	< 0.5s	<1s
30 CandidateServices	<0.5s	< 0.5s	<1s
40 CandidateServices	<0.5s	<1s	<1s
50 CandidateServices	<0.5s	<1s	<1.5s

Table 3.3: Performance of hybrid algorithm in [58]

Alrifai and Reese[1] present a distributed approach to service selection. They first decompose global QoS constraints, using mixed integer programming, into local ones, and then perform local service selection at each of the distributed service registries, that are available.

Workflow Size	Time (in msec)
WS–5	50ms
WS–10	45ms
WS–15	50ms
WS–20	70ms
WS–25	60ms

Table 3.4: Distributed approach with 100 CandidateServices

Each service broker returns the best matching service from its respective service registry, for composition. Since each service broker performs selection for an AbstractService, independently of other services, the Workflow size does not affect the time complexity of the search. The time complexity of forming a composition plan is dominated by the integer programming used to decompose the global QoS constraints into local ones. MIP's time complexity in constraint decomposition is $O(2^{n \cdot l})$ and the number of decision variables in MIP is $n \cdot m \cdot d$ where 'd' is the levels of quality available. In Table 3.4, we report on the QWS dataset and also chose figures from the best performing of the hybrid curves shown in the paper.

Li et al. [62] are highly influenced by [1]. One of points of departure is the calculation of local QoS constraints. Instead of using MIP, they use non-linear programming to achieve a set of quality bounds. These bounds are then sent to each QoS registry for local service selection. Each registry then sends back a set of CandidateServices,that match the quality bound and a central optimizer uses MIP to choose from the set of returned CandidateServices. Since [1] and [62] are so close in the core idea of their technique, it would be quite insightful to compare their performance results. However, as seen in Table 3.5, the Workflow size and number of CandidateServices used do not match the ones reported by [1].

Ardagna and Mirandola[2] must deserve special mention for their systematic analysis of scalability of their approach. Not only do they identify the variables that vary and the range that they vary over, but the authors also identify [1] and [3] as the reference approaches to

Workflow Size	Time (in msec)
WS–5	10ms
WS–10	20ms
WS–15	40ms
WS–20	50ms
WS–25	70ms
WS–30	80ms
WS–35	90ms
WS–40	95ms
WS–45	100ms
WS–50	110ms

Table 3.5: Workflow size increases with 500 CandidateServices

Workflow Size / Candidate Services	10 Candidates	20 Candidates	25 Candidates	50 Candidates
100	8.10s	9.54s	9.98s	14.30s
1000	19.60s	144.30s	149.60s	451.30s
5000	444.90s	1000.05s	n.a.	n.a.
10000	970.15s	n.a.	n.a.	n.a.

Table 3.6: Per-flow optimization times (sliced from Table 6 in [2])

benchmark against. In this paper, instead of optimizing on a per–request basis, the authors propose optimizing on a per–flow basis. In this approach (first proposed in [21]), optimization is not triggered every time a composite service is requested. Rather, it is triggered only when significant events like non-availability of a selected service or change in QoS exhibited by a service, happen. Due to per–flow optimization however, the figures reported are on different Workflows. Ardagna and Mirandola[2] allow for the concept of *AbsShare*, which is the percentage of AbstractServices that are shared across different Workflows. This is a relevant notion, but completely un-addressed by any of the other approaches.

3.6.1 Revisiting the Research Questions

From an examination of all the papers selected for full-text screening, we can infer the set of variables that affect the scalability of each of the techniques mentioned. Each of these variables affect each technique differently. These variables are as follows:

1. Number of AbstractServices in a Workflow (Workflow size)
2. Number of CandidateServices considered per AbstractService
3. Number of QoS attributes (with constraints) considered per CandidateService
4. Number of constraints considered on both: per-abstract-service (local) as well as end-to-end basis (global)

Unfortunately, as we saw from the review, there are no commonly used scales or variable ranges used, for reporting scalability information. In fact, most techniques do not even report their performance numbers on all of the variables above. Even when techniques do report the same variables, they use different scales. In such a situation, comparing techniques amongst each other become difficult. Based on the literature surveyed, we recommend that while reporting timing information, the variables in table 3.7 be used. Needless to say, that the whole range of values for these variables should be reported. The upper-limits on these ranges are the median numbers of the papers, before applying the exclusion criteria, E12. Needless to

Variable affecting performance	**Recommended Range**
Number of AbstractServices in a Workflow	1–10
CandidateServices per AbstractService	1–20
QoS attributes (with constraints) considered per CandidateService	1–3

Table 3.7: Median value of variables affecting performance, reported in literature

say, that all variables are not equally important in every situation. Each domain has a different emphasis and differing circumstances. Depending on the the particular characteristics of the domain, a technique's scalability will differ. An application architect considering which technique to adopt would do well, to look at his specific case and ponder on which variables matter. If Workflow size is the most important variable, then the approach in [107] seems to be the most promising approach. Since Ardagna et al.[2] take a per-flow approach, it would be interesting to evaluate their approach against Yang et al.[107], with the same dataset. On the other hand, if the number of CandidateServices is expected to vary hugely, the approach

in [1] turns out to be quite promising. The number of QoS attributes vary across most papers, but the mechanism of evaluating QoS seems to have settled on Simple Additive Weighting. Stochastic Workflow Reduction proposed by Cardoso et al.[23] has also been used in a few papers, but is not as popular.

The number of QoS constraints makes a difference to the scalability, but it has not been systematically evaluated. It is the addition of constraints that makes a distributed solution to the service selection problem, difficult and interesting.

3.7 Discussion

3.7.1 Threats to Validity

Like Gu and Lago[39], we use the suggestion in Perry[81] to explicitly state the threats to validity that accompany any empirical study: (a) construct validity (b) internal validity (c) external validity.

Construct Validity Construct validity refers to whether the model constructed can actually answer the research question. Since our research question is intended to summarize the existing body of work, this threat is not applicable to our study. Also, any potential misunderstanding between the study designer and the study executor is avoided, since both roles are being performed by the same person. An explicit definition of scalability also allows readers to understand what we're looking for, in a paper and how we evaluate it.

Internal Validity Internal validity refers to the accuracy of modelling of the system under study, *i.e.*, do the variables being studied actually affect the dependent variables? To a large extent, the variables chosen for the study have emerged from the pool of papers themselves. In such a situation, the selection of the pool assumes great significance, as some variables might be over-emphasized and others de-emphasized, depending on the selection process.

To mitigate this cause of bias, we explicitly created a review protocol and followed it strictly. Since the study was performed by a single person, a PhD candidate, the supervisor was used as a third-party control. The supervisor randomly selected papers, both included and excluded ones, to verify whether the selection decision was correct or not. The arbitrary decision to take the first 50 papers from each search engine is admittedly difficult to justify. Any paper that ranked lower down in a particular search engine's ranking, but contained a scalable method for dynamic service composition, would be lost. Also, values inferred from graphs published in papers are approximate, due to the low resolution of such graphs. Hence, while these values are internally consistent, they are not necessarily exact.

External Validity External validity refers to the generalisability of the study, *i.e.*, do the conclusions of the study hold even outside the parameters of the study? We made a conscious decision to use only academic data sources. This automatically precludes any commercial report or presentation, that is not indexed by these sources. Also, our exclusion criteria excludes domain-specific papers and papers relating to improvements in SOAP, WSDL, OWL, etc. It is entirely possible that a commercial entity that has not published academically, has used a highly scalable technique. Or that improvements in the underlying infrastructure enable pruning of the search space of QoS attributes, such that current techniques are able to scale to a high level.

3.7.2 Quality Assessment

It is important to note that we have not attempted to attribute any degree of quality to any of the approaches claiming to be scalable. We have merely defined scalability in a more rigorous manner, than available in corresponding literature and then mapped existing approaches onto a common scale.

3.8 Conclusion

As a result of this survey, we were able to establish that scalability of service-selection techniques has not been rigourously evaluated. We then identified *four* variables on which the scalability of dynamic service composition depends. However, since different techniques report figures with different emphasis on each of the axes, it is difficult to draw conclusions across techniques. Apart from [2], which explicitly considered results from two different papers, none of the others do a systematic comparison of results from others. Even while comparing against *integer programming*, each paper would have its own implementation, and therefore its own results. Although, the notion of complexity classes provides an implementation-independent mechanism for evaluating the time complexity of a particular algorithm, it does not seem to be a popular metric to report. Also in this survey, we did not consider techniques that have been cited by other papers, as being scalable. For instance, Canfora et al.[20] use a GA-based approach and have been cited as a scalable technique. However, both [20] and [19] did not meet our definitional requirements of discussing experimental evidence which shows the technique being benchmarked over a range of operational values.

This survey provides an insight into variations and gaps in experimental evidence and reporting, that occurs in the domain of dynamic service composition. In the following chapters:

1. We review market-based methods for designing a self-adaptive mechanism,
2. We propose a market-based mechanism that remedies the unsystematic approach to scalability presented in this chapter and,
3. We provide concrete mechanisms to evaluate a particular resource for its QoS.

CHAPTER 4

Towards Self-Adaptive Architecture: A Market-Based Perspective

4.1 Introduction

Self-Adaptation as a concept, has been recognized for a long time. From the domain of biological systems[18] to human affairs[86]. Most natural systems exhibit this phenomena, viz, the effecting of change by a system to ensure that it continues to achieve the utility that it previously did. Different self-adaptive systems exhibit adaptivity in different ways. In human-designed systems, the first systematic efforts to create a self-adaptive system have been in the domain of control loop design. It has been a comparatively recent entrant in the field of software. Regardless of the type, most systems can be differentiated on the basis on where the locus of control for self-adaptation lies:

1. **Centralized:** In these type of systems, there is usually a hierarchy of components. Components at higher levels are responsible for goal management and planning for change [59], while those at lower levels are responsible for immediate action and feedback. Decision-making is concentrated in one or a closely related set of components. Cen-

tralized self-adaptive systems exhibit a communication pattern that is characterized by sensory information (data) flowing from dumb components to the central decision maker, and instructions (commands) flowing from the decision maker to the dumb components. 'Dumb'is used here, in the sense of a component having no awareness of itself and its effects on the environment. Predictable and cohesive response to change are advantages of this type of system. However, reaction times get slower and slower as the size of the system increases.

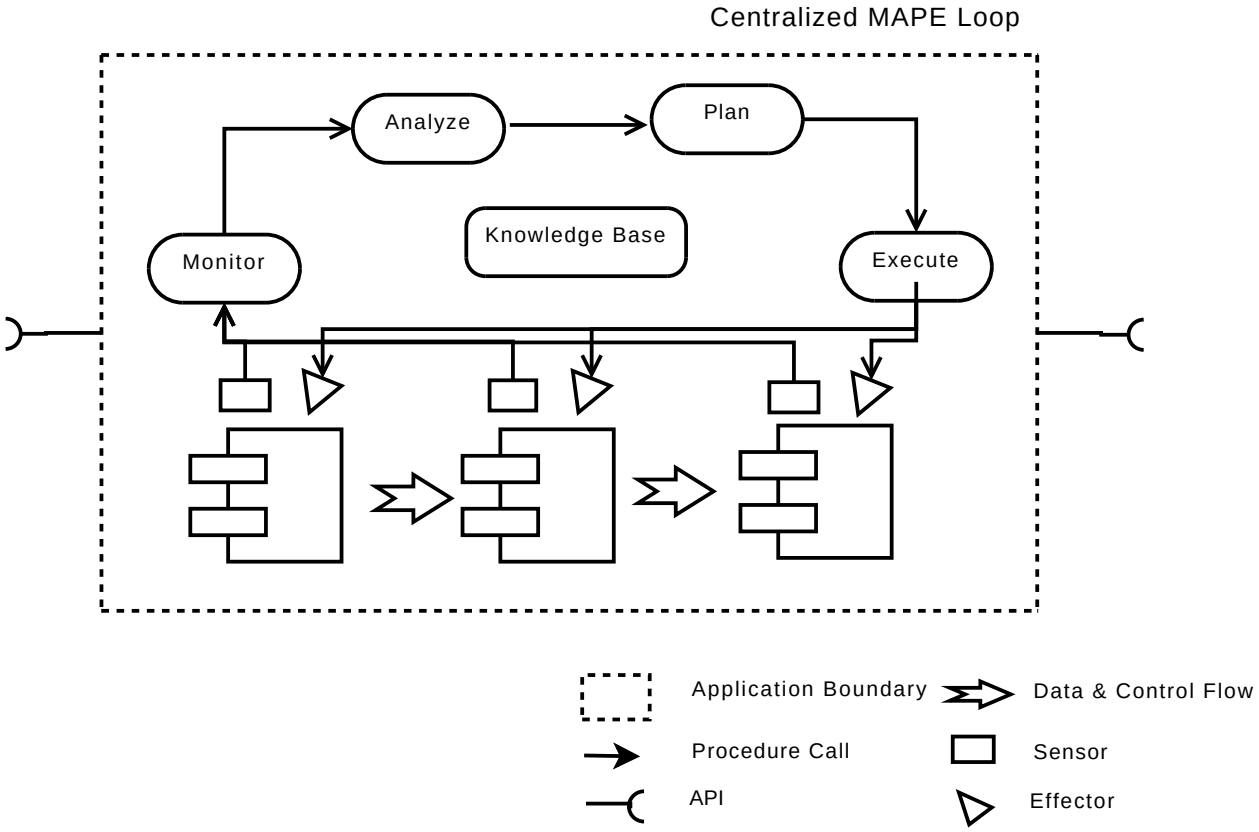


Figure 4.1: Self-adapting application with a centralized MAPE loop

2. **Decentralized:** On the other hand, decentralized systems do not have a hierarchy of components. Each component acts as an individual agent with its own goals, and its own perception of the environment. This has the advantage of quick reaction to change. But it also has the disadvantage of being fragmented. That is, different components may react differently to the same change stimulus. This could make the self-adaptation inefficient or even deleterious. The challenge in building a decentralized system is to

ensure that all the agents collectively move the system towards a common goal. This is usually accomplished by agents communicating amongst themselves. However, the lack of a centralized communication pattern means that the communication protocol must be decentralized and environment-based.

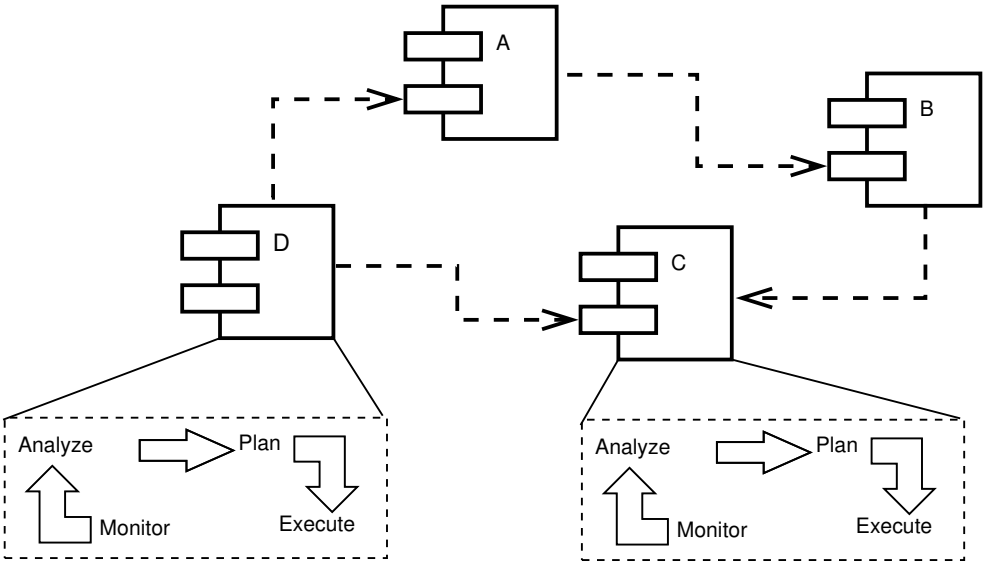


Figure 4.2: Decentralized MAPE loop in an application

In this chapter, we shall first explicate on decentralized self-adaptive architectures and then introduce our market-based mechanism in detail. We conclude the chapter with a reflection on the possible design alternatives for implementation of our mechanism.

4.2 Self-Adaptive Architectures

As software usage continues to grow and become increasingly pervasive in our daily lives, major commercial entities such as IBM warn of a software complexity crisis [47]. As the number of inter-connected devices increase, code interacts with other code in ways that have not been anticipated by its designers. Changing environments, requirements like 24/7 availability, are putting a strain on the best of software designers. Soon increased complexity will baffle even skilled administrators, and it will be impossible to make intelligent and timely decisions in the wake of rapid change. Most experts believe that software must increasingly

become autonomic in order to meet the demands being made of it. By autonomic, we mean computer systems that can manage themselves, given high-level objectives by administrators [77]

Autonomic computing is also frequently referred to by its characteristics, the so called self-* properties. [78] define eight characteristics of autonomic computing:

- **Self-configuration:** The system should be able to configure and re-configure itself under varying and unpredictable conditions.
- **Self-optimization:** The system should be able to detect sub-optimal behaviours and optimize itself to improve its execution.
- **Self-healing:** The system should be able to detect and recover from potential problems and continue to function smoothly.
- **Self Awareness:** An autonomic application/system 'knows itself' and is aware of its state and its behaviours.
- **Self Protecting:** An autonomic application/system should be capable of detecting and protecting its resources from both internal and external attack and maintaining overall system security and integrity.
- **Context Aware:** An autonomic application/system should be aware of its execution environment and be able to react to changes in the environment.
- **Open:** An autonomic system must function in an heterogeneous world and should be portable across multiple hardware and software architectures. Consequently it must be built on standard and open protocols and interfaces.
- **Anticipatory:** An autonomic system should be able to anticipate to the extent possible, its needs and behaviours and those of its context, and be able to manage itself proactively.

Although these are listed as distinct capabilities in [78], they can be viewed as complementary features, with each one feeding into and enabling the others. Specifically, they can be considered to be essential characteristics of long-lived applications. We refer to them collectively as self-adaptation properties. Self-Adaptation in software can be engineered on many dimensions: structural, behavioural, quality of service, etc. In this thesis, we focus on the Quality of Service (QoS) that an application exhibits. By QoS, we mean qualities like performance, reliability, security, maintainability, etc. These qualities are not related to the functionality of the system. Rather, they form the intangible user-experience that is almost as important, as the function itself. These qualities are experienced universally, be they small, student-written, toy systems to enterprise-spanning, world-facing, production-quality systems. Some of these qualities are measurable, like performance, while others are more difficult to quantify, like security.

In engineering a mechanism to adapt QoS for an application, we had to decide whether we would use a centralized adaptation mechanism or a decentralized one. The domain of the application that we focus on, is a service-based application resident in the cloud. Since a service-based application consists of potentially large number of, and disparate services, it seems reasonable to follow a decentralized approach. This leads to a further decision on communication pattern that such a decentralized system should exhibit. In literature, decentralized self-adaptive systems exhibit communication/coordination patterns of the following types[103]:

1. Gradient Fields
2. Market-Based Control
3. Digital Pheromone Paths
4. Tags
5. Tokens or key-based

Gradient Fields: Typically used in situations involving coherent, spatial movement of agents. It is a decentralized mechanism to achieve propagation of spatial and contextual information amongst large groups of agents. This form of coordination takes its inspiration from physics and biology. Particles in our physical universe, for example, adaptively move and self-organize, on the basis of locally perceived gravitational/electro-magnetic fields. Implemented in software systems, all agents have access to an environment, which provides the numeric value of a *GradientField*. The GradientField is a data structure that contains location-dependent contextual information. The strength of the GradientField guides the agent in the direction to move. The environment takes the responsibility for propagation of information and this frees up the agent from having to do any exploration, on its own. This also allows for robust and flexible coordination in a dynamic environment.

Market-based Control: This communication pattern is typically used in efficient, decentralized resource allocation. Multiple autonomous agents act in a self-interested manner, to achieve a globally coherent goal. Taking inspiration from human-economies, each individual agent participates as a buyer or a seller of goods. In a software setting, the goods might be *cpu-cycles*, *bandwidth*, *disk-usage* or any other scarce resource. The underlying idea is that market equilibrium represents the optimal allocation of resources, achieved through local computations and no global coordination. This is a flexible and robust mechanism, that reacts swiftly to changes in resource levels.

Digital Pheromone Paths: In this pattern, agents explicitly search for goals and leave trails for other agents to follow. Depending on the number of agents that follow a particular path, its attractiveness increases for all succeeding agents. In this manner, tasks like shortest-path-routing can be done with relatively little overhead of computing at the level of the individual agent. This type of communication takes its inspiration from ant colonies, where ants quickly communicate the shortest path between a food source and the nest, by the use of pheromones. This type of communication is robust to failure and changes in the global state.

In a computational implementation, agents sense the strength of the pheromone from the environment, and the environment is responsible for tunable parameters like evaporation and aggregation.

Tags: Inspired by social phenomena of cooperation and specialization, this coordination pattern rewards altruistic behaviour. Thus, even self-interested agents find that it is better to cooperate than to exploit other agents. This is achieved through the use of *tags*, which are externally settable and observable labels. Agents will tend to select other agents that share the same tag value as themselves. In an environment where cooperative work tends to yield a higher utility, this mechanism results in a sort of tribe formation as agents with identical tags cooperate. Coordination emerges because agents discriminate on tags. A cooperative tribe grows larger and larger, as more agents copy the cooperative strategy to gain a higher utility. A selfish agent can exploit such a situation, but the tribe withers away quickly leaving the selfish agent with other selfish agents. This process mirrors the evolution of formation, growth and destruction of tribes. Coordination happens directly between agents, and therefore no environmental mediation or infrastructure is required. This is a highly decentralized and scalable model for solving problems like "tragedy of the commons" in a peer-to-peer filesharing system.

Tokens: This communication pattern does not have a parallel metaphor in nature or human societies. Tokens, like tags, are arbitrary pieces of information, held in a data structure. However, unlike a token, there is only one or a limited number of tokens within a group of agents. The agents holding a token are allowed to have access to a resource, or information, or perform a specific role. Information on the token can be updated by the agent holding the token. This allows for an emergent history of the path of the token through the group, which can further be utilized in future token-passing decisions. The only infrastructural support needed by this mechanism is the presence of some communication channel, to pass tokens. A fully decentralized mechanism, the communication overhead does not rise with increase in

group size and hence, is very scalable.

4.2.1 Choosing the right mechanism

Each of the communication patterns outlined have advantages and disadvantages. Choosing amongst these alternatives is therefore, an exercise in making trade-offs. We look at salient characteristics of the problem domain and map how well these patterns fit those characteristics.

Scalability of technique: Matching service providers in the cloud to service consumers, requires that communication overhead does not increase proportionally with the number of agents in the environment.

Confidentiality: Since different service providers and consumers belong to different commercial entities, the mechanism should not demand that agents reveal their decision criteria. Also, it cannot expect agents to depend completely on signals from the environment.

Natural fit to problem: Some patterns encourage altruism, whereas others work in competition (self-interested agents). Given that service providers and consumers belong to different commercial entities, it is reasonable to expect that their agents would be self-interested.

Ease of Implementation: Each pattern requires different kinds of infrastructural support. For e.g., Gradient-Field and Digital Pheromones are dependent on a suitable middleware for the right kind of signals. Whereas, Market-Based Control, Tags and Tokens require only a communication channel, for agents to function.

As can be seen from table 4.1, Market-Based Control fulfils more criteria than any other decentralized communication pattern, in the context of our problem. We now proceed to

	Scalability	Confidentiality	Natural Fit	Implementation
Gradient-Field	✓	✓	–	–
Market-Based Control	✓	✓	✓	✓
Digital Pheromone	✓	–	–	–
Tags	✓	–	–	✓
Tokens	✓	–	–	✓

Table 4.1: Mapping mechanism characteristics to problem domain

examine this mechanism in more detail.

4.3 Market-Based Control

Market-based techniques have been used previously for solving distributed resource allocation problems. The field of computational economics is concerned with economics-inspired techniques to guide interactions between components of a distributed system. Although inspired by human macro-economic theories, they do not depend in any way on real macro-economic behaviour. A key reason for this, is that agents in computational systems can be programmed to behave in any manner, from completely selfish to completely altruistic.

Tucker et al[93] present a good survey of market-based techniques used in the software domain. Notable examples include Clearwater's bidding agents to control the temperature of a building[25], Ho's center-free resource algorithms [45] and Cheriton's extension to operating sytems to allow programs to bid for memory[41]. Other examples include distributed Monte-Carlo simulations[98], distributed database design using market-methods for distributing sub-parts of queries[89] and proportional-share resource management technique[99]

4.3.1 Auctions

Auctions, specifically Double Auctions (DA), have increasingly been studied in Computer Science, as a mechanism of resource allocation. Daniel Friedman [34] 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, and there has been much work on the design space of market-institutions[105, 71], bidding strategies [26, 83], agent-based implementations of traders[52, 42, 43, 72], etc. Gupta et al[40] 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.

A Double Auction is a two-sided auction, *i.e.*, both the buyers and the sellers indicate the price

that they're willing to pay and accept, respectively.

4.4 Review of Literature using MBC

There has been a lot of work in applying market-based techniques to solving computational problems. By far, the most popular one has been the use of agents in an artificially constructed economy, designed to act in a specific manner. In economic theory, the term *agent* is used to refer to an actor that makes decisions, within the constraints of a particular model[66]. Specifically, in computer science, the term *agent* is used in the context of multi-agent systems (MAS) that have been used to implement market-based solutions. The creation of a multi-agent based system is appealing, since it allows researchers to actually simulate a marketplace and not just use it as a metaphor. Part of its appeal lies in the fact, that it gives researchers a tool to conceptualize and reason about multi-agent behaviour, since they can now view parts of a distributed system as autonomous agents. Wellman [100] echoes this when he says

> In order to reasonably view a distributed system as a multiagent one, we must be able to attribute to the agents particular knowledge, preferences, and abilities, which in turn dictate their rational behavior. This rationality abstraction is shared by Artificial Intelligence and Economics and is the common element that binds the complementary disciplines in this context.

If we accept that *Market Oriented Programming* is indeed a paradigm that can be useful to study the behaviour of multiple agents in a distributed setting, then we must ask ourselves what the motivation for choosing this particular form of computation is. Specifically, what're the properties that it possesses that will render it a useful mechanism for solving our problem? To enable an objective view of the utility of the mechanism, it is imperative that we define the axes on which we shall measure it.

1. **Results:** What is the quality of the solution obtained? Market Oriented Programming, as a paradigm, has mainly been applied to problems of resource allocation and load balancing. Using it as a mechanism for adapting to QoS requirements in a scalable way is a novel application.

2. **Computation:** How computationally intensive is the mechanism? How much distribution of computation takes place?

   As a measure of the efficiency of a distributed system, we have to quantify how much of the computation is really distributed. In other words, does 20% of the system do 80% of the work? This would render the system fragile, if the 'workhorse'nodes went down. One of the properties, we're trying to achieve is robustness of the mechanism itself.

Although there have not been any formal results on the computational complexity of a MAS-based solution versus a non-MAS based solution, there are many researchers who advocate the usage of agents in precisely the kind of situation that we attempt to solve. He *et al.* advocate the usage of agents[43]:

> Specifically, we believe agents are most useful in the partnership formation, brokering, and negotiation stages because these stages all involve complex issues related to decision making, searching, and matchmaking that agents are well suited to.

4.4.1 Auction-Oriented Agent Design

Zambonelli, Jennings and Wooldridge[111] propose the Gaia methodology for the analysis and design of multi-agent systems. The Gaia methodology views a MAS as an abstraction of an organization, that in the course of its functioning, solves the problem that the MAS was intended to solve. Interactions amongst the agents of the MAS need a driver, and auctions have been used as the primary driver in many approaches. There are many different types of auctions based on various parameters that can be modified. Indeed *Wurman et al.* [106] identify around 25 million auction types!

This thesis does not attempt to provide an overview of all of them (see [60] for a comprehensive review). The most commonly known ones are the *English auction*, the *Dutch auction*, *First-Price Sealed Bid auction*, *Vickrey auction* (second-price, sealed-bid) and the *Double auction*.

English Auction An *English auction* is a single-sided auction, *i.e.,* there is one seller and multiple buyers. The buyers indicate their bids and the auctioneer accepts only ascending bids, *i.e.*, every new bid must be greater than the previous one. As the bid values rise, buyers drop out when it exceeds their private valuation function or budget. The auction comes to an end, when there is only one buyer left. The transaction price is the value of the last bid.

Dutch Auction A *Dutch auction* is also a single-sided auction; there is one seller and multiple buyers. However, in this case, the auctioneer starts with a high price and continuously lowers it, with every time tick. The first buyer to accept the valuation wins the auction. The transaction price is the highest price at which a buyer stops the auctioneer.

First-Price Sealed Auction A *First-Price Sealed Bid auction* is an auction mechanism where participants involved in the auction cannot observe the bidding behaviour of the others. Each participant makes a bid, that is sealed and submitted to the auctioneer. The auctioneer collects all the bids, and opens all the bids one by one. The bid containing the highest valuation wins.

Vickrey Auction A *Vickrey auction* is a variation on the above theme. Again, none of the auction participants are able to observe the others' bidding behaviour and submit sealed bids. Also, the winner is still the bidder that submits the highest bid. However, the winner does not pay the amount that she submitted in the bid. Rather, she pays the amount bid by the second-highest bidder. A Vickrey auction has the property that *truth-telling* becomes the dominant strategy. In other words, the most rational course of action for any auction participant is to bid whatever she thinks the correct value of the good is. Although, this property is extremely interesting, implementing a Vickrey auction is a computationally intensive process.

Double Auction A *Double auction* (DA) is a two-sided auction mechanism where, (unlike the mechanisms discussed before) both buyers and sellers place bids. In auction terminology, the buyer's bid is called a *bid* and the seller's bid is called an *ask*. The *bids* and *asks* are stored in an orderbook. The market then matches the bids and asks to create potential transactions. A major aspect of differentiation amongst various types of double auctions is the time-period of the auction. In a discrete-time auction, all traders move in a single step from initial bid and ask, to the final allocation. However, in a continuous-time auction, bids and asks are entered into the market continuously. Transactions also, therefore, take place continuously.

A DA is very popular for its price efficiency. The continuous-time variant, called *Continuous Double Auction* (CDA), is used in stock-markets and commodity exchanges around the world[57]. 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. Thus, high bids (a bid where the buyer is willing to pay a high price) and a low ask (where the seller is willing to accept a low price) are favoured, in the mechanism. Also note, that at any point, the only bids and asks present in the system are ones that do not have any match. For e.g., if a bid enters the mechanism at time *t*, it is evaluated against all the asks present at that time. If no match is found, the bid can subsequently be matched only against new asks that enter. If no new asks enter, that bid will never be matched. The situation is symmetric with asks. An ask is evaluated when it enters the system and if a matching bid is found, it is immediately paired off for a transaction. Once this is done and no pairing has occurred, then it can only match a new bid. Thus, the only bids and asks remaining on the orderbook (after a round of matching), are ones that haven't been matched. To prevent stale bids and asks from remaining on the orderbook, most market institutions clear all un-matched bids and asks after a set period of time. A CDA is known to be surprisingly, highly allocatively efficient [38], *i.e.*, it achieves a very high percentage of all the possible trades, between buyers and sellers. The most important property of the work referred to above, 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 an assurance about the lower bound of efficiency of the mechanism. There has been a plethora of work done on the structural properties of CDAs[97, 73], strategies [46], algorithms [6, 84, 85], and agent-implementations [91, 80]

Eymann et al. [32] demur on the usage of auctions as an allocation mechanism. They argue that most auctions detract from the principal advantage of a market-based system, *i.e.*, robustness. The presence of the auctioneer results in the mechanism becoming a centralized one, and thus the failure of the auctioneer would result in a failure of the market, as a whole. This is a valid criticism, but it can be easily addressed by the use of multiple auctioneers. A market that contains multiple auctioneers for each good, with BuyerAgents selecting amongst these auctioneers, would result in a decentralized system.

4.4.2 Agent-Based Computational Economics

A key aspect of the methods discussed in the previous section, is that all of the methods indulge in a top-down approach to building the market. That is, structural properties are specified *a priori* in the form of *bidding rules, clearing prices, negotiation protocols*, etc. However, one of the reasons that Market-Based methods are used is because modelling these agents and their interactions, in the form of a game is highly complex. The feedback loops and interaction effects are difficult to analyze using game-theoretic methods. In such a situation, developing a top-down mechanism is not necessarily the *only way to fly*. In the words of Tesfatsion [92], "agents in these models have had little room to breathe".
In contrast, the field of *agent-based computational economics* (ACE) attempts to study economies through evolving systems of autonomous, interacting agents. The main thrust of ACE has been to study agents co-evolve their strategies for interaction, negotiation, trading, etc. in a controlled environment. Keeping track of the evolution of agents, allows researchers to ascertain the effect of microscopic changes on the macroscopic behaviour, and the constraints and feedback from the macroscopic environment on the microscopic structures. Thus, the aim of

ACE is two-fold: *descriptive* and *normative*. That is, ACE aims to both, describe the effects of certain starting parameters on the resultant macroscopic behaviour, and prescribe necessary conditions for the rise of desirable behaviour.

Robert Marks was one of the earliest researchers in this field [92], and his experiments on oligopolistic markets [65] threw up a then-surprising result for economists, that of the evolution of bottom-up cooperation to result in globally optimal joint allocations. However, a more interesting result was that chance was a big determiner of final outcomes, as did "behavioural quirks that individual firms had evolved in response to their interaction histories"[92]. This implied that an optimal pricing strategy created for one kind of game, would not necessarily prove to be good in a different game. This result proved that analyzing market-economies using game-theoretic approaches was not only difficult, but also potentially futile. In this scenario, Nicolaisen et al. showed that creating a definite structure for agents to trade in, in the form of a double auction, rendered attempts by agents to exercise strategic market power ineffective[70]. Due to the symmetry of the double auction design (both buyers and sellers can specify prices), market-efficiency was really high.

When combined with Gode and Sunder's [38] study of *Zero-Intelligence* agents, it is fairly obvious that a *continuous double auction* market mechanism is the most structurally suited to our task of getting multiple-agents to choose services. In the next chapter, we explain our market-mechanism in detail.

4.5 Conclusion

In this chapter, we took a short tour of decentralized self-adaptive communication patterns, motivated the selection of a market-based mechanism, and took a deeper look at auctions. Amongst auctions, we took a look at the popular structural variants in auction-design, highlighted the properties that we wanted, and use results from auction literature and ACE to support our idea of using agents in double-auctions for service selection. In the next chapter,

we shall take a deeper look at the auction mechanism we propose, its structural format, and agent decision-making strategies.

CHAPTER 5

Mechanism Design

The whole is more than the sum of its parts

Metaphysics, *Aristotle*

5.1 Introduction

Mechanism-design is a sub-field of micro-economics and game theory, that considers how to design systems and interaction rules for solving problems that involve multiple self-interested agents[79]. The desiderata from our mechanism's design is the following:

1. The mechanism should be highly allocatively efficient
2. The mechanism should be decentralized, so that there is no dependence on a single entity
3. The mechanism should be scalable to hundreds of applications, simultaneously adapting

We design a marketplace that allows individual applications to select web-services, in a decentralized manner. This is important from the view of robustness, and practicality. The cloud is designed to be an ultra-large collection of applications, web-services and raw computing power. Given this large scale, any solution that is implemented, should ideally not rely on global knowledge, or centralized coordination.
We view an application as a composition of several web-services. A web-service corresponds to a specific piece of functionality, along with associated QoS levels. Varying levels of QoS require different implementations and depending on the complexity of the implementations, will likely be either scarce or common. This leads to a differentiation in pricing based on QoS levels. Thus, in a market for clustering web-services (for example), each seller is assumed provide a web-service that performs clustering, but offers varying performance and dependability levels. Each of these web-services contributes towards the total QoS exhibited by the application. We now define terms that will be used throughout the thesis.

AbstractService (AS): The functional specification of a certain task. This is also sometimes referred to simply, as a *task* or *service class* in a Workflow. An architect composes several AbstractServices into a Workflow, to create an application.

ConcreteService (COS): An implementation of an AbstractService. Each ConcreteService has a QoS that it advertises through its SLA.

CandidateService (CAS): A ConcreteService provided by a service-provider, that is present in a particular market, and meets the QoS requirements of a certain BuyerAgent. The service-provider may be the cloud itself, or a third-party. The same ConcreteService may therefore, be a CandidateService for one BuyerAgent, and not be a CandidateService for another.

Market (M): A virtual meeting place for BuyerAgents and SellerAgents. Each Market is homogeneous in the sense that, all ConcreteServices in that market, are implementations of the same AbstractService.

BuyerAgent (BA): A trading agent that Bids for, and buys a CandidateService. Typically, a BuyerAgent is responsible for one AbstractService.

SellerAgent (SA): A trading agent that sells a CandidateService. Again, typically a SellerAgent is responsible for only one AbstractService. The SellerAgent adjusts the price at which it is willing to sell the CandidateService, based on demand and supply.

MarketAgent (MA): A trading agent that implements trading rounds for a Market. It accepts Bids, and asks from BuyerAgents, and SellerAgents. It performs matching of Bids and asks.

ApplicationAgent (AA): An agent responsible for ensuring that an application meets its QoS requirements. It is responsible for distributing the budget, and initiating and stopping adaptation by the BuyerAgents.

Workflow (W): The set of AbstractServices that compose an application.

5.2 Agents in the System

5.2.1 BuyerAgent

The BuyerAgent (BA) is a trading agent that buys a CandidateService for an ApplicationAgent(AA), i.e., it trades in a Market (M_x) for a specific AbstractService (AS_x). Each web-service (COS_x), available in M_x, exhibits the same QoS ($\omega \in QoS$). The only differentiating factor is the degree to which that is exhibited. Hence, if an application has K QoS that it is concerned about, then the QoS that it gets for each of the AS_x that it buys is:

$$\Omega^{\text{AS}_x} = \langle \omega_1^{\text{AS}_x}, \omega_2^{\text{AS}_x}, \omega_3^{\text{AS}_x}, \ldots, \omega_K^{\text{AS}_x} \rangle \tag{5.1}$$

The amount that the BuyerAgent is prepared to pay is called the *bid price*($\text{BA}_{bidprice}$) and necessarily $\text{BA}_{bidprice} \leq \text{BA}_{budget}$, where BA_{budget} is the budget available with the BuyerAgent[1]. The combination of Ω demanded and the *bid price* is called a *Bid*. A BuyerAgent's *Bid* is denoted as BA_{BID}. A BA_x makes several *Bids* to explore the QoS–and–cost space (as detailed in section 5.4.2).

5.2.2 ApplicationAgent

The Application is described as a composition of different AS_x. At the agent-level, the ApplicationAgent communicates with each of its BA, and communicates the local QoS constraints and a budget (B_{f_x}). The buyer then starts trading. The adaptation occurs through the revision of Bids, that happen after each round of trading. After each round of trading, depending on the Ω obtained by the buyer, the Application has procured a total QoS that is given by applying *Stochastic Workflow Reduction* (detailed in section 5.4.1). The relationship between the ApplicationAgent and the BuyerAgents is shown in Figure 5.1, for a random application structure consisting of eight services.

[1] This is different from typical CDA literature, where BA_p is used to denote the *bid price*

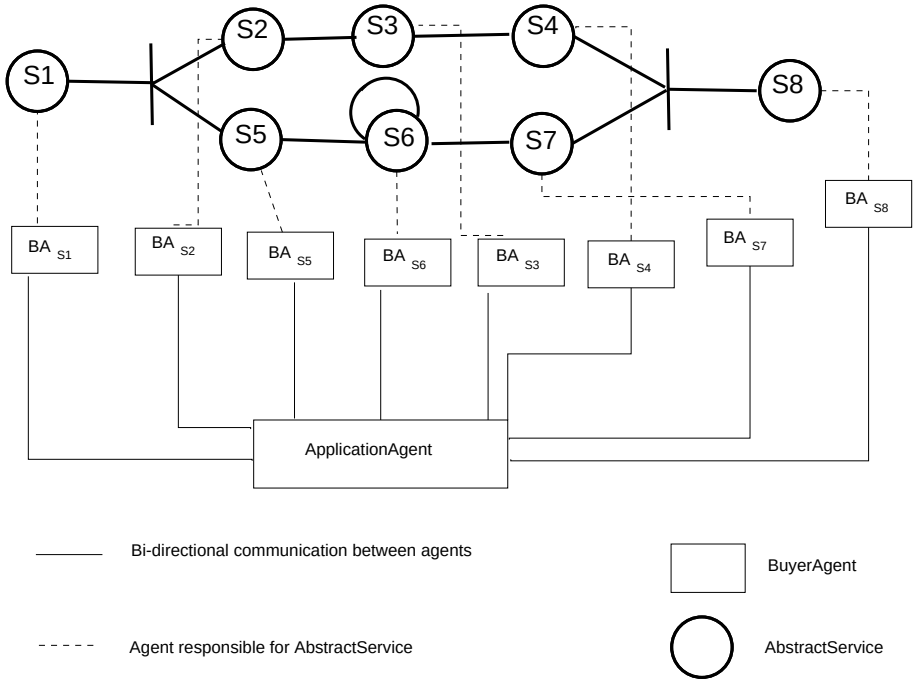


Figure 5.1: Relationship between ApplicationAgent and corresponding BuyerAgents

5.2.3 SellerAgent

Each SellerAgent is a trading agent, selling a particular ConcreteService (COS_x) that exhibits the QoS mentioned in (5.1). The degree to which each QoS attribute(ω) is exhibited in each COS_x being sold, is dependent on the technological and economic cost of providing it. Hence, if the cost of providing AS_x with $\Omega = \langle 0.5, 0.6 \rangle$ is low, then there will be many sellers providing AS_x with a low *Ask price*. Conversely, if the cost of providing AS_x with $\Omega = \langle 0.5, 0.6 \rangle$ is high, then the *Ask price* will be high. An individual seller's *Ask price* can be higher or lower based on other factors like number of sellers in the market, the selling strategy etc., but for the purpose of this experiment we consider only a simple direct relationship between cost and *Ask price*, where the *Ask price* is always *greater than or equal to* the cost. The combination of Ω and *Ask price* is called the *Ask*. Every service is sold for a pre-specified n calls, i.e., a purchase of a service entitles the buyer to make n (market specified) calls to that service. Provenance regarding the actual usage of the service, can be easily established through the use of authentication keys, or other mechanisms.

5.2.4 MarketAgent

A Market (M_x) is a set of BA and SA, all interested in the same functionality AS_x. The factors differentiating the traders are:

- Ω**:** The combination of $\langle \omega_1, \omega_2, \ldots \omega k \rangle$
- ***Price:*** Refers to the *Bid price* and *Ask price*. The BA will not pay more than their respective *bid price* and the SA will not accept a transaction lower than their respective *Ask price*.

An M_x is responsible for accepting *Bids* and *Asks* from BA_x and SA_x. It is also responsible for conduction the auction, and creating provisional transactions (see section 5.3). When a BA_x^i (say) accepts a transaction, the M_x is responsible for setting up a final transaction, and deleting all the other *Bids* that BA_x^i had made.

5.3 Structure of the Auction

The mechanism of finding a matching buyer-and-seller is the *continuous double auction*(CDA). A CDA works by accepting offers from both buyers and sellers. It maintains an orderbook containing both, the *Bids* from the buyers and the *Asks* from the sellers. The Bids are held in descending order of price, while the asks are held in ascending order, *i.e.*, buyers willing to pay a high price and sellers willing to accept a lower price are more likely to trade. When a new *Bid* comes in, the offer is evaluated against the existing *Asks* in the book and a transaction is conducted when the price demanded by the *Ask* is *lower than or equal to* the price the *Bid* is willing to pay **and** all the QoS attributes in Ω of the Ask are *greater than or equal to* all the QoS attributes in the Ω of the *Bid*. After a transaction, the corresponding *Bid* and *Ask* are cleared from the orderbook. Table 5.1 shows the state of the orderbook at some time, t_0. Maximizing the number of transactions would lead to 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 needed increases exponentially.

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 5.1: Orderbook at time t_0

With a CDA, the set of transactions is: $[B1-S1, B2-S3]$. This is so because a CDA evaluates Bids and asks, as they come in 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 (Table 5.2) 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 [38]. 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.

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 5.2: Orderbook at time t_1

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 made from a buyer or seller, 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[97]. We modify this rule to accept multiple Bids from BuyerAgents. This modification allows the BuyerAgents to explore the QoS-cost search space in a more efficient manner (see section 5.4.2).

2. **Information Revelation Rule:** This refers to the market information that is available to buyers and sellers. 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 one, is also called a Clearing House. In our mechanism, as soon as an Ask meets all the QoS constraints specified in a Bid, and the bid_price is greater than or equal to the ask_price, a provisional transaction is identified. The transacting parties are now given the option of accepting or rejecting the transaction. Again, this is a departure from typical CDAs, but essential to our mechanism (see 5.4.4.1). If accepted, the shouts of the transacting agents 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 BuyerAgent or the SellerAgent. We use $k = 0.5$ (*i.e.*, $0.5 * bid\_price + (1 - 0.5) * Ask\_price$) as the transaction price, as this does not favour either the BuyerAgent or the SellerAgent.

A CDA can be varied substantially by changing the strategies, for each of the rules above. In addition, the market may impose additional costs such as registration charge, transaction charge, additional information charge, etc. Since these costs do not structurally impact the CDA, for the purpose of our analysis, we assume that these costs are zero.

5.3.1 Modifications to the CDA

We have made two significant modifications to the typical CDA, in the *Shout Accepting Rule* and the *Clearing Rule*. Our *Shout Accepting Rule* accepts **multiple Bids** from a BuyerAgent. This modification allows the BuyerAgent to explore the space of QoS attributes and cost combinations, much more efficiently than a single-Bid-at-a-time approach. We follow a systematic method to create multiple Bids, given a BuyerAgent's local QoS constraints and budget (see 5.4.2).

Also, in a traditional CDA, a buyer and seller whose shouts match, are immediately paired-off for a transaction. However, we modify this rule to create **provisional transactions**. Once paired off for a transaction, a buyer and seller have the **option** to transact. Since the BuyerAgent makes multiple Bids, it is possible that it gets multiple possible transactions. The MarketAgent generates provisional transactions, and informs the BuyerAgent. At this point, the BuyerAgent has to rank the various Asks in the provisional transactions, and chose the best one. Systematically, choosing the best Ask in light of various QoS attributes is a difficult task, and we use a multi-criteria decision making approach (see 5.4.4.1).

5.4 Calculation, Communication, Decision-Making

5.4.1 QoS Calculation

One of the critical aspects of using a multi-agent system is the distribution of constraints and consistency checking. In the domain of service-based systems, applications usually enjoin constraints like `cost of all services` $< C$ or `all services must use encryption`.

Constraints: There are three types of 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 it as a hard constraint, which means that a CandidateService that does not support SSL is not acceptable, 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, might 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* [22]. Using this method, the Workflow is reduced using a series of reduction steps. Each time a reduction rule is applied, the structure of the network 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 Workflow. The QoS metrics aggregated for this task represents the QoS metrics corresponding to the Workflow. In Figure 5.2, we see that a set of parallel tasks, $t_1, t_2, t_3 ... t_n$, 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:

$$Cost(t_N) = \sum_{1 \leq i \leq n} Cost(t_i) \tag{5.2}$$

$$Reliability(t_N) = \prod_{1 \leq i \leq n} Reliability(t_i) \tag{5.3}$$

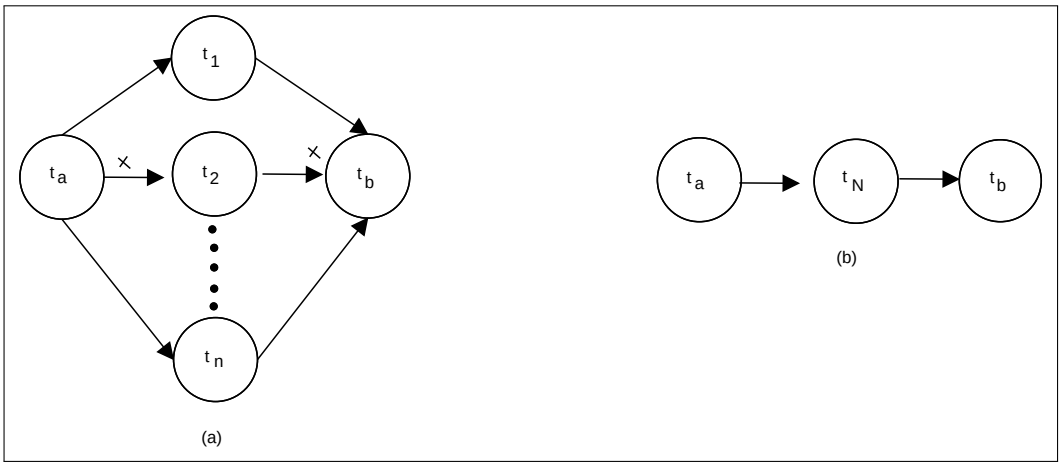


Figure 5.2: Reduction of parallel tasks(reproduced from [23]). For additive QoS, the attributes are summed up, while for multiplicative QoS, the attributes are multiplied together

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 [23] are used for other types of tasks, such as sequential tasks, loop tasks, conditional tasks, etc.

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 global constraints that the application imposes, 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, the last two types of constraints (boolean and categoric) do not need

to be decomposed at all. They are given to all agents, as is. For example, if `all services must use encryption` is a constraint, then every agent can be given the constraint, without any processing. On the other hand, constraints like `cost of all services` $< C$ need to be decomposed on a per-Agent basis. Numeric constraints are decomposed, as shown in Algorithm 1

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 $a^{\omega\_m}_{low}, a^{\omega\_m}_{median}, a^{\omega\_m}_{high}$
    end
  end
  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**
        $a^{\omega}_{bid} \propto SWR^{\omega}$
      end
    end
    Choose different m
    if $allMarketsExhausted$ **then**
      TerminateWithFailure
    end
  end
end

Algorithm 1: Decomposition of Numeric Constraints

Decomposition of Numeric Constraints Numeric constraints cannot be decomposed by simply dividing the numeric value, across all the agents. That is, if an application has a cost constraint of C, and three BuyerAgents, it cannot simply divide $C/3$ and allocate it to each agent. This is so, because depending the kind of service, each service could have vastly different costs. Also, depending on the structure of the application, a small change in QoS

of one service, could drastically change the QoS exhibited by the application (as explained in 5.4.1). Hence, the ApplicationAgent performs the following steps:

1. It asks each BuyerAgent to register in the markets available for that service.
2. Next, each BuyerAgent queries its markets for the last k transactions.
3. From these transactions, each BuyerAgent calculates the lowest, median, and highest value of each Numeric QoS, it is concerned about.
4. Each BuyerAgent then communicates these values to the ApplicationAgent.
5. The ApplicationAgent applies SWR to each value, and checks for constraint violation.
6. The ApplicationAgent allocates the first combination of numeric QoS values that meets all its constraints, to all its BuyerAgents.
7. If no combination exists, the ApplicationAgent instructs the BuyerAgents to try a different market.
8. If all markets are exhausted, then the ApplicationAgent terminates the process, with a failure signal.

5.4.2 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, that not only meet the application's QoS requirements, but also explore the space of possible matches. We therefore, recap the relevant parts that go into a Bid in table 5.3.

The most important part of deciding on a value for a QoS attribute, in a Bid 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

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 5.3: Elements in a Bid

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. That is, the BuyerAgent makes one Bid at its target value, one at the median value obtained from the market, and the last one at the mid-point between the target and the median value.
Thus, the total number of Bids that a buyer generates is given by:

$$Num(buyer_{Bids}) = 2 * Num(BooleanQoS) * Num(AcceptableAlternatives) * 3 * Num(NumericQoS) \tag{5.4}$$

Thus, if one of the BuyerAgents of BizInt (our fictional company) has the following QoS and preferences:

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)80	(cannot exceed)100	N.A.

Table 5.4: QoS preferences for a BuyerAgent

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

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

5.4.3 Bids Generated for Sample Scenario

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

Attribute	Value
Bid-Price	80
SSL	No
Framerate	24fps
Latency	99

Attribute	Value
Bid-Price	80
SSL	Yes
Framerate	32fps
Latency	99

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

Attribute	Value
Bid-Price	80
SSL	Yes
Framerate	24fps
Latency	97

Attribute	Value
Bid-Price	80
SSL	No
Framerate	24fps
Latency	97

Attribute	Value
Bid-Price	80
SSL	Yes
Framerate	32fps
Latency	97

Attribute	Value
Bid-Price	80
SSL	No
Framerate	32fps
Latency	97

Attribute	Value
Bid-Price	80
SSL	Yes
Framerate	24fps
Latency	95

Attribute	Value
Bid-Price	80
SSL	No
Framerate	24fps
Latency	95

Attribute	Value
Bid-Price	80
SSL	Yes
Framerate	32fps
Latency	95

Attribute	Value
Bid-Price	80
SSL	No
Framerate	32fps
Latency	95

Table 5.5: All the generated Bids given Equation 5.5 and the QoS preferences in Table 5.4

5.4.4 Decentralized Decision-Making Using Ask-Selection

5.4.4.1 Ask Selection

In any adaptation scenario, deciding quickly on which alternative to take, is a critical task. After generating multiple possible alternatives, and once the obviously incorrect choices are eliminated, the task of the decision-maker becomes subtle. There are three possible scenarios:

1. **No feasible alternatives remain**: From a decision-making perspective, this is an easy task. New alternatives have to be generated, else adaptation will not occur.

2. **One feasible alternative remains**: Again, decision-making is easy. Since there is only one feasible alternative, it must be chosen.

3. **Multiple feasible alternatives exist**: In this case, the obvious (and simplest) solution is to choose the best out of the feasible alternatives. However, choosing the best is a NP-hard problem (as discussed in section 3.2). If the search-space is high (due to the multiple dimensions involved in service selection), the time taken to evaluate all the alternatives and choose the optimal might not feasible. An alternative to choosing the optimal, is to use heuristics. Heuristics are *rules-of-thumb*, born out of expertise, that help in decision-making. However, heuristics are not guaranteed to always choose the best alternative either. Between the strict rationality of optimality, and the human-oriented heuristics method, we choose a third alternative, a systematic ranking method for multi-criteria decision making.

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 that it could get, for its stated budget. Now, there might be multiple Asks in the market that match these multiple Bids. The market mechanism now moves on to its second-stage, where the BuyerAgent has to select amongst the possible transactions. Given that the mechanism for Bid generation precludes Bids that will violate hard constraints, placed by the ApplicationAgent, it is safe to assume that none of the possible transactions 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[12], to select between multiple Asks that are possible transactions. More details about the PROMETHEE method can be found in appendix A.1

5.4.4.2 Calculation of Preference

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. Depending on the number of QoS attributes, the preference of CandidateService A over CandidateService B is calculated by aggregating the preferences over each QoS attribute (and its weightage). The weightage (w) for each QoS attribute needs to be set by the application architect, since this is domain-specific, and needs human expertise. The aggregate preference value of a CandidateService with n QoS attributes, is calculated as:

$$\pi(A,B) = \sum_{i=1}^{n} P_i(A,B)\, w_i \tag{5.5}$$

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 flows and the negative outranking flows:

$$\phi^{+}(A) = \sum_{x \in TS} \pi(A,x) \tag{5.6}$$

$$\phi^{-}(A) = \sum_{x \in TS} \pi(x,A) \tag{5.7}$$

Finally, the net outranking flow is created to get a complete ranking of all the CandidateSer-

vices:

$$\phi(A) = \phi^{+}(A) - \phi^{-}(A) \tag{5.8}$$

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

5.4.4.3 A Worked-out Example

Suppose that the 12 Bids generated in Table 5.5 were entered into a market, and the MarketAgent returns the following 4 Asks, as provisional transaction matches:

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 5.6: Asks returned by MarketAgent as provisional transactions

Ranking amongst multiple Asks is done on a per-QoS attribute basis. That is, each Ask is ranked on each QoS attribute. Depending on the type of QoS attribute, the criteria used from PROMETHEE is shown in Table 5.7

QoS Attribute Type	PROMETHEE Criterion Type
Boolean	Usual Criterion
Categoric	Quasi-Criterion
Numeric	Criterion with Linear Preference

Table 5.7: QoS Attribute to PROMETHEE Criterion Mapping

Table 5.7 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 it has a desired boolean value, while the other doesn't. 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 (higher if the attribute is to be maximized, lower otherwise) 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:

$$P_{ssl}(x) = \begin{cases} 0 & \text{if } x = (ssl = no), \\ 1 & \text{if } x = (ssl = yes) \end{cases} \tag{5.9}$$

$$P_{framerate}(x) = \begin{cases} \frac{1}{32}x & \text{if } x < 32fps, \\ 1 & \text{if } x \geq 32fps \end{cases} \tag{5.10}$$

$$P_{latency}(x) = \begin{cases} 0 & \text{if } x > 103ms, \\ \frac{1}{2} & \text{if } 103 > x > 95ms, \\ 1 & \text{if } 95 \geq x \end{cases} \tag{5.11}$$

Based on the preference equations 5.9, 5.10 and 5.11, we can calculate the relative values of the 4 CandidateServices, as given in Table 5.8 Once we have a relative per-attribute value, we

Table 5.8: 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	—

can calculate the outranking values based on Equations 5.6, 5.7 and 5.8, as shown in Table 5.9

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

Table 5.9: Calculation of outranking values

It is clear from Table 5.9 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.

5.5 Use of MDA for QoS adaptation

We design an economy which is a hybrid of ideas from the fields of traditional MBC and ACE. Although both approaches recommend CDA as an effective mechanism, we are mindful of

Eymann's criticism [32]. Therefore, we create *multiple double auctions* (MDA) for individual applications to select web-services. This is important from the view of robustness and even practicality. The cloud is designed to be an ultra-large collection of applications, web-services and raw computing power. Given this large scale, any solution that is implemented, must not rely on a single market or coordination. We view an application as a directed graph of abstract web-services. Thus, an application can select from multiple candidate services for each AbstractService in its Workflow. Agents for the application can trade in multiple markets for selecting an appropriate CandidateService. An application thus creates at least one agent per AbstractService in its Workflow. Each agent now registers with multiple markets for trading, to procure the CandidateService corresponding to its AbstractService. The total QoS exhibited by the application, is a function of the individual QoS exhibited by each of the services. Each service is assumed to be traded in its own market. Thus, in a market for clustering web-services, each seller will provide a web-service that performs clustering, but offering 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.

We populate several markets with BuyerAgents and SellerAgents. As mentioned before, each service (AS_x) is sold in a market that is specific to AS_x. All the buying agents and selling agents in this market, trade web-services that deliver AS_x. We assume that there exists a market M_x for every AS_x, that is needed.

5.5.1 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

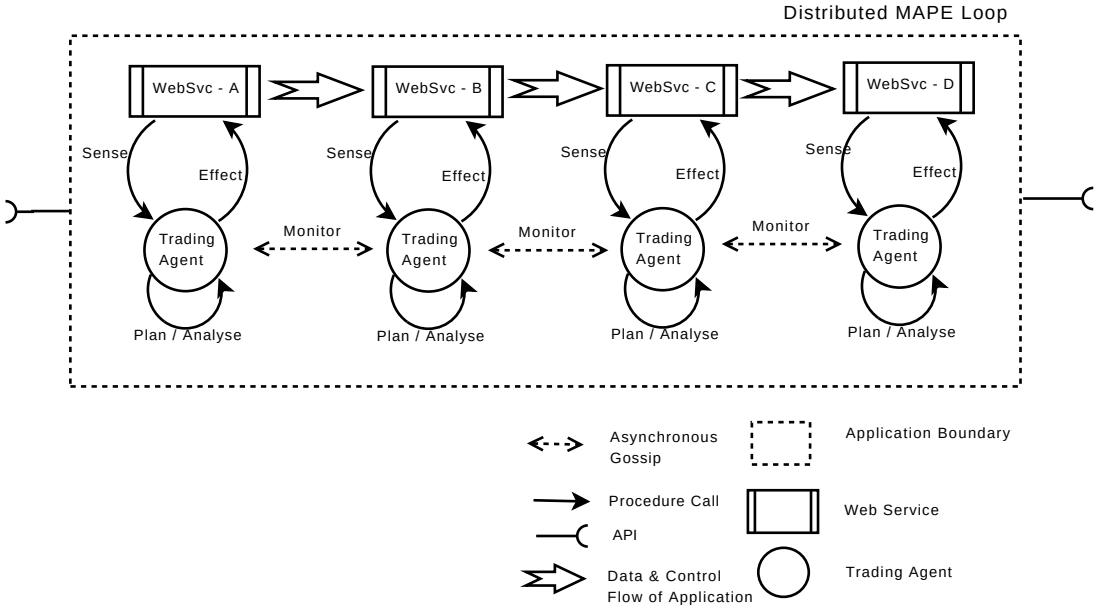


Figure 5.3: Self-Adapting Application with a decentralized trading agents

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. The ApplicationAgent merely has to sum up the cost of all the BuyerAgents to ensure that the total cost is within budget. However, in the worst case scenario, all the QoS attributes that the ApplicationAgent is interested in, 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 5.4, we show the various steps in the CDA as activities carried out by the principal agents.

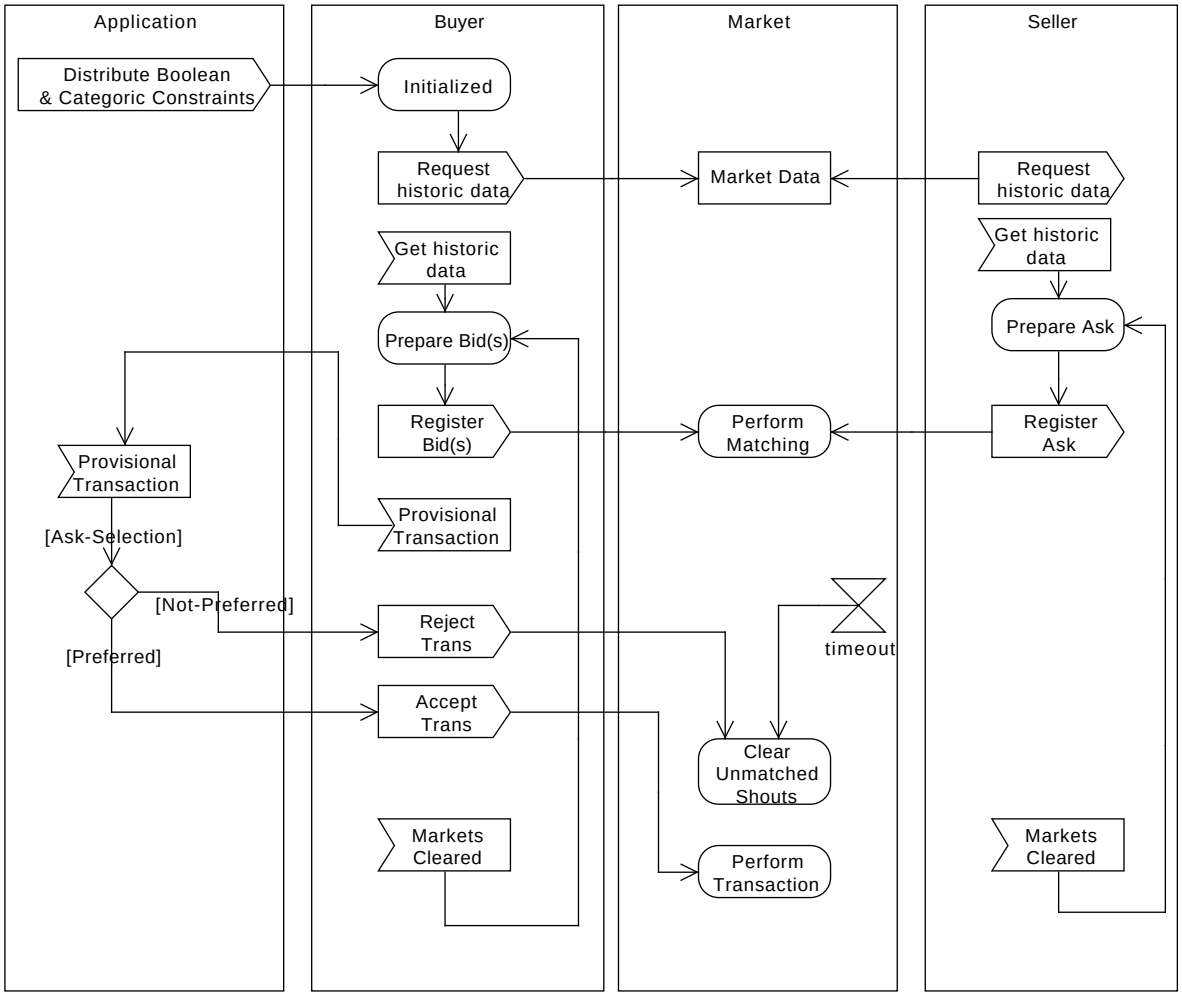


Figure 5.4: Activity diagram for principal agents

5.6 Description of ApplicationAgent and BuyerAgents' Lifecycle

We now describe the algorithms used by the ApplicationAgent and BuyerAgents, in the course of adaptation. There are two distinct phases in the agents' lifecycle :

- **Initialization Phase:** In this phase, the ApplicationAgent is responsible for achieving the following goals:

  1. Distribute total budget amongst BuyerAgents
  2. Distribute end-to-end constraints amongst BuyerAgents

The BuyerAgents are responsible for achieving the following goals:

1. Register with a market that deals with their specific web-service
2. Acquire historical data regarding services sold in the market. Specifically, the BuyerAgent must get data regarding the lowest transaction price, the highest transaction price, the median transaction price and their corresponding QoS values.
3. Communicate the median transaction price and corresponding QoS values to the ApplicationAgent

- **Adaptation Phase:** In this phase, the ApplicationAgent is responsible for achieving the following goals:

1. Communicate to the BuyerAgents, in the following cases:
   (a) The total budget available changes
   (b) The target QoS changes

The BuyerAgents are responsible for achieving the following goals:

1. Generate multiple Bids for registering in markets
2. Based on possible transaction matches, generate ranking amongst matched Asks and conclude transaction
3. Communicate with ApplicationAgent about concluded transaction
4. Register ConcreteService with QoS Monitoring Engine
5. Monitor communication with ApplicationAgent for change in QoS
6. Monitor communication with QoS Monitoring Engine

5.6.1 Adapting Bid and Ask Prices

Bids and Asks form the core of the adaptive process of our mechanism. After an initial start, the BuyerAgents perform the equivalent of a local search, by means of making multiple

Bids. Multiple values for QoS attributes in Bids can be generated by the process described in section 5.4.2. The *bid_price*, on the other hand, requires a different treatment. We take our starting price to be the median price in the market. However, since the median price reflects past data, it will not necessarily result in a trade. We need a different algorithm to adjust the *bid_price*. We use the least mean squares method (also known as *Widrow-Hoff learning rule*) [102] to adapt the price. ZIP [26] implements the Widrow-Hoff learning rule to perform a gradient descent of the current price (in the Ask or Bid) to the last transaction price, in the market. This enables agents (both BuyerAgent and SellerAgent) to move their shout prices, in line with the market trend. If a SellerAgent is able to trade, it increases the price it charges, in the next Ask, using ZIP. If the seller is unable to find buyers at a higher price, it drops back to its older price.

The buyers are constrained by their budget, for the prices that they are willing to pay. The sellers are also constrained by their minimum support prices, below which they will not trade.

5.6.2 Stopping Criteria

Like most research on dynamic service composition, we evaluate the goodness of a particular set of services, by means of a utility function. Given a set of services, we use their associated QoS to calculate application's end-to-end QoS. 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* (IUL). The achieved QoS is fed through the same process, to attain the *achieved utility level* (AUL). Depending on the domain, the application may decide to tolerate a small (ϵ) level of deviation from the IUL. This ϵ creates a *satisfactory zone*. Trading by the agents (and hence adaptation) stops when the application's AUL is within the satisfactory zone. The level of tolerance can be made arbitrarily small, or even zero. The ϵ can be different for different applications, and indeed, has no restriction whatsoever. Clearly, if it is set injudiciously, the application may either stop adapting too quickly or may never stop adapting. Thus, it becomes an important parameter to tune.

5.6.3 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 event**. There are two kinds of adaptation events:

1. Internal Event: When a QoS violation is detected at any of the CandidateService, the ApplicationAgent informs the BuyerAgent that it needs to re-start 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. External Event: If the application's end-to-end QoS target, or available budget changes, or available service calls finish, the ApplicationAgent informs all of its BuyerAgents, and restarts the initialization phase of the BuyerAgents' lifecycle 5.6. Else, the BuyerAgents stay in the adaptation phase, until the stopping criteria are met.

Both these events occur at different time scales. The internal check for QoS violation is a continuous check and concerns itself with performance of the web-service across relatively shorter periods of time. The external event, 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.

5.7 Description of the SellerAgent's Lifecycle

The SellerAgent represents a ConcreteService, that is being offered. Hence, it cannot change the QoS that it advertises in its Ask. However, it changes the price at which it is willing to trade, to ensure that it gets some price for its ConcreteService. The SellerAgent, in order to achieve profit maximization, will continue to sell as long as the marginal price it gets, is greater than the marginal cost of providing the service([64], pages 245-247). This is concretized for the

SellerAgent in the constraint that its *Ask price* is always greater than or equal to its *cost*. There are two phases in the lifecycle of a SellerAgent:

- **Initialization Phase**: In this phase, the SellerAgent is responsible for achieving the following goals:

  1. Find the market that it can sell the ConcreteService in
  2. Register in all the markets found
  3. Request historical data about prices
  4. Prepare Ask
  5. Register Ask

- **Adaptation Phase**: In this phase, the SellerAgent is responsible for achieving the following goals:

  1. If traded, withdraw from market(s)
  2. Else, adapt price using ZIP

5.8 QoS Monitoring Engine

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 specific monitoring mechanism. This could be market-specific or domain specific. The monitoring mechanism needs to be neutral, since it will be used by both parties: BuyerAgent and SellerAgent. The BuyerAgent needs to know whether the SellerAgent's service is providing the QoS that it promised. The SellerAgent needs to know that the BuyerAgent's application is not abusing the service. Zeng [112], and Michlmayer [68] 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.

5.9 Conclusion

This chapter introduced several decentralized multi-agent based methods for self-adaptation. We provided a rationale for picking Market-based control and then introduced our MDA-based mechanism in detail. We outlined the principal algorithms used for initialization of agents, revising of Bids, and communication of attained QoS. In the next chapter, we shall discuss the requirements, and design of the *multi-agent system* that we create, design alternatives and decisions.

CHAPTER 6

Requirements and Design of Clobmas

> The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination.
>
> Frederick Brooks Jr.

6.1 Introduction

In the previous chapter, we explicated our market-based mechanism, and the goals of each agent. Now we proceed to establish the engineering foundations of the same. Designing a multi-agent system is a complex task, with decisions to be taken regarding the computational ability of each agent, the hierarchy of control (if any), the mechanism of communication, the frequency of communication, etc. Each of these decisions has an impact on the properties exhibited by the system, in terms of its flexibility, scalability, etc. These decisions once made, are difficult to change, and therefore demand careful consideration before starting to build a system. Therefore, in this chapter, we perform the following steps:

1. We state the quality goals that the mechanism should have,
2. Identify the important variables in the system that are likely to change,
3. Isolate target variables that are relevant to the quality goals, and
4. Isolate the operational variables, which drive the quality goals and specify their operational ranges

6.2 Requirements

Goal-Oriented Requirements Engineering (GORE) aims to systematically proceed from the elicitation of the goals (the **why** issues), the operationalization of these goals into service specifications and constraints, (the **what** issues) to the assignment of responsibilities for the resulting requirements to agents (the **who** issues)[94]. This rigourous approach to creating goals, specifying constraints, and assigning responsibility aims to provide software architects with specific guidance, on how to design a particular system. Starting from the goals that the system must achieve, GORE processes aim to create a traceable path between the needs of the user (explicit and implicit) and the design elements that realize these needs. In our case, the functional goals of the system were derived in the previous chapter. Our non-functional

goal viz, scalability, has not been elucidated yet. In the following sections, we describe what we mean by scalability, and how we plan to measure it.

6.2.1 Goal Oriented Scalability Characterization

Scalability is one of the most important non-functional goals of a software system, but it is often mis-understood. Scalability is not just the ability of a system to meet its performance goals. Rather, depending on the system's goals, it is the ability to maintain quality levels that are demanded by the system stakeholders, when certain characteristics vary over expected operational ranges[31]. An underwater autonomous vehicle (UAV) must be able to navigate to its desired target area, in spite of large number of obstacles; an air traffic control system should maintain safe levels of aircraft separation, even in times of increased traffic. Both these goals differ conceptually from a web-based search engine's goal to return results within a certain specified time, but they are all scalability goals.
We used the method outlined in [31] to uncover system characteristics in our specific case, and the operational ranges that these characteristics are expected to vary over. Duboc et al [31] use concepts and terminology from the KAOS framework[94], and build on it, specifically to cover issues of scalability. More information about the KAOS framework, and the process of goal elicitation and refinement, can be found in Appendix B.1.1. Based on the method given in [31], we specified the quality goals of the system.

Quality Goals:

1. The proportion of applications that are able to adapt, should be high. Since our mechanism is stochastic, it cannot guarantee that every application will be able to meet its QoS requirements. This could be due to various reasons:

    - An application requires QoS levels that are simply not available
    - An application does not have enough budget, for its desired QoS levels

Therefore we require that, given a gaussian distribution of QoS values and appropriate budget values, at least 80% of the applications should be able to achieve their QoS requirements.

2. The amount of time taken to perform adaptation does not increase exponentially with increase in trading agents. One of the big problems with service-selection is that a deterministic selection of the optimal service, is NP-hard. Since we relax the condition of optimality and determinism, our mechanism should be able to tolerate a high number of applications, and by extension, trading agents.

Given these quality goals, it is possible to design a system that achieves these goals. However, in any real-world situation, the environment that the system operates in, will always change. To be called scalable, the system must continue to achieve the quality goals, when important characteristics of the environment or the system itself, vary.

Identification of characteristics that are expected to vary:

1. Characteristics of the application:
   (a) Number of AbstractServices per application
   (b) Number of QoS attributes per application
2. Characteristics of environment:
   (a) Number of ConcreteServices available (per AbstractService)
   (b) Number of Markets available (per AbstractService)

We use the characteristics identified, to refine the quality goals, into more concrete goals. Each goal is refined, until it can be operationalized, and is assigned to an agent.

6.2.1.1 Goal refinement for scalability

In Figure 6.1, we see a portion of the goal refinement graph. It has a higher-level goal which is concerned with achieving the maximum possible matching of ConcreteService providers to service consumers. We see it refined into `[Fast Price and QoS Matching]` and `[All applications that need services should be matched]`. These sub-goals are restricted by the presence of obstacles. Obstacles are conditions (like properties of the domain) that prevent a goal from being satisfied. An obstacle can be resolved through several alternative obstruction resolution tactics [95]. As shown in Figure 6.1, we resolve the obstacles in the following ways:

1. Refining the obstacle into sub-obstacles which are then resolved using requirements
2. Introducing expectations, which resolve the obstacle
3. Introduce a weakened goal, which avoids the condition described by the obstacle

After obstacle resolution, we see goal-weakening that results in a specific formal operational range for a particular variable. Through this process, we arrive at specific ranges that denoted the expected operational range, for that particular variable. Thus, to characterize a system as being scalable, we arrive at specific, measurable sub-goals that a particular agent is responsible for achieving. If each agent is able to achieve its allocated goal, then the system is said to be scalable. In light of this, we formulate scaling goals for Clobmas.

Scaling Goals:

1. Ratio of Time-Taken-to-Adapt to increase in ConcreteServices increases linearly or as a low-order polynomial
2. Ratio of Time-Taken-to-Adapt to increase in Markets increases linearly or as a low-order polynomial
3. Ratio of Time-Taken-to-Adapt to increase in QoS attributes increases linearly or as a low-order polynomial

4. Number of applications that have adapted in all cases shall not be lower than 70% of the market

To each of these goals, there could be a corresponding obstacle, that prevents Clobmas from reaching its goal.

Scaling Obstacles:

1. As the number of ConcreteServices increase, the Time-Taken-to-Adapt could increase exponentially

2. As the number of Markets increase, the Time-Taken-to-Adapt could increase exponentially

3. As the number of QoS attributes increase, the Time-Taken-to-Adapt could increase exponentially

4. In a certain market configuration, the number of applications that are able to adapt might be less than 70%

Given that these goals and obstacles do not prescribe any specific, measurable quantities, it is difficult to know whether Clobmas scales successfully, or not. In order to assign specific numbers to the operational variables, we make some scaling assumptions.

Scaling Assumptions: To arrive at a specific, measurable sub-goal, we elicited expert judgement. We contacted Capacitas Inc.[1], which is a capacity and performance planning consultancy. They advise their clients on the kinds of SLAs needed to establish a certain level of end-to-end QoS on the *Azure* cloud. In conversation with Mr. Danny Quilton (from Capacitas Inc.), we established the following assumptions about the range of values, taken by the operational variables:

[1] www.capacitas.co.uk

1. Number of AbstractServices per application goes from 2 through to 20
2. Number of ConcreteServices per AbstractService goes from 1 through to 50
3. Number of QoS attributes per application goes from 1 through 10
4. Number of Markets per AbstractService goes from 1 through 10

Note: These ranges subsume the values, we obtained from the literature review. At their upper ends, these variables together, represent applications that do not exist today. Rather, we posit that the upper ends of these ranges are an extreme level, which adequately stress Clobmas.

Measurable Scalability Goals: We can now formulate measurable scalability goals. The reported complexity of decentralized mechanisms in the literature review is around $O(2^n)$, and hence, any polynomial with low order would do, as a scalability goal. The polynomial chosen should have a order high enough to be practical, and yet low enough to be useful. We choose a biquadratic polynomial, since we want to be practical in the application of QoS contraints. Given the scaling assumptions, and the higher-level scaling goals, we require that: *within the operational bounds specified, the polynomial describing the Time-Taken-to-Adapt*:

1. As AbstractServices increase, must be biquadratic (fourth-degree) at worst. (**Scalability Goal 1**)
2. As ConcreteServices increase, must be biquadratic at worst. (**Scalability Goal 2**)
3. As QoS attributes increase, must be biquadratic at worst. (**Scalability Goal 3**)
4. As Number of Markets increase, must be biquadratic at worst.(**Scalability Goal 4**)

More detailed goal-derivation diagrams can be seen in Appendix B. We summarize the operational ranges that our system is expected to deal with, in Table 7.1

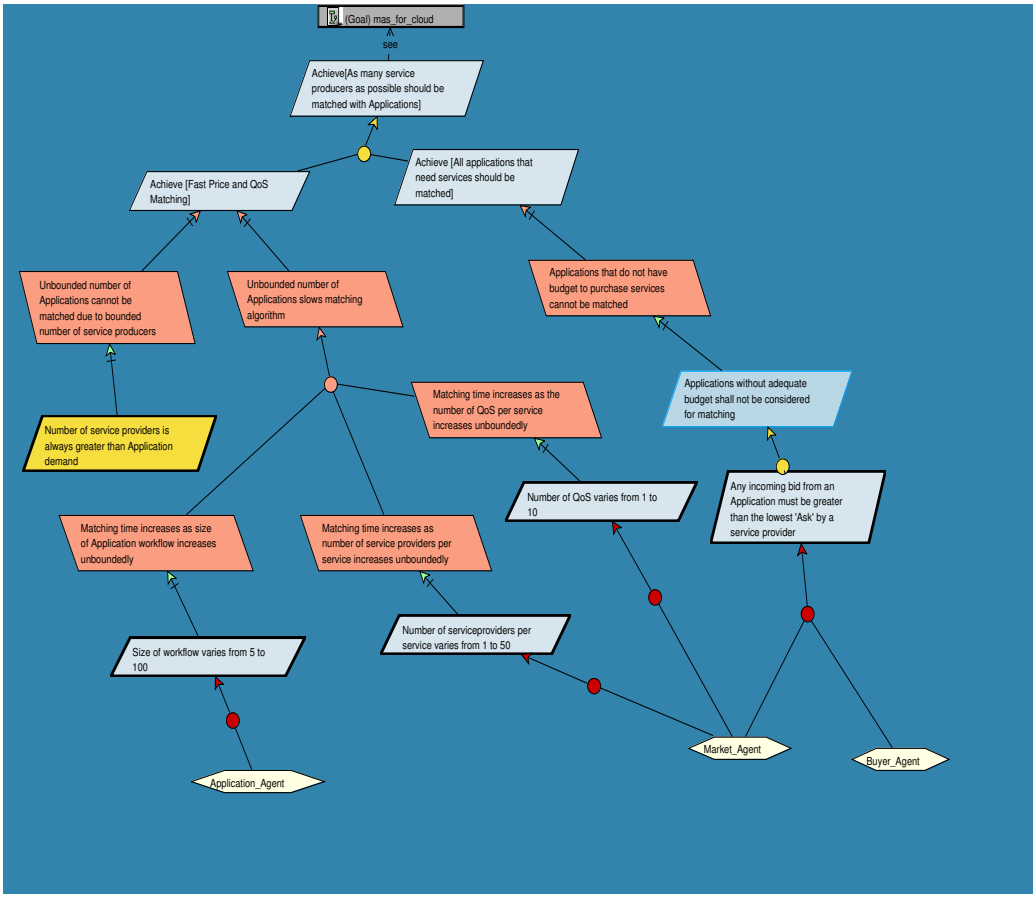


Figure 6.1: Goals for maximum possible matching and assignment to agents

Variable affecting performance	From Lit. Review	Target Goals
Number of AbstractServices in a Workflow	1–10	1–20
Candidate services per AbstractService	1–20	1–50
QoS attributes per CandidateService	1–3	1–10
Number of Markets per CandidateService	N.A.	1–10

Table 6.1: Operational range for scalability goals

6.3 Design

There are several approaches to understanding, and describing the structure and behaviour of a multi-agent system:

1. **Gaia Methodology**: This approach regards the MAS as a computational organization, with agents performing roles, enforcing rules of communication, etc[111].

2. **BDI Agents**: This approach considers the autonomy of the agents to be the most significant characteristic and therefore, documents the behaviour of an MAS as an aggregation of the *beliefs*, *desires*, and *intentions* of its constituent agents[13].

3. **Architecture-Driven Design of MAS**: This approach places the architecture at the centre of the development activity, for an MAS. Based on the development of several industrial-strength MAS, Weyns[101] believes that developing agent-based systems is 95% software engineering and 5% agent systems theory. Therefore, UML is extensively to document the architecture.

We use the last approach (Architecture-Driven Design) because:

1. UML is well-known to most practitioners of object-oriented modelling and, thus easier to communicate

2. A major part of our system's domain is software engineering, with agents that act autonomously and concurrently, being a small but necessary part of it. Therefore, we philosophically agree with [101].

Architecture-Driven Design (ADD) of a system can start when the main architectural drivers, i.e., functional and non-functional requirements of the system are known. Depending on the utility derived from each of the functional and non-functional requirements, the main requirements form the architectural drivers. Each architectural driver is refined in an iterative manner, until all of the drivers are realized or in a position to be realized. In Figure 6.2, we see that design and requirements are inter-twined, with several feedback loops.

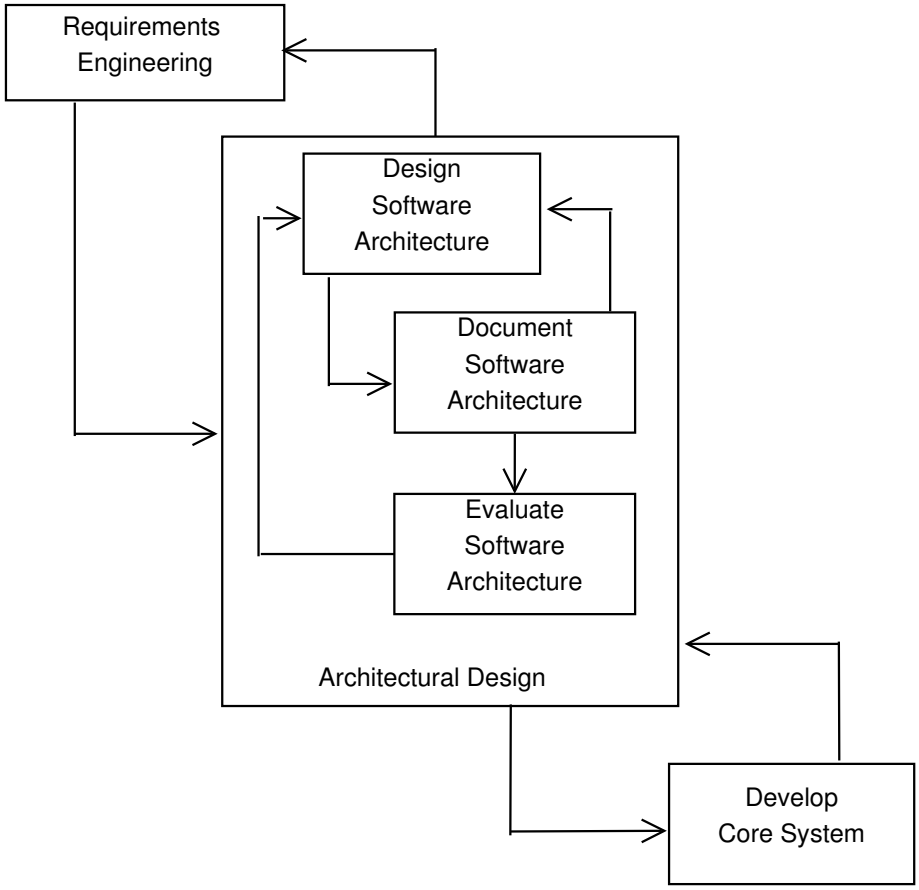


Figure 6.2: Architecture-Driven Design lifecycle

We followed the ADD methodology for designing Clobmas. First, each of the basic system functionalities was mapped onto an agent of Clobmas. Then each agent was systematically decomposed to arrive at its constituent classes, and their responsibilities.

6.3.1 Design Rationale

The two main principles underlying the design of Clobmas are: partially decentralized control and asynchronous communication. Decentralized control means that each agent acts autonomously, and thus local decision making informs a large part of Clobmas' behaviour. Although in general, agents act independent of other agents, however they will wait for instructions at specific stages, for example during initialization and during the 2-stage protocol (see Figure 6.11). Thus, the mechanism is not completely decentralized. Rather, it is a hybrid of partly centralized, and partly decentralized control. Asynchronous communication is really a design force that acts on a decentralized system, and allows it to function effectively. Registration with markets, preparing Bids, making decisions about multiple possible transactions, are all done in an asynchronous fashion.

6.3.2 Architectural Pattern

An architectural pattern describes the kind of components, connectors and their topology, that are known to be useful in resolving certain problems in software design. Patterns represent the accumulated wisdom of multiple practitioners, on known ways of problem-solving. Typically, each pattern has certain strengths, and weaknesses. It is the responsibility of the system designers to weigh the advantages of using a certain pattern, against the liabilities it incurs. In our case, we chose the Publish-Subscribe pattern to model the communication and control within our system. This pattern decouples the communication between the components, thus allowing dynamic changes in the structure of the system. Components can dynamically change whether they want to react to changes in a certain components or not. In a decentralized system, this capability is critical to the functioning of the system. We instantiate the Publish-Subscribe pattern through the use of distributed events. Each agent has a list of events that it wants to be notified of, and it registers with the appropriate agent for this purpose.

We combine the use of Publish-Subscribe with the Broker Pattern. The Broker Pattern decou-

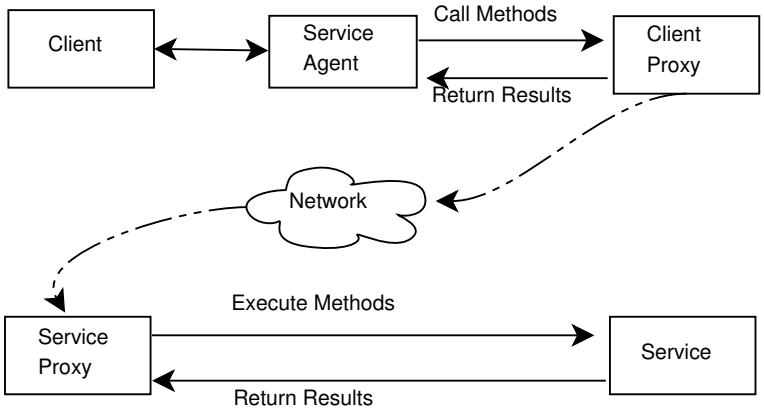


Figure 6.3: The Broker Pattern

ples the point of request, from the point of service (Figure 6.3). That is, when the client makes a service request from the service agent, the agent forwards this request across a network. When computation is done by the actual service, the results are communicated back to the client, via the service agent. This pattern insulates the client from low-level details such as the physical location of the service, network-protocols to reach the service, etc. Therefore, if we substitute the service with another that is functionally equivalent, the client will not be able to tell the difference.

In the following sections, we talk about the high-level design of the system in detail. We first model Clobmas structurally, and then we look at the behavioural aspects of Clobmas.

6.3.3 Structural Modelling

We document the system 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. Using these diagrams, we wish to highlight the modular nature of Clobmas, and how easily it can be changed to accommodate new code.

High level Package Diagram: A package diagram shows the coarse-level organization of code in a system, with the packages and sub-packages demarcating the encapsulation of concepts in the system. In Figure 6.4, we see that all the agents, viz., buyer, seller, application,

market are created and maintained separately from concepts like **Strategy** and **Communication**. This independence of packaging also highlights the various axes along which Clobmas could change its behaviour. Thus, Clobmas could change its auction style from a continuous double auction to a double auction with a clearing house, with no effect on the other parts of the system. In section 5.6.1, we had explicated on the usage of ZIP to modify Bid and Ask prices. This can be easily changed to an implementation of (say) Roth-Erev[83], Gjerstad-Dickhaut[37] or AA[97], by simply implementing the PricingStrategy interface.

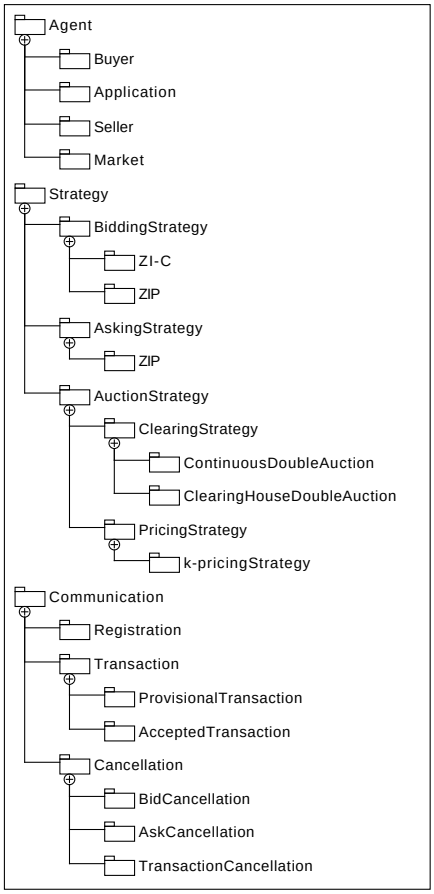


Figure 6.4: Package diagram of entities in Clobmas

Component Diagram: A component diagram illustrates how a system is structurally organized, with communication and dependencies between entities. There are several different definitions of *component*. We do not attempt to provide yet another definition, but use the

common features of most definitions, that it is a *unit of reuse and replacement*[88]. In Figure 6.5, we see a subset of the components, and their communication links. The component *Service Registry* is depicted as a separate entity, since it is a third-party component, and we do not propose to have any control over it. The communication link shown between *Seller* and *Service Registry* is therefore necessarily one that is likely to be different for every *Service Registry*. While not explicitly enforcing a particular deployment, this architecture hints at the distribution of components, amongst several entities.

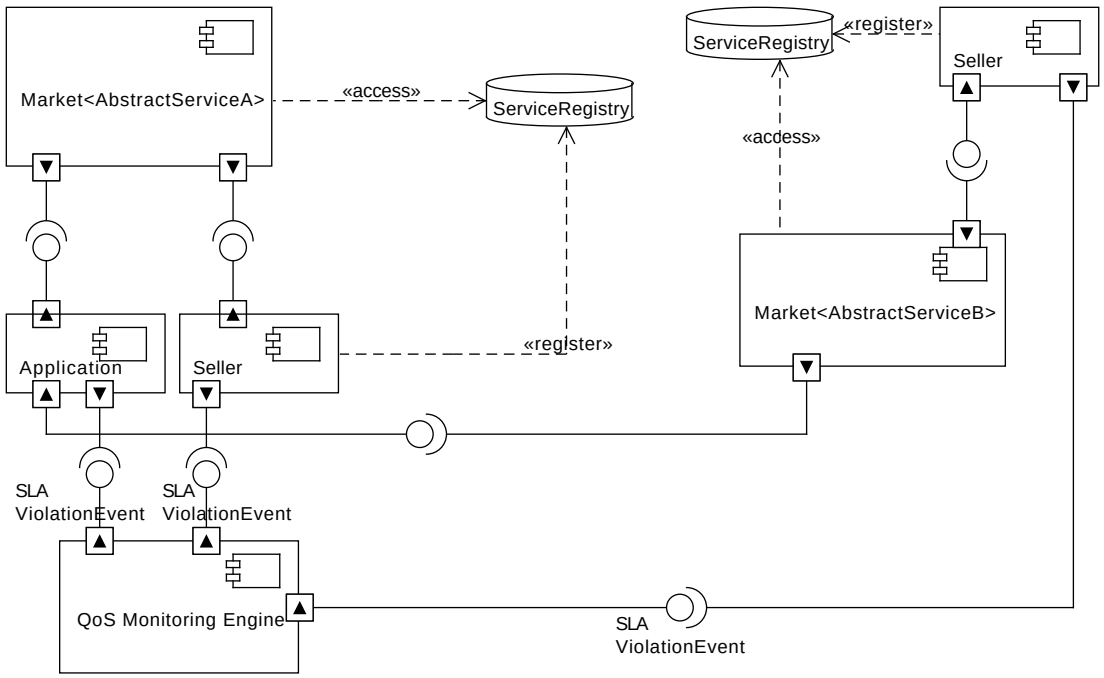


Figure 6.5: Component diagram of entities in Clobmas

Class Diagram: The class diagram shows the static relationships between the main entities in the MAS. It drills down and provides an implementation-level design view of the system. The Bid and Ask interfaces shown in Figure 6.6 are also shown in the component diagram (see figure 6.5), with the *Market* class implementing both of them.

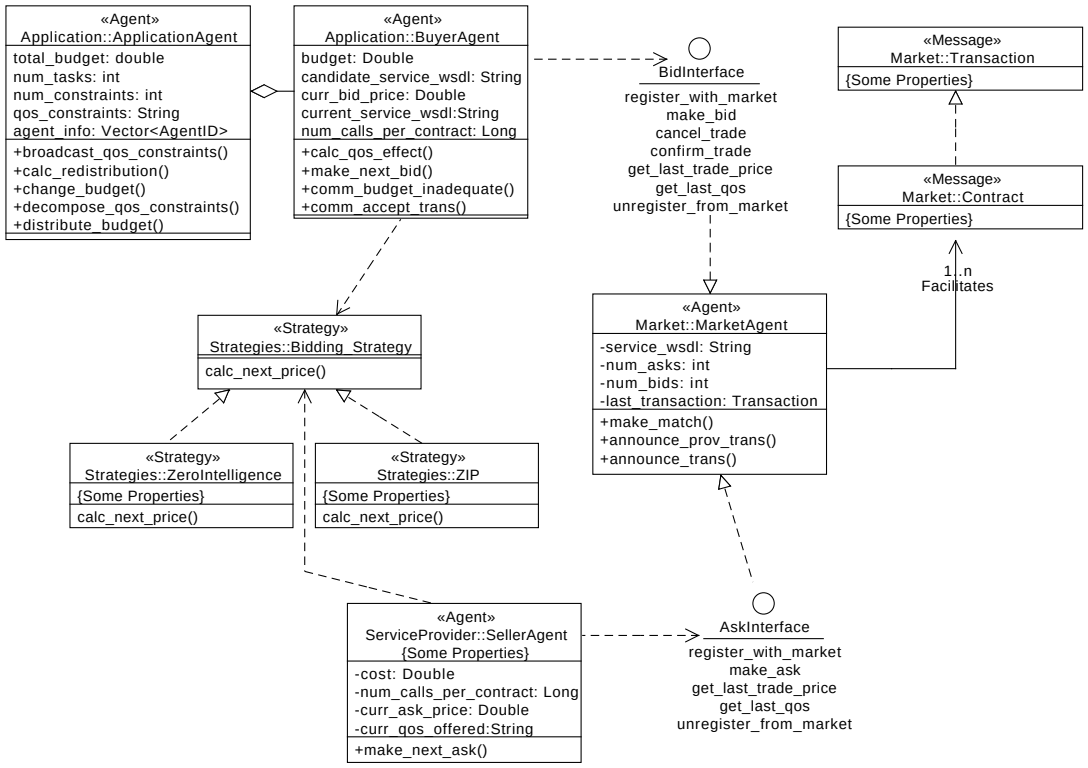


Figure 6.6: Class diagram of entities in Clobmas

6.3.4 Behavioural Modelling

Interaction 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. The standard activity diagrams in UML are *Sequence Diagrams* and *Collaboration Diagrams*. However, the semantics in both these diagrams assumes a single thread of control, i.e., each object that sends or receives a message

is assumed to be the only active object at that time instant. This is not true in the case of Clobmas. To deal with these situations, the agent community has evolved AUML (Agent-UML) which modifies the semantics of a sequence diagram, to allow for the possibilities above. We document the communication steps of our mechanism through AUML diagrams.

6.3.4.1 Setup Phase

In the following figures, we show the communication for an application with two Abstract-Services, `A` and `B`, and its corresponding BuyerAgents and the market agent. The *Setup Phase* (Figure 6.7) 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.

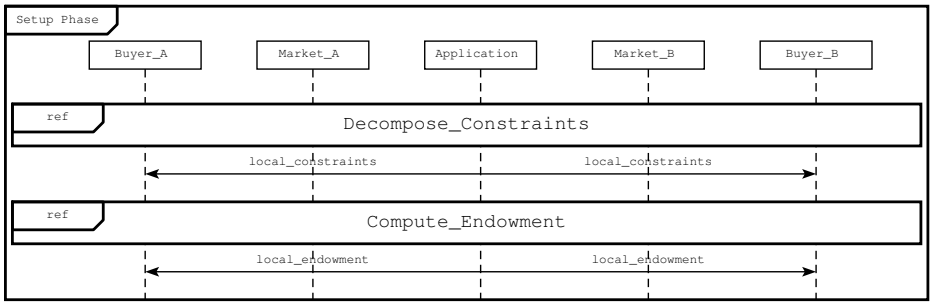


Figure 6.7: Setup phase for an application with two AbstractServices A & B

Decomposing constraints: Decomposing constraints (Figure 6.8) 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[22] and checks whether the combination of QoS attributes 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.

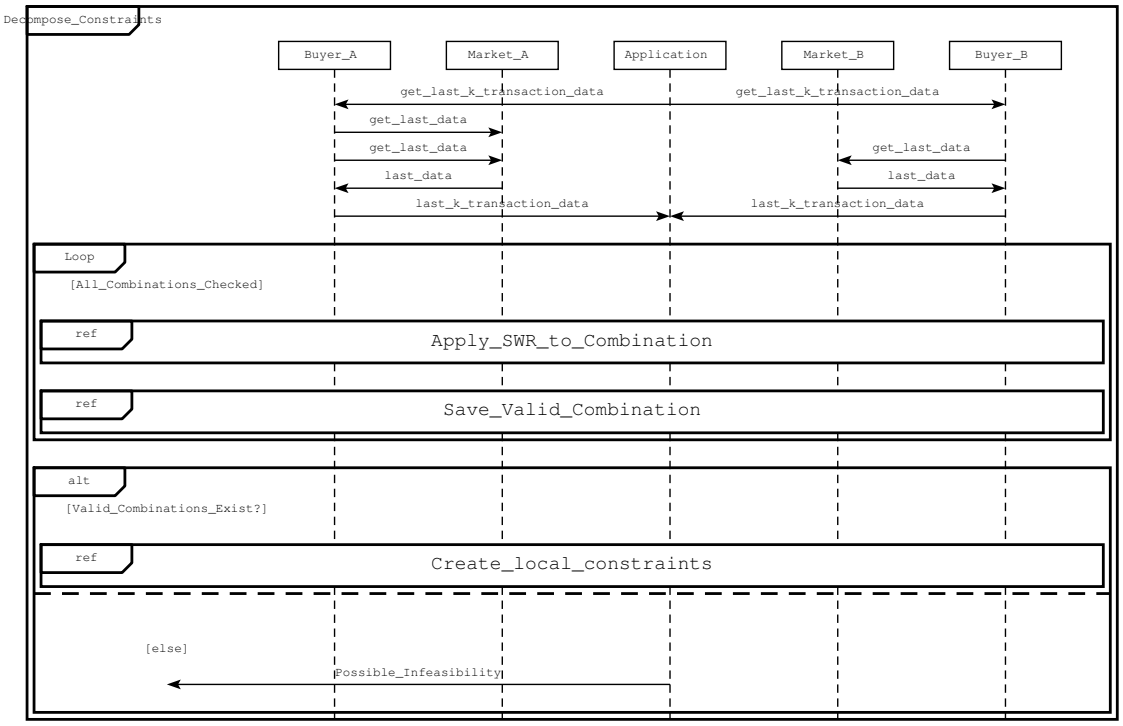


Figure 6.8: Application decomposes its constraints into local constraints

Compute Endowment: The budget for the individual agents is split in the ratio of the transaction prices that are prevalent in the individual markets. 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.

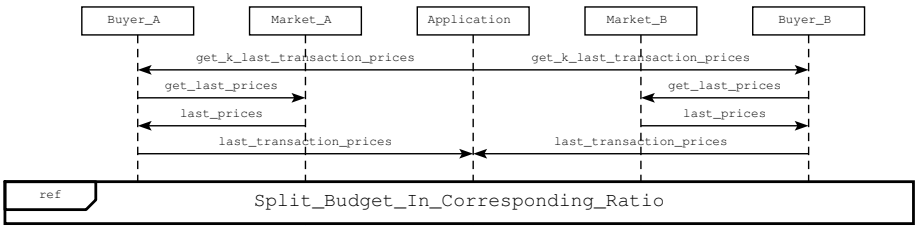


Figure 6.9: Application computes endowment for its BuyerAgents

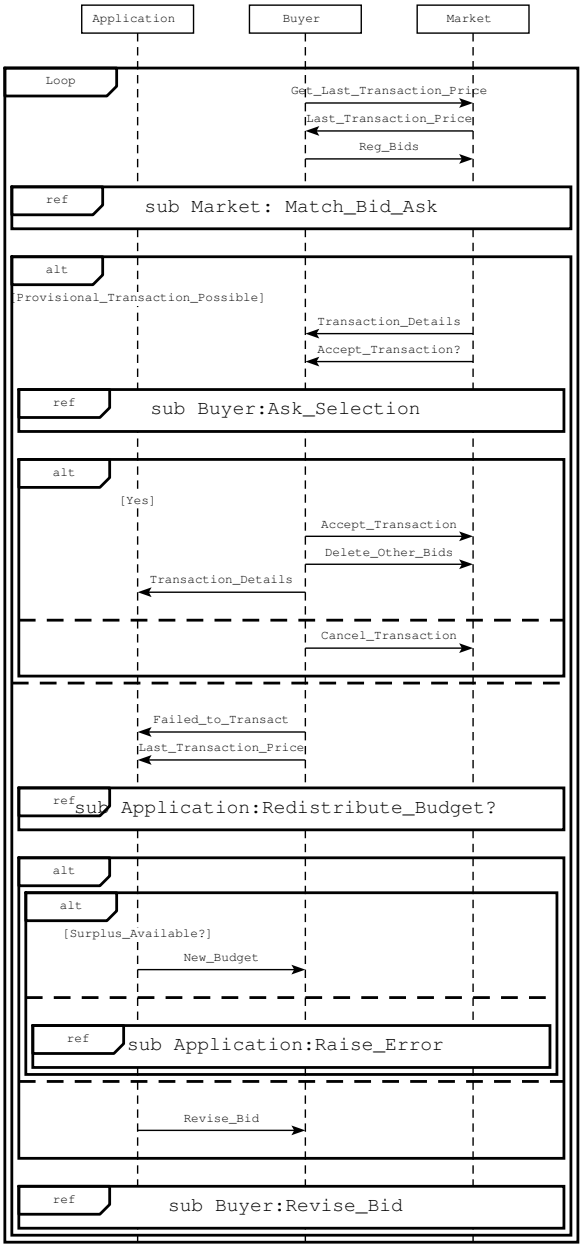


Figure 6.10: The trading phase of buying a service

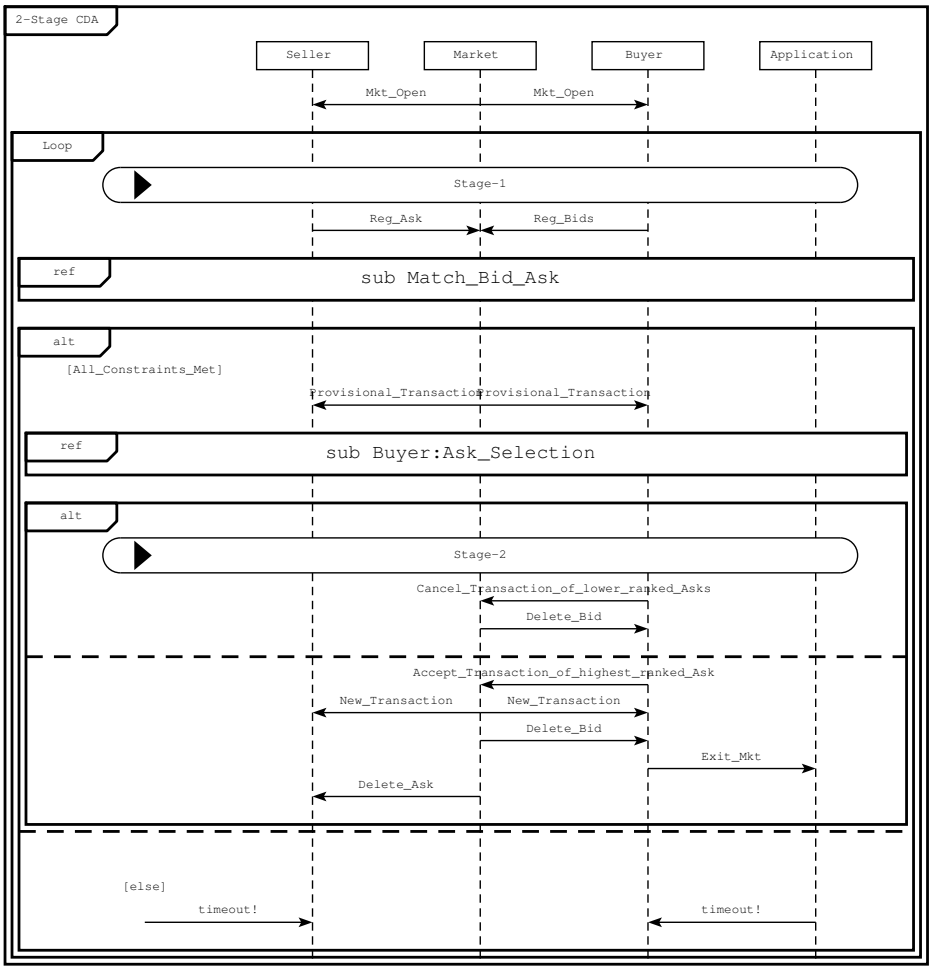


Figure 6.11: Two-stage CDA protocol

6.3.4.2 The Trading Phase

In Figure 6.10, 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 6.11). Based on whether a trade occurs or not, the application evaluates whether it needs to re-distribute 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. 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 (see Section 5.4.4.1). 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.

6.3.5 Implementing vs. Simulating a MAS

As Steve Phelps notes, there are many issues with implementing a MAS, specially in the context of evaluating the MAS itself for mechanism design[82]. In many cases, it is preferable to simulate a MAS, instead of implementing one. We now detail why we chose a simulation:

1. **Reproducibility**: The most important issue is, that we want to be able to reproduce the results of an experiment exactly, given the starting conditions. Using a real MAS, this may not be possible since it is impossible to reproduce the environment exactly. This is particularly so, when the interaction involves human decision-making of setting QoS targets, QoS violations by other services, etc. While simulating a MAS, all of these can be modelled as processes that happen with a certain probability.

2. **Interaction with the Environment**: A real MAS has to be tailored to deal with several conditions like varying computational power available to different agents, keeping track of real-world time, race conditions that occur due to parallelism amongst distributed machines. All of these are implementation issues, and serve no purpose in evaluating the basic structural and behavioural properties of our mechanism. To concentrate on multiple types of situations, it is handy to be able to ignore such issues, and hence simulating a MAS is much more suitable.

3. **Programmer Libraries**: On a more practical level, the standard libraries used for implementing randomness (for example, `rand()` from `libc`) have lower periods than are recommended for scientific experimentation. Simulation toolkits like Repast, SimPy, Matlab, etc. provide high-quality *pseudo-random number generators* (PRNGs), which can be used for statistical validation of experiments.

In light of these reasons, we decided to simulate a MAS, with human interaction and QoS violations being modelled by processes running with some probability. These do not change the architecture of the MAS, only that the parallelism of the real world is modelled using a sequential computer.

6.4 Conclusion

In this chapter, we described the process that we used to ascertain the requirements, and elucidated the scalability requirements. We started with abstract quality goals, which were refined into scalability goals. These goals were further refined with scaling obstacles, and finally, after concretizing the scaling assumptions, we arrived at measurable, and specific goals. We then described the design of Clobmas, both structurally, and behaviourally. In the next chapter, we evaluate Clobmas for its functional, as well as scalability properties.

CHAPTER 7

Evaluating Clobmas

You can tear a poem apart to see what makes it tick. You're back with the mystery of having been moved by words. The best craftsmanship always leaves holes and gaps, so that something that is not in the poem can creep, crawl, flash or thunder in.

Dylan Thomas

7.1 Introduction

In the previous chapter, we looked at Clobmas's architecture, and its scalability goals. In this chapter, we look evaluating Clobmas. We evaluate Clobmas in two stages. The first stage of evaluation is functional evaluation. This is to ensure that Clobmas meets the core objectives that it was set up to fulfill. The second stage of evaluation is to judge whether Clobmas possesses desirable non-functional properties. Note, we seek only that Clobmas satisfies the functional, and scalability goals, and not that it optimizes. In this context, this thesis is evaluated as follows:

1. We evaluate Clobmas from the perspective of BizInt (functional)
2. We evaluate Clobmas from the perspective of SkyCompute (functional)
3. We evaluate the architecture of Clobmas and reflect on the architectural choices (behavioural)

7.2 Context for Evaluation

In order to set the context for evaluation, we now reprise the example that we had introduced in chapter 1 (sections 1.1.1 and 1.1.2).

BizInt, a small 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 7.1, 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

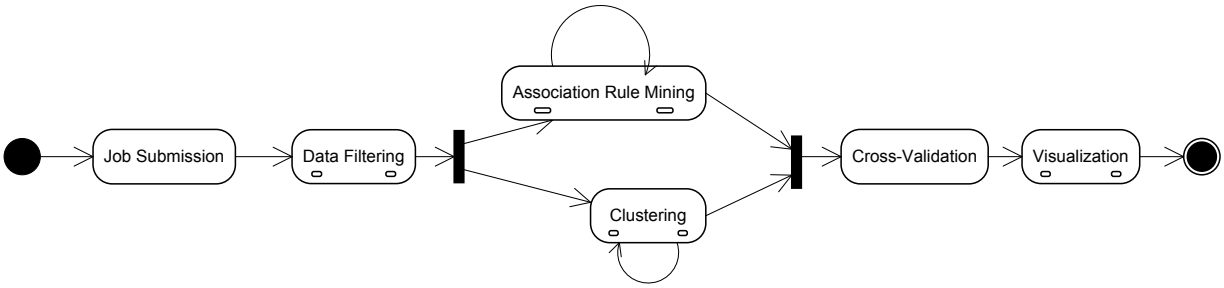


Figure 7.1: BizInt's Workflow constructed using composite services from the hosting cloud

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 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 framework 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. Depending on the QoS provided by the service, the cost of using the service will change. SkyCompute enables this, by implementing various versions of its indexing service (say), one which is extremely fast but does not use SSL, another which uses SSL but has lower availability, yet another that has high availability, SSL and high performance. However, implementing and keeping services available is a cost-burden on SkyCompute. In

other words, regardless of the goodness of its services, any service that is not being utilized by an application, is a drain on SkyCompute's finances. Hence, SkyCompute would like to ensure that any mechanism that it offers for self-adaptation results in high utilization of its services.

7.2.1 Qualitative Criteria

Thus, Clobmas must fulfill the following criteria:

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

Like most research on dynamic service composition, we evaluate the goodness of a particular set of services, by means of a utility function. Given a set of services, we use their associated QoS to calculate application's end-to-end QoS (see Chapter 5, Section 5.4.1). 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*. 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**. 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:

$$Utility = \sum_{k=1}^{K} w_k * V(k) \tag{7.1}$$

where

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

In order to measure how well BizInt has adapted, we define the following variables:

1. **Utility Achieved:** The number of trading rounds that an application is satisfied, over a trading period

2. **Time to Reach QoS:** Number of trading rounds that it takes for an application to be satisfied, after an adaptation event

We measure the goodness of Clobmas for SkyCompute by the following variables:

1. **Market Satisfaction Rate (MSR):** The percentage of applications, out of all applications, that have been satisfied.

2. **Time to Market Satisfaction (Time-to-MSR):** The time taken to achieve a certain level of Market Satisfaction Rate. To evaluate for scalability, we first specify a certain level of MSR, and then measure Time-to-MSR, under varying conditions.
   Note that Time-to-MSR is a concretization of Time-to-Adapt from the scalability goals (Chapter 6, Section 6.2.1.1).

7.2.2 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 7.1

Variable affecting performance	From Lit. Review	Scalability Goals
Number of AbstractServices in a Workflow	10	1–20
CandidateServices per AbstractService	20	1–50
QoS attributes per CandidateService	3	1–10
Number of markets per CandidateService	1	1–10

Table 7.1: Operational range for scalability goals

7.3 Experimental Setup

Although open-source cloud implementations like OpenNebula[1], and Eucalyptus[2] are freely available for download, they are primarily targetted at the IaaS market. Since we are concerned with SaaS, these tools do not help us. In the same vein, using simulators such as CloudSim would require major modifications. CloudSim is a fairly new toolkit, and does not have the ability to explore market mechanisms in a sophisticated manner [14].

Software In this scenario, we wrote our own simulator in Python (v. 2.6.5), using a discrete-event simulation library, SimPy[3]. 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

[1] www.opennebula.org
[2] http://www.eucalyptus.com/
[3] http://simpy.sourceforge.net/

a generator of very high-quality pseudo-random numbers [67]. We use this generator due to the fact it was designed with Monte Carlo and other statistical simulations in mind. [1]

Reporting All simulations are reported as an average of a 100 simulations. The simplest configuration chosen is the recommended value, from the literature review. Therefore, each simulation reported, unless otherwise specified, was run with the following parameters given in Table 7.2. In the section on scalability (subsection 7.6.2), we stress the mechanism by scaling up variables on each of the axes.

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)	0.01
Applications simultaneously adapting	300

Table 7.2: System parameters and their standard values

External Adaptation Event: Once an application achieves its required QoS, it withdraws from trading until an adaptation event (internal or external) occurs. A violation of the SLA by a service counts as an External Adapation Event. We model the occurrence of this event by a small probability, based on a study conducted by Cloud Harmony [27]. Thus, with a small probability, the ApplicationAgent is sent an ExternalAdaptationEvent, which causes the BuyerAgents to re-enter the market.

[1] For a k-bit word length, the Mersenne Twister generates an almost uniform distribution in the range

$$[0, 2^k - 1]$$

.

Internal Adaptation Event: A change in the QoS required, or a change in the available budget counts as an Internal Adaptation Event. Each ApplicationAgent is sent an Internal Adaptation Event randomly. We also test a specific case, where all ApplicationAgents get an Internal Adaptation Agent simultaneously (as a Market shock).

7.4 Results

We simulate applications 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 trading agents (buyers) can never bid above their budgets. Each application generates a random level of QoS that it must achieve.

An external adaptation event occurs when the application decides to change its required QoS for any reason (say, a new job requires different QoS levels). At some random time, every application receives a different job that requires different QoS levels. This prompts that particular application to adapt, and therefore, approach the market again.

7.5 Evaluation from BizInt's Perspective

BizInt is interested in the satisfaction achieved by an average application. An application is said to be satisfied, if the *quality gap* achieved by its agents, is within the tolerance zone. The tolerance zone is the range of values specified by the tolerance level, both **above and below** the *ideal utility level*. For the purpose of our experiments, the tolerance level for each numeric QoS was fixed at 1%. Once the quality gap is within this tolerance zone, agents stop adapting. By this we mean that they withdraw from the market. The agents will only re-enter the market, in case of an adaptation event. In Figure 7.2 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 7.2, we show the adaptation occurring under conditions of normal distribution of QoS demand amongst applications and supply in the cloud. To compare the goodness of trading in a CDA market, we contrast its use in a *posted-offer* market.

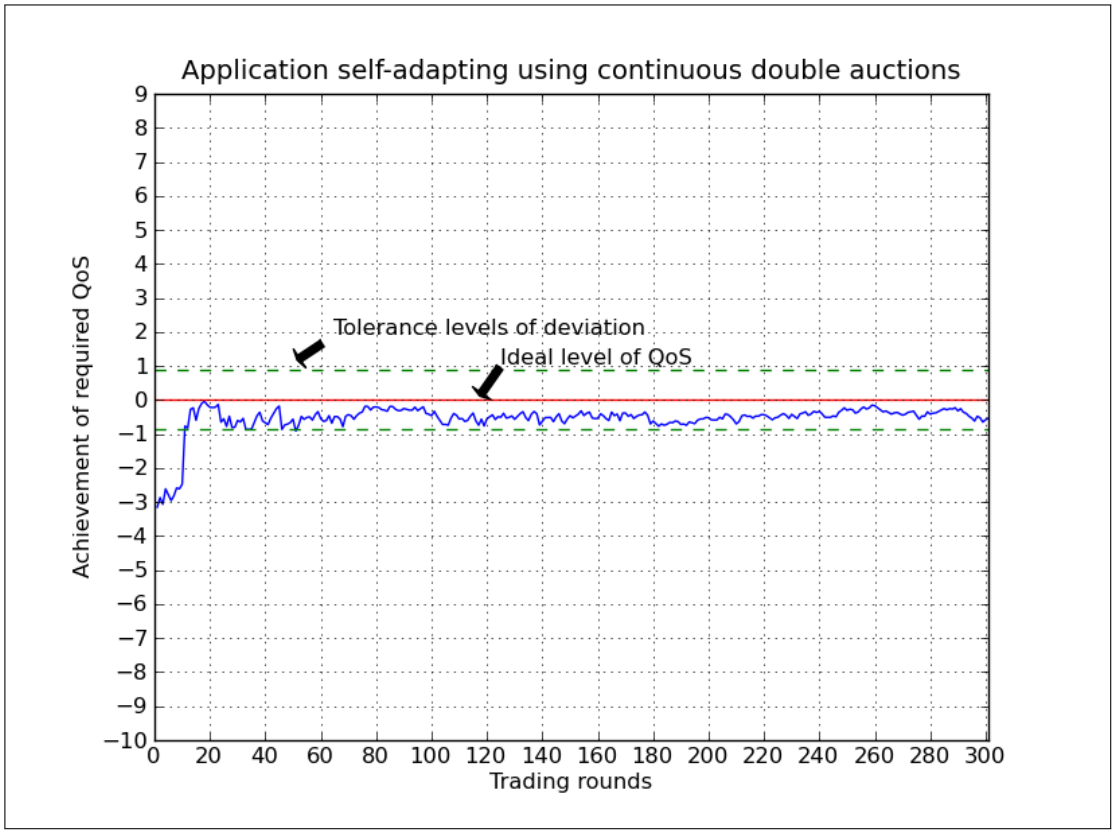


Figure 7.2: Utility gained by adaptation by a single application

Posted Offer Market: 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*. This is the current prevailing mechanism for buying services on the cloud. In such a scenario, the probability that an application will find services at the *bid-price*, decreases.

In Figure 7.3, we see that the application is able to adapt and acquire the QoS that it requires.

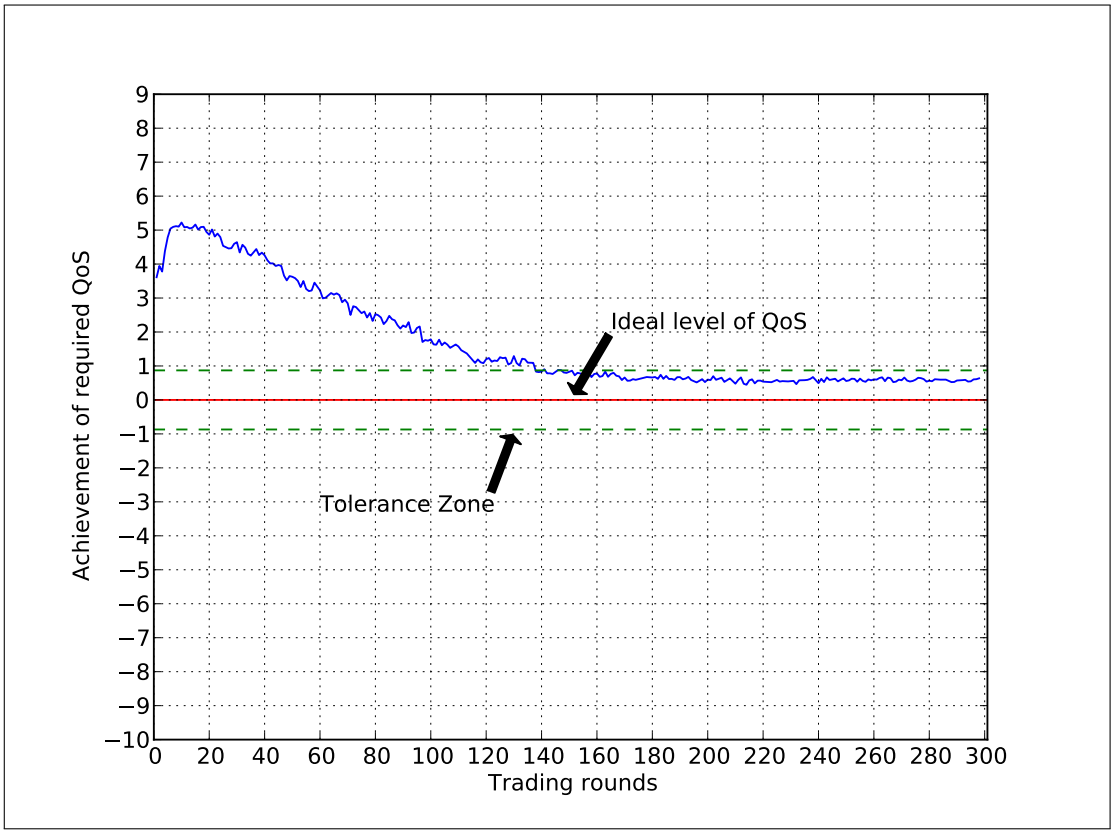


Figure 7.3: Utility gained by a single application in a posted offer market

However, it is obvious from comparing the two figures, that in Figure 7.2, the application is able to reach the tolerance zone a lot quicker than the application in Figure 7.3. 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 BuyerAgents 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	Posted-Offer
Number of rounds satisfied	282	160
Time to reach QoS	18	140

Table 7.3: Comparative performance of adaptation in CDA vis-a-vis Posted-Offer

In the next section, we look at adaptation in the aggregate, i.e., from the perspective of all the market participants. SkyCompute would like to look at market-wide measures, rather than individual measures.

7.6 Evaluation from SkyCompute's Perspective

SkyCompute would like to enable infrastructural support for Clobmas, only if the market-based mechanism ensures that a high percentage of the services that it has available for use, are used by applications. In other words, the use of Clobmas must result in a higher utilisation of SkyCompute's services, than with a competing mechanism. The competing mechanism, as we discussed previously, is the one currently used by cloud providers, viz. *posted-offer*. Again, intuitively, the *posted-offer* mechanism will result in a lesser number of trades, as compared to a *CDA* mechanism.

7.6.1 Market Satisfaction Rate

We focus on the performance of the market as a whole, in enabling applications to self-adapt. The market's performance is measured in terms of number of applications successfully able to achieve their target QoS. 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 7.4). As expected, it performs quite poorly, with only

10-20% of applications managing to acquire the required QoS. Our mechanism achieves a much higher level of applications, able to attain their desired level of QoS(see Figure 7.5). Adaptation using our mechanism allows 85% of the applications in the market to adapt. Note, that this figure only indicates applications that have their QoS within the tolerance zone. There are many applications that have overshot their QoS requirements, but are unable to find sellers of services with lower QoS values.

In Figure 7.4, we see the effects of using a Zero-Intelligence strategy. Unsurprisingly, applications using Zero-Intelligence are unable to satisfy their QoS requirements. On an average, only about 20% of the applications in the market reach their desired QoS, which is clearly unsatisfactory. SkyCompute would not be able to sustain a model where only 20% of its hosted applications are satisfied with their QoS requirements.

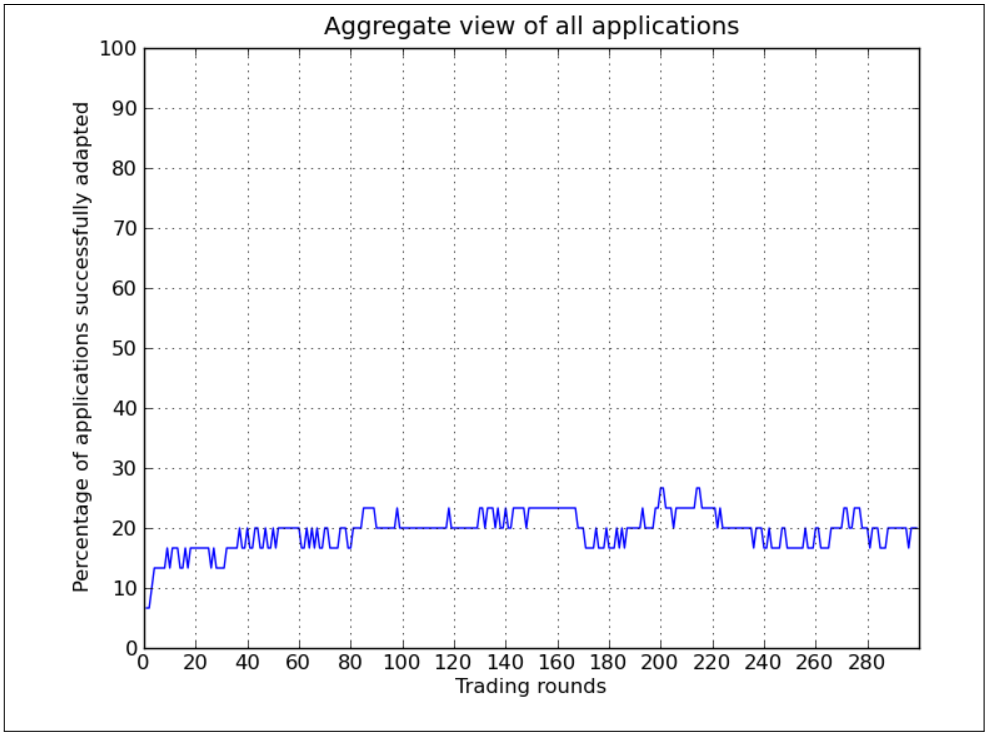


Figure 7.4: Efficiency of adaptation in a CDA market with *Zero Intelligence*

On the other hand, we see (in Figure 7.5) 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.

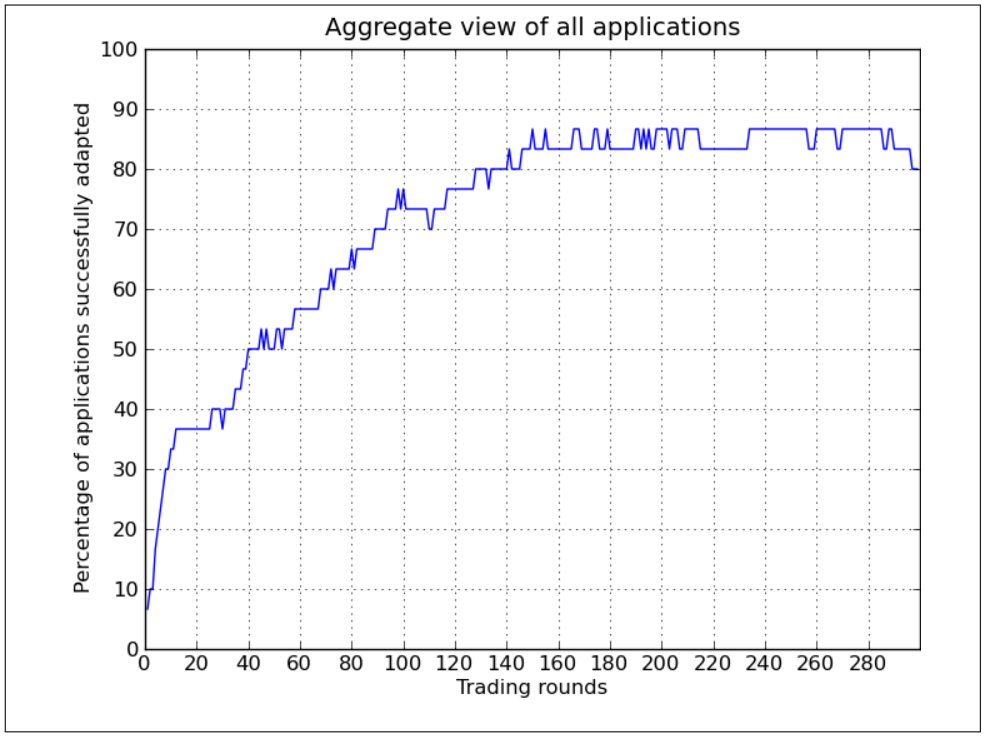


Figure 7.5: Efficiency of adaptation in a CDA market

Quite similar to the CDA, the posted-offer also performs well. In this case, the sellers *never* adjust their prices, but the buyers do. For buyer-agents' adjustment, we continue to use ZIP.

Market Conditions :

Figure 7.5 shows the adaptation of applications, when the demand and supply of QoS in services follow a normal distribution. However, this may not always be the case. Conditions like paucity of supply, has the effect of less applications being able to satisfy their QoS requirements. Therefore, we investigate the following kinds of market conditions:

1. A normal distribution of QoS amongst the services sold, but skewed distribution amongst buyers representing applications. This situation occurs in the case where a particular QoS is suddenly in high demand. For example, in the wake of a high-profile security flaw being publicized, all applications will increase their demand for services that use a high level of encryption.

2. A normal distribution of QoS amongst services demanded by applications, but skewed

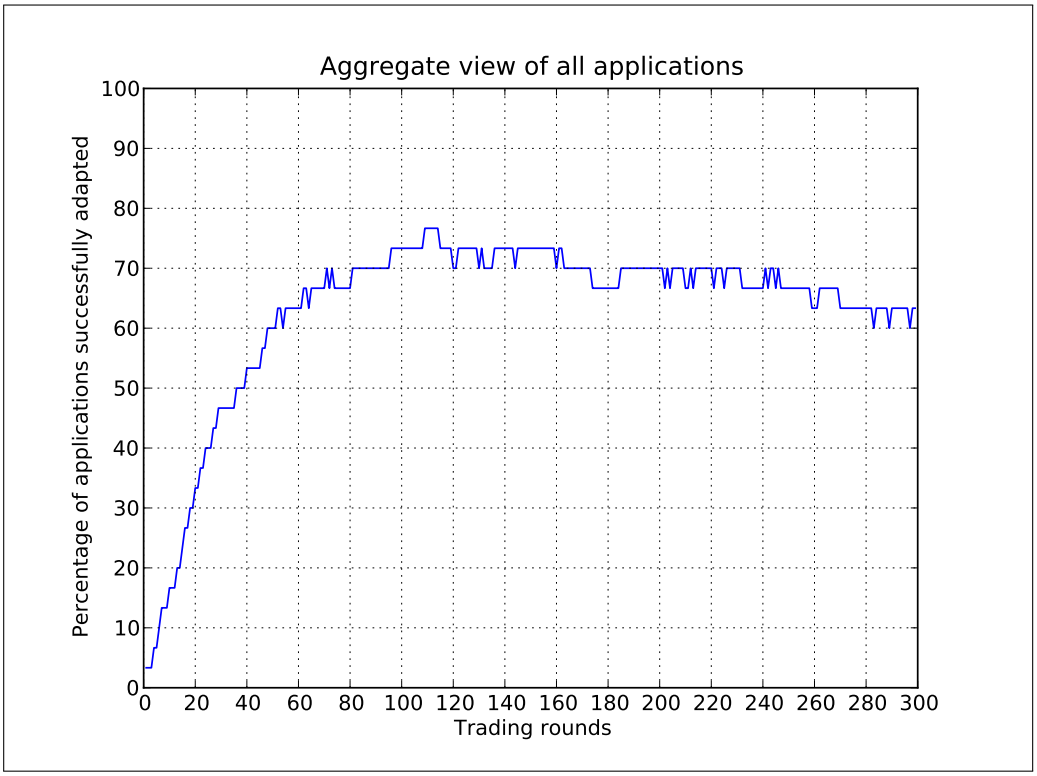


Figure 7.6: Efficiency of adaptation in a Posted Offer market

distribution of QoS values in services supplied by SkyCompute. Such a situation occurs when SkyCompute mis-reads the demand for a particular QoS, and changes the supply of services (and correspondingly their prices). The impact of this is felt in the number of applications that are able to successfully adapt and attain their required QoS within budget. This is the worst of all conditions, since not only are the applications unable to attain their QoS, but there are a large number of services with SkyCompute, that are lying idle.

3. A bimodal distribution of QoS values amongst both services provided and services demanded by applications. This situation occurs when there is a natural grouping in the market. Low-end consumers that are price-sensitive demand a lower QoS value, while high-end consumers that are QoS sensitive demand a higher QoS value and are willing to pay for it. In such a case, SkyCompute also attempts to capture both ends of the market by providing services at both QoS levels. This situation, while still less than optimal, is better than the previous one, since the natural schism in the market

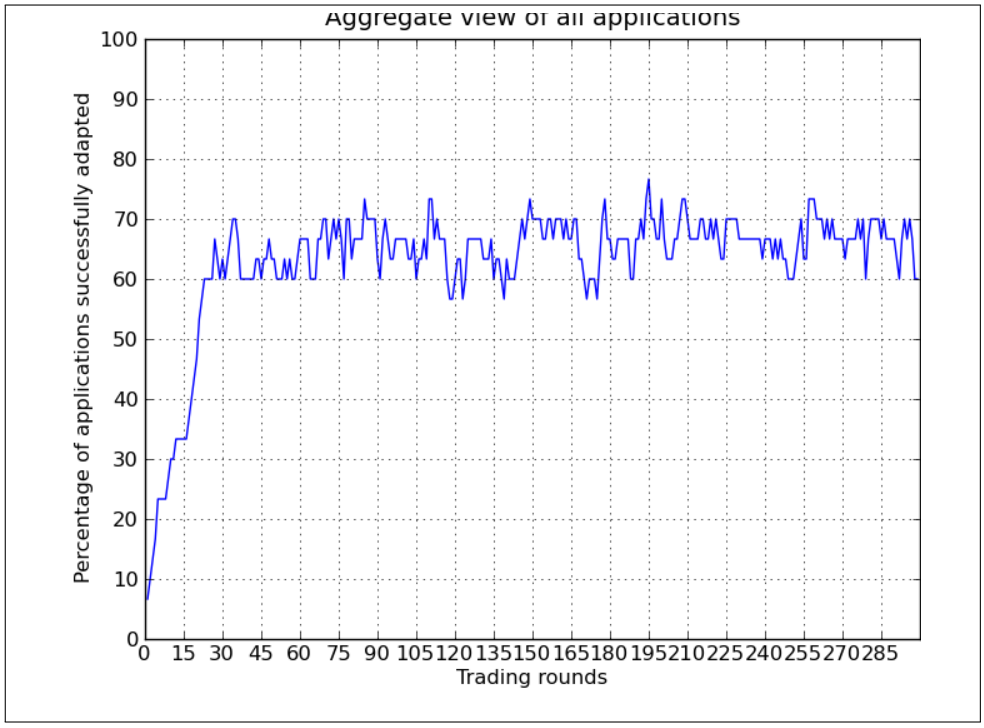


Figure 7.7: A normal distribution of QoS values amongst ConcreteServices, but skewed distribution of QoS values amongst BuyerAgents

means that the number of applications in the middle of the spectrum are relatively few in number.

Skewed Buyers and Uniform Sellers Figure 7.7 shows the state of market satisfaction, where the buyers have a skewed preference for QoS values. Applications that are able to acquire their preferred services hold on to them, and hence there is relatively little fluctuation in the MSR. Although, there is a small amount of fluctuations, the actual level of market satisfaction is lower than that shown with a uniform distribution (in Figure 7.5)

Uniform Buyers and Skewed Sellers We see in Figure 7.8, that this is a terrible situation to be in. Market satisfaction rates rarely go above 70% and frequently fall below 60%.

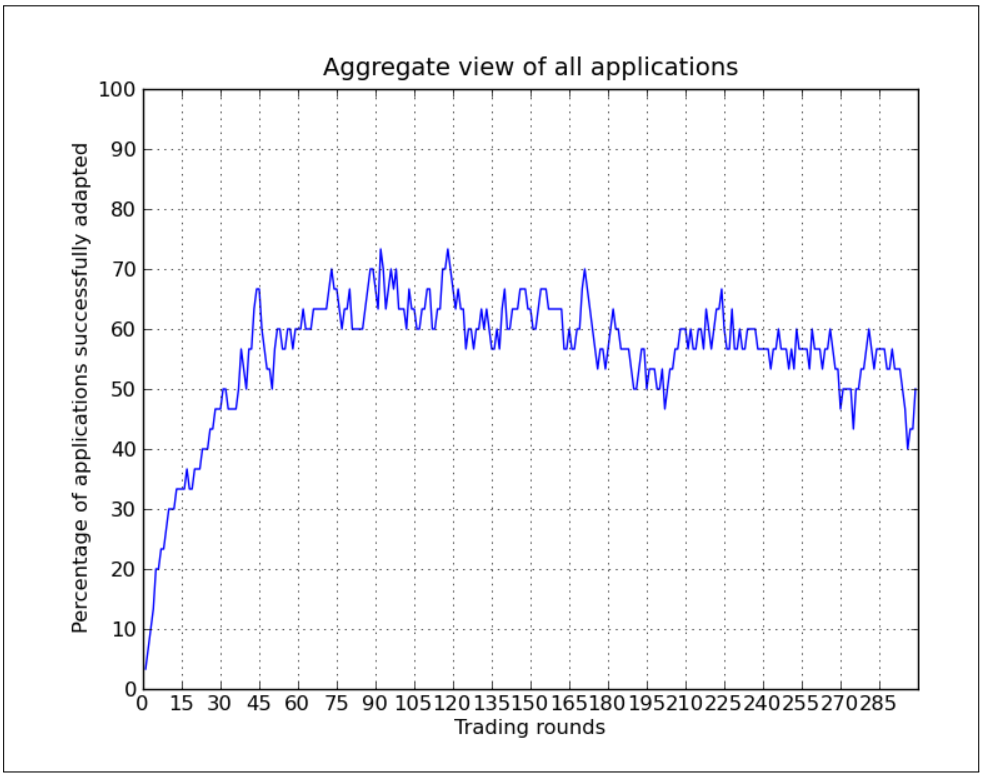


Figure 7.8: A normal distribution of QoS values demanded by the BuyerAgents, but skewed distribution of QoS values amongst ConcreteServices available

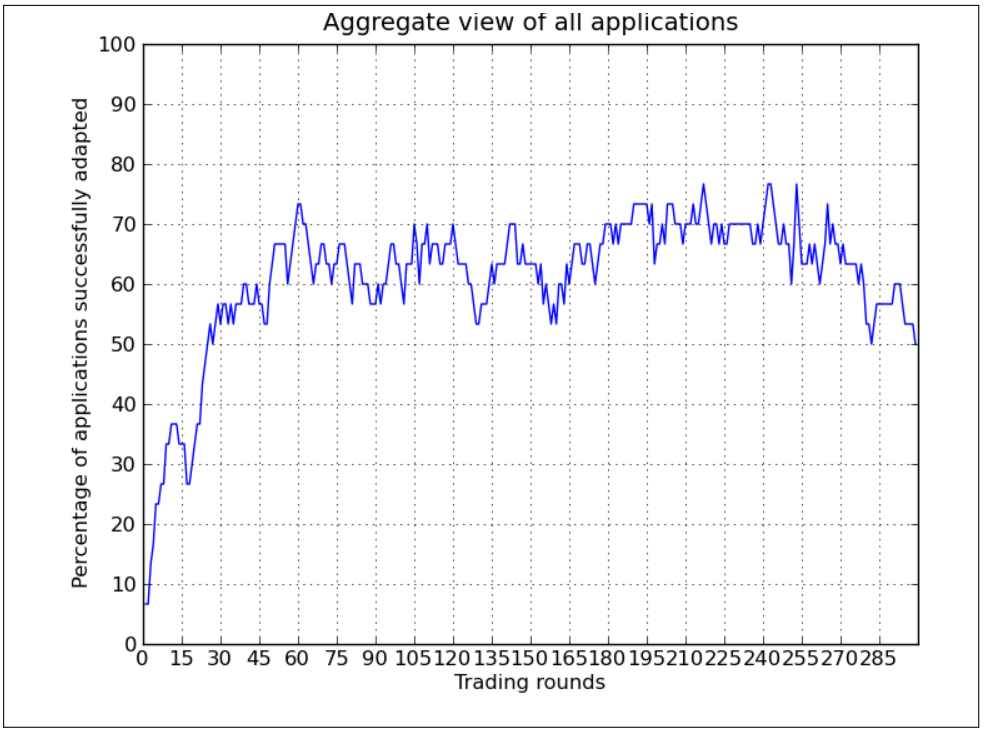


Figure 7.9: A bimodal distribution of QoS values amongst both BuyerAgents and ConcreteServices

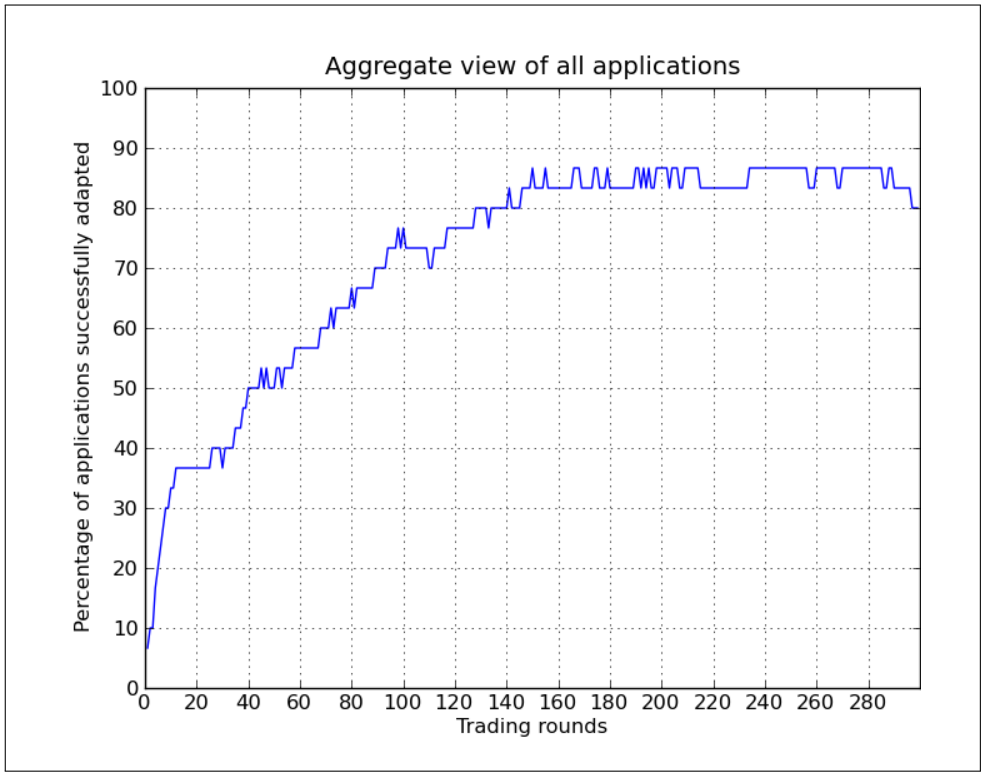


Figure 7.10: Independent probability of change in QoS Demand

Skewed Buyer and Skewed Seller Figure 7.9 shows the effect on market satisfaction, when there is a bimodal demand and supply of QoS values. We see that the market satisfaction rates vary mainly between 60% and 70%, with some dips below 60% and some peaks higher than 70%.

Internal Adaptation Events:

In Figures 7.10 and 7.11, we compare the differences in the aggregate behaviour of all applications, in response to internal adaptation events of two kinds. In the first instance, we model all applications having independent probabilities of changing their target QoS attributes, over time. This means, any application could change its target QoS, at any point in time. In the second instance, we simulate all applications changing their target QoS, every 25 rounds of trading. Each application that succeeds in reaching its target, changes its target QoS with a small probability. We see that in the aggregate, there's not much change in the percentage of applications being able to self-adapt.

In Figure 7.11, we administer a market shock in the form of an Internal Adaptation Event to

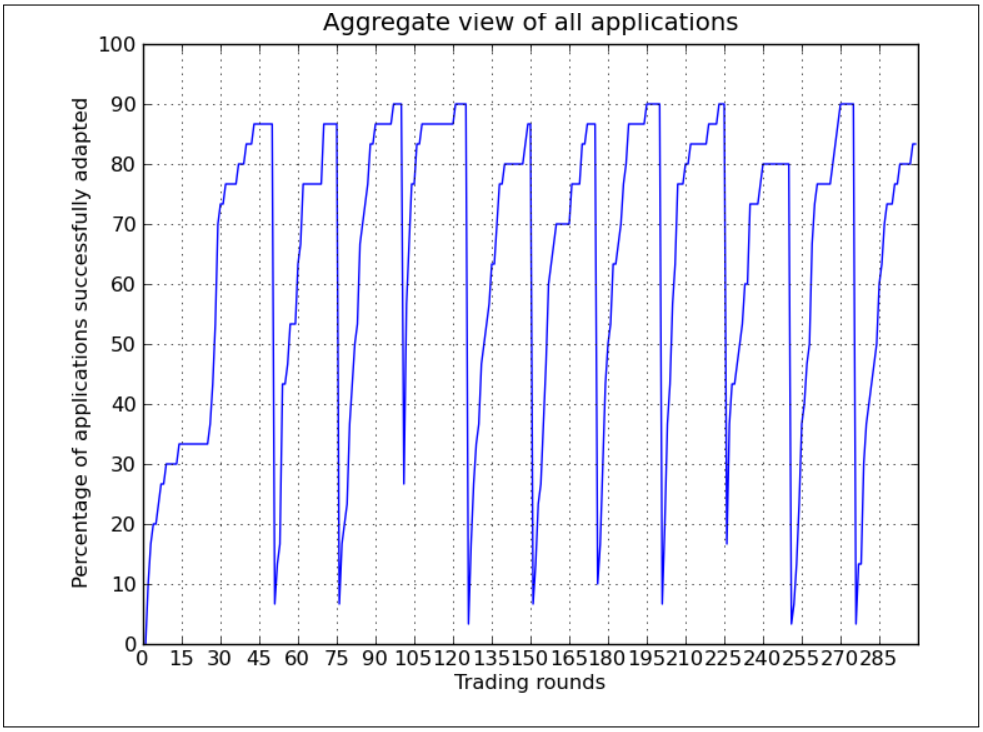


Figure 7.11: Market shocks after every 25 rounds of trading

every ApplicationAgent. This has the effect of all BuyerAgents re-approaching the market for new services. This makes the aggregate adaptation easier to see, since the market, as a whole, experiences a change in demand, periodically. Predictably, the periodic external event causes the market satisfaction rate to drop precipitously, every 25 rounds. But we see that the MSR is able to get back to a desired level within **approximately 15 rounds**.

7.6.2 Scalability

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. To this end, we choose a **market satisfaction rate of 80%**, and measure the time taken to achieve this level. The adaptation process can be highly dependent on many variables. In chapters 3.8 and 5, we elucidated on the variables that are most important to us, and the operational ranges that they are expected to take. In this section, we tease out

how each of these variables affect the time taken for adaptation. We reproduce the scalability target goals in Table 7.4 below, that we introduced previously.

Variable affecting performance	Scalability Goals
Number of AbstractServices in a Workflow	1–20
CandidateServices per AbstractService	1–50
QoS attributes considered per CandidateService	1–10
Number of Markets considered per CandidateService	1–10

Table 7.4: Operational range for scalability goals

In each case, we also perform a polynomial curve-fitting to find an equation that approximates the worst-case scenario, of an axis(x) and Time-to-MSR(y) with a $residual\ of \leq 1\ second$. This lets us know in concrete terms, whether we have achieved our scalability goals or not[1].

AbstractServices Vs. CandidateServices: Arguably, these are the variables that change most often from application to application, and from time to time. 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 available per AbstractService 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.

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

$$y = -0.0361x^3 + 4.7619x^2 - 5.4985x + 3.8978 \tag{7.2}$$

In the worst case, the polynomial growth of Time-to-MSR (y) when CandidateServices (x)

[1]The data for all subsequent graphs is online at: www.cs.bham.ac.uk/~vxn851/graphdata.tar.bz2

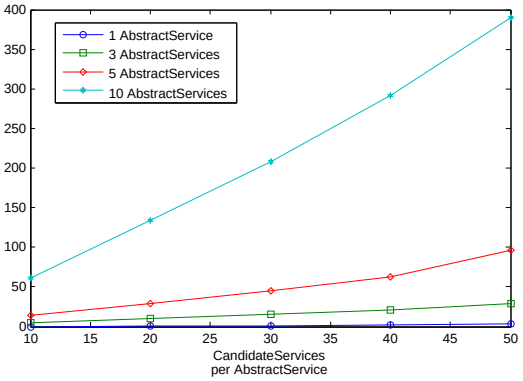


Figure 7.12: Time taken for adaptation when CandidateServices per AbstractService increase

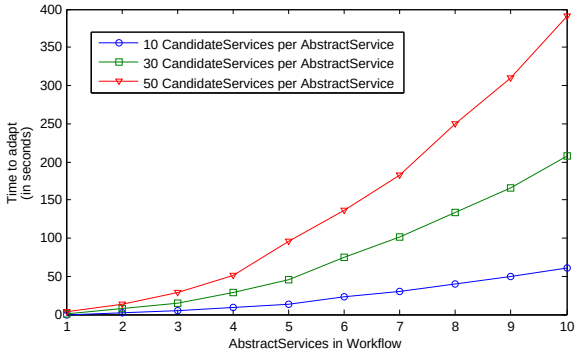


Figure 7.13: Time taken for adaptation when AbstractServices in Workflow increase

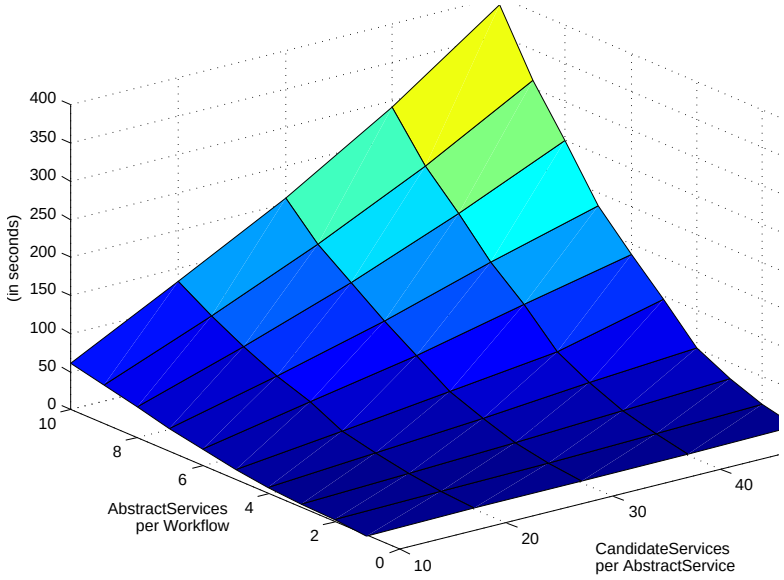


Figure 7.14: Both AbstractServices and CandidateServices increase

increase with regard to AbstractServices, is given by

$$y = 0.0012x^3 - 0.0611x^2 + 8.2678x - 16.6664 \tag{7.3}$$

From the equations (Equation 7.2 and Equation 7.3), we see that Clobmas is more sensitive to the number of CandidateServices than to the number of AbstractServices (see Figures 7.12, 7.13 and 7.14). Although, both equations are cubic in nature, the cubic term in Equation 7.2 is negative. This indicates that it grows slower than Equation 7.3. This makes intuitive sense, since the increase in AbstractServices merely increase the number of BuyerAgents, which are distributed in any case.

AbstractServices Vs. QoS attributes per Service: Another variable that applications are most interested in, after CandidateServices, is the number of QoS attributes per service. Clearly, since every CandidateService needs to be examined for all the QoS attributes, the time taken for service selection increases. In the following graphs, we only show how time increases when numeric QoS attributes are changed. For boolean and categoric QoS attributes, decision making is much faster, since end-to-end calculations can be broken down and performed in a decentralized manner by individual agents. As we see in Figure 7.17, the time taken for achievement of QoS increases polynomially.

In the worst case, the polynomial equation describing the growth of Time-to-MSR (y), when QoS attributes (x) increase with regard to AbstractServices is given by:

$$y = -0.0387x^3 + 0.7186x^2 + -1.6377x + 20.3967 \tag{7.4}$$

In the worst case, the polynomial equation describing the growth of Time-to-MSR (y) when AbstractServices (x) increase with regard to QoS attributes, is given by:

$$y = 0.0458x^2 + 0.9252x - 0.5624 \tag{7.5}$$

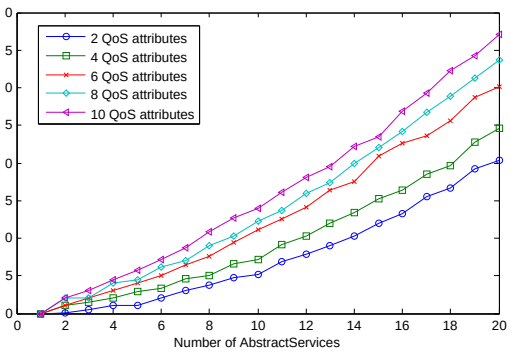


Figure 7.15: Time taken for adaptation when AbstractServices increase

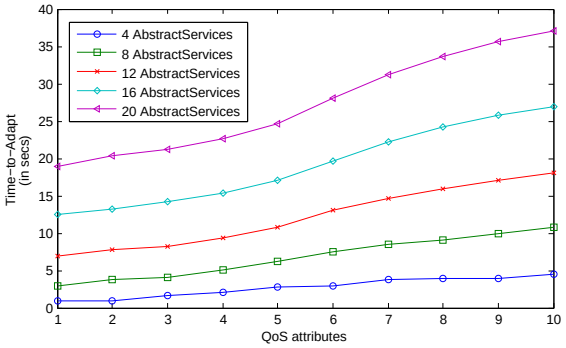


Figure 7.16: Time taken for adaptation when QoS increase

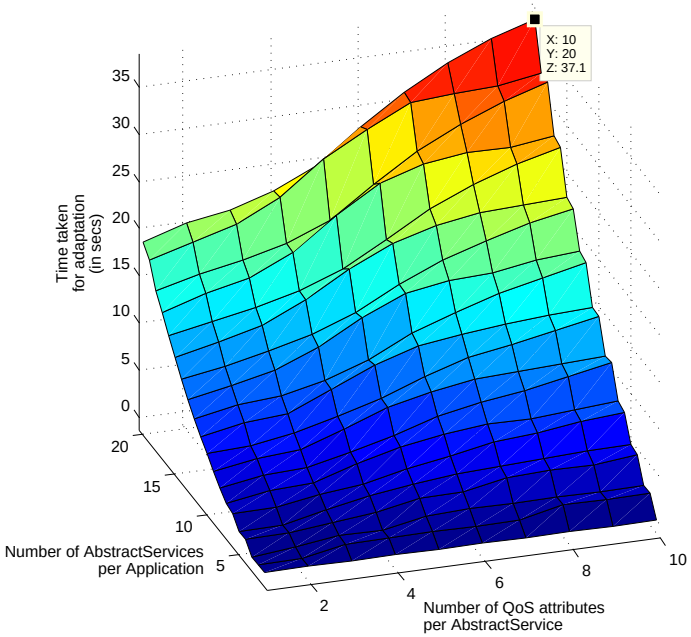


Figure 7.17: When both, the number of AbstractServices and QoS attributes, increase

Equations 7.4 and 7.5, show how AbstractServices and QoS attributes interact. Equation 7.5 grows slower than Equation 7.4, since it is quadratic in nature. Clobmas deals with increase in AbstractServices better, than increase in QoS attributes. Again, this is quite intuitive, since an increase in AbstractServices means an increase in BuyerAgents, but an increase in QoS attributes involves increased comparisons across all Agents.

CandidateServices Vs. QoS attributes per Service: As the number of applications that use Clobmas increase, it will have to make a decision on whether to increase the number of CandidateServices per AbstractService, or increase the number of QoS attributes available per AbstractService. Regardless of the business implications of each, both will have an impact on the scalability of Clobmas.

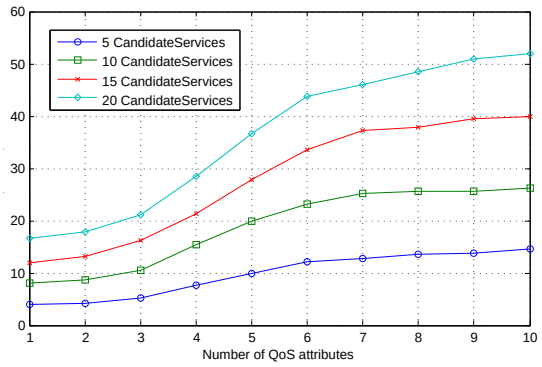


Figure 7.18: Time taken for adaptation when CandidateServices increase

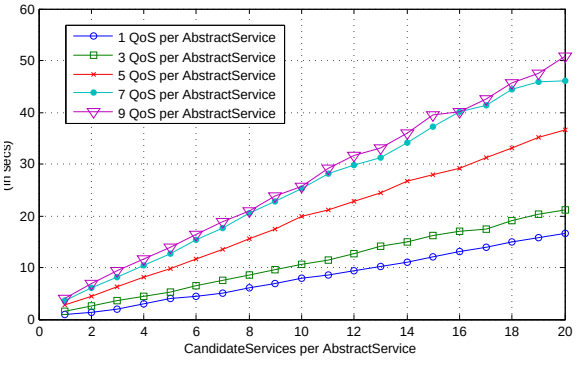


Figure 7.19: Time taken for adaptation when QoS increase

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

$$y = 0.0303x^4 - 0.7769x^3 + 6.4554x^2 - 14.4945x + 25.7250 \tag{7.6}$$

In the worst case, the polynomial describing the growth of Time-to-MSR (y) as CandidateSer-

vices (x) increase with regard to QoS attributes is given by

$$y = 0.0030x^2 + 2.3721x + 2.4722 \tag{7.7}$$

Equations 7.6 and 7.7 indicate that in terms of Time-to-MSR, it is preferable to increase CandidateServices than QoS attributes. The equation describing the increase due to QoS attribute (Equation 7.6) is actually biquadratic. This touches on the boundary of the scalability goal that we had set, however since its coefficient is small, and the cubic term is negative, we consider it to be within acceptable scalability range.

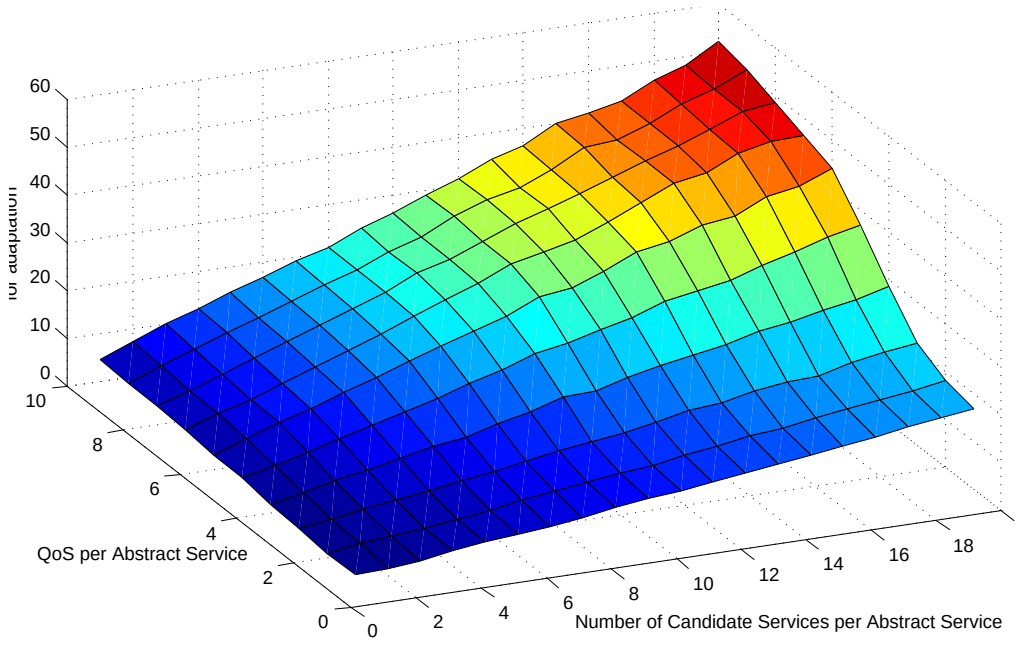


Figure 7.20: Both CandidateServices and QoS increase

The implications of the equations are borne out visually. We see in Figure 7.20 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 increase in number of CandidateServices.

7.6.3 The Cost of Decentralization

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? Increasing the number of markets increases the robustness of the system.

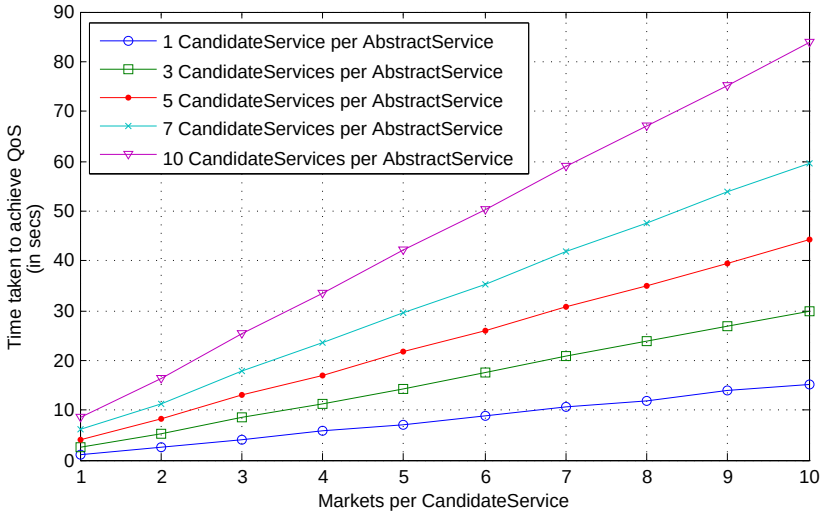


Figure 7.21: CandidateServices increase per Market

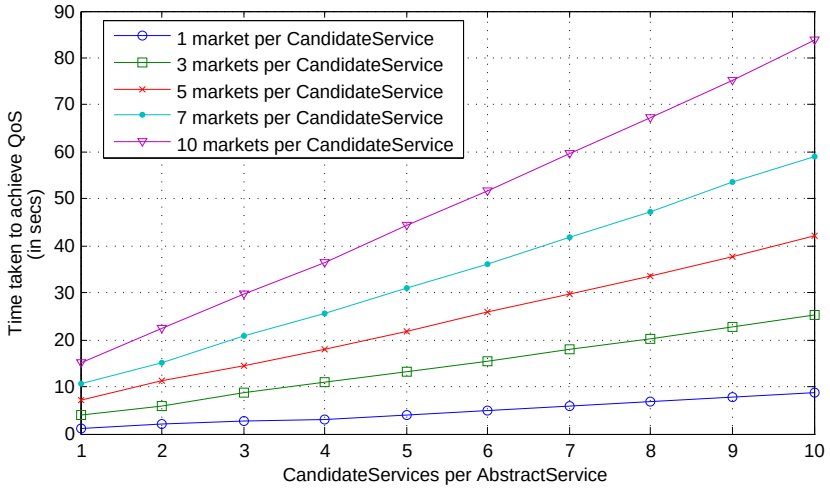


Figure 7.22: Markets increase per CandidateService

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

$$y = 8.3745x + 0.0800 \tag{7.8}$$

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

$$y = 7.5972x + 6.8222 \tag{7.9}$$

We see from Figures 7.21 and 7.22 that the slopes of the lines is 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-MSR. The equations (Equation 7.8 and Equation 7.9) bear this out, as they are both linear.

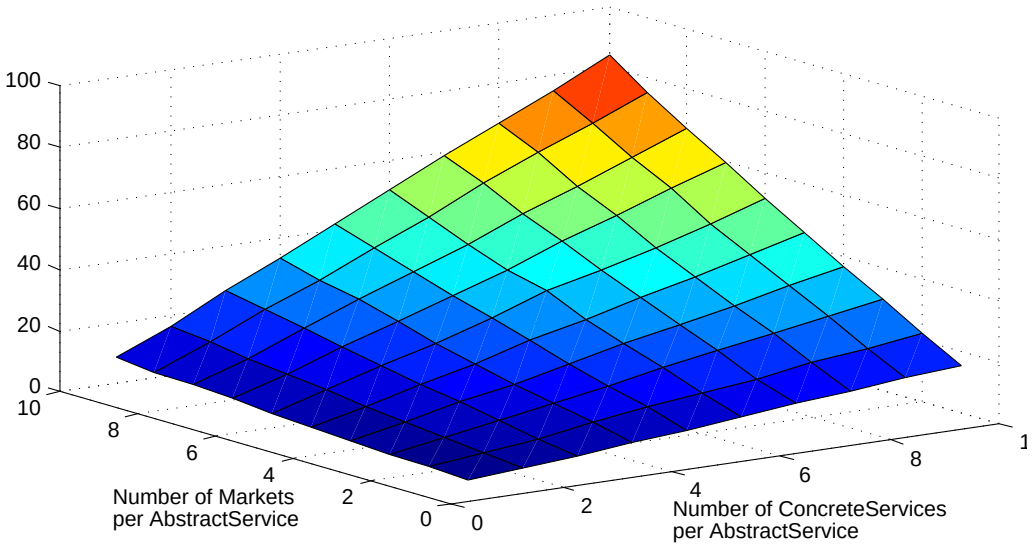


Figure 7.23: Both Candidate Services and Markets increase

Hence, the decision to increase robustness, by increasing the number of markets, does not negatively impact the time-to-MSR exhibited by Clobmas.

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.

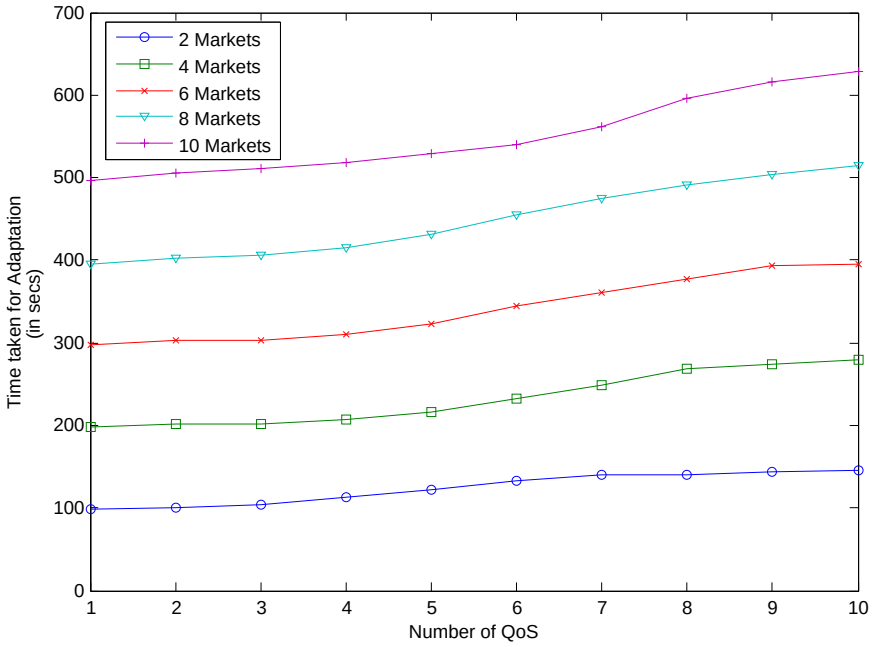


Figure 7.24: Markets increase per QoS

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

$$y = -0.4595x^2 + 65.0048x + 26.6683 \tag{7.10}$$

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

$$y = -0.1103x^4 + 2.3380x^3 - 14.9209x^2 + 42.0606x + 465.6389 \tag{7.11}$$

As is expected by now, the increase in QoS attributes causes Time-to-MSR rise in a biquadratic way, as opposed to increase in number of Markets (Equation 7.10 and Equation 7.11). Again, the biquadratic term in the equation (Equation 7.11) has a negative coefficient, so the scale of growth is actually cubic.

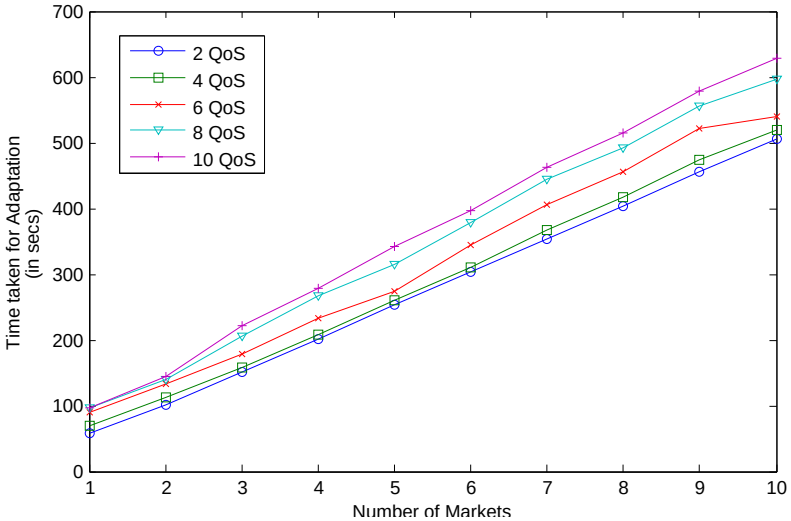


Figure 7.25: QoS increase per Market

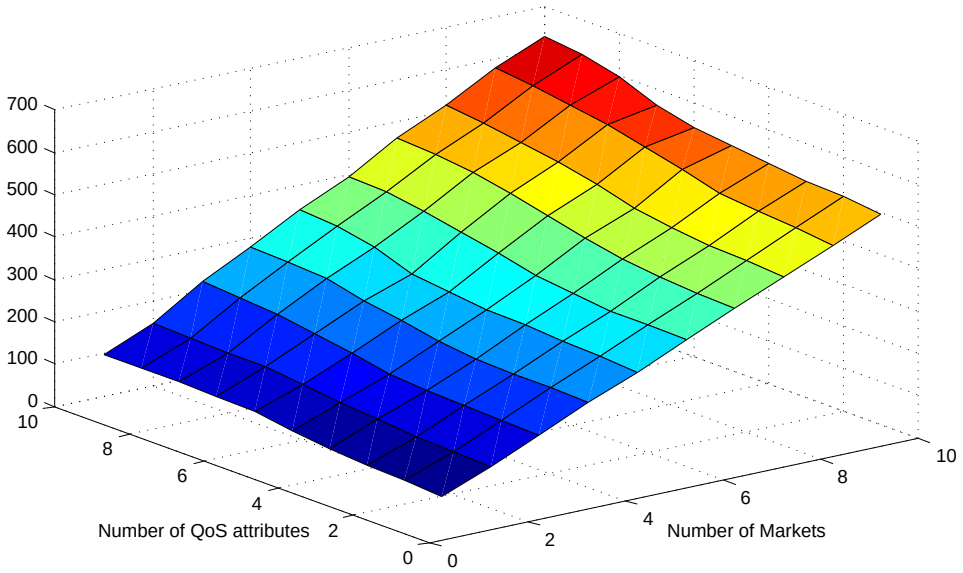


Figure 7.26: Both QoS attributes and markets increase

This leads us to the conclusion that, of the four variables that affect service-selection, Clobmas is most sensitive to QoS attributes. However, it should also be noted that these are worst-case scenarios. In all of our experiments, we used Numeric Constraints as QoS attributes, which have to be calculated on an application-wide basis. In reality, it is more likely that some applications will have Boolean or Categoric Constraints, which can be handled locally by the BuyerAgents. Hence, in the general case, Clobmas is likely to perform better with increase in QoS, with biquadratic growth being the worst case.

Meeting Scalability Goals Since none of the variables exhibited growth in Time-to-MSR, with a fifth-degree polynomial, we can say that our scalability goals have been met. More specifically:

1. From Equation 7.2 and Equation 7.5, we see that **Scalability Goal 1** is met.
2. From Equation 7.3 and Equation 7.7, we see that **Scalability Goal 2** is met.
3. From Equation 7.4 and Equation 7.6, we see that **Scalability Goal 3** is met.
4. From Equation 7.8 and Equation 7.10, we see that **Scalability Goal 4** is met.

7.7 Evaluating the architecture of the MAS

Don't fight forces, use them.

R. Buckminster Fuller

The architecture of a software system is perhaps the important artefact generated during the engineering of a system. A Software Architecture is a succint yet sufficient documentation of a software system, that enables various stakeholders to understand and reason about various properties of the system. It enables the client and requirements engineers to see that the system is capable of fulfilling the tasks for which it was constructed. It enables

designers to view the components, connectors and the constraints between them, that give rise to the functional and non-functional properties of the system. It enables programmers to understand the framework within which their components and connectors must live, and it allows testers to selectively test parts of the software system.
Now we concern ourselves with the designer's view. We elaborate on the choice of architectural pattern used, and how they affect the properties of the system.

7.7.1 Common Architectural Patterns

According to Buschmann et al., "An architectural pattern expresses a fundamental, structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them." [15]. As is perhaps obvious from the definition of an architectural pattern, it represents the highest level of technical vocabulary in describing the system. To carry the metaphor further, the creative ability of the system is heavily influenced by this vocabulary, the elegance it provides, and the ideas that it expresses. In more prosaic terms, not only do the specific choice of components and their connectors make a difference to the various qualities of the system, but that their arrangement in specific ways, also influences these qualities. For example, a set of computers connected in a ring-based network exhibit a very different throughput from a set of computers connected in star-network. Although the computers present in the network are the same, the specific arrangement gives rise to a different communication protocol. This has implications on features like error-detection, error-correction, latency, throughput, etc.
In the domain of self-adaptive systems, the pattern of MAPE loop (monitor-analyze-plan-execute) interaction determines properties like speed of adaptation, scalability of system, communication and decision bottlenecks, etc. We now briefly talk about five different, commonly used Control-loop patterns in self-adaptive systems [29]:

1. **Hierarchical Control**: The overall system is controlled in a hierarchical manner with

complete MAPE loops at every level in the hierarchy. Generally, every level operates at different time-scales, with lower levels at shorter time-scales, and higher levels at longer time-scales.

2. **Master/Slave**: The master/slave pattern creates a hierarchical relationship between a master, that is responsible for analysis and planning, and slaves that are responsible for monitoring and execution. It is suitable for scenarios, where slaves are willing to share information, and adhere to centralized planning and decision-making. In case of large systems, this could lead to a bottleneck.

3. **Regional Planner**: In this pattern, instead of a single master, there are many *regional* hosts. Each regional host is responsible for several local hosts. The regional host is responsible for planning, while the local hosts monitor, analyze and execute, based on the plan. This pattern is a possible solution to the scalability problem with master/slave.

4. **Fully Decentralized**: In this pattern, each host implements its own complete MAPE loop. This leads to very flexible information sharing, as each M, A, P and E component on a host communicates with its corresponding part on another host. While flexible, achieving a globally consistent view, and a consensus on the next adaptation action is difficult. Achieving optimality is also correspondingly difficult, but not impossible (TCP adaptive flow control, for example).

5. **Information Sharing**: In this pattern, similar to the fully decentralized pattern, each host owns a local M, A, P, and E component. But, only the M component communicates with its peer components. There is no coordination on planning, analyzing or execution. Most often used in peer-to-peer systems, this sort of system is highly scalable, but difficult to coordinate.

We see in Table 7.5, that no single MAPE pattern meets all the criteria that we would like a system to have. The key point to be noted is that all of the above architectural patterns have benefits and liabilities. It is up to the application architect to balance these, with the

	Scalable	Achieving Optimality	Coordinated Communication	Flexibility
Hierarchical	—	✓	✓	—
Master/Slave	—	✓	✓	—
Regional Planner	✓	—	✓	—
Fully Decentralized	✓	—	—	✓
Information Sharing	✓	—	—	✓

Table 7.5: Properties of MAPE Patterns

demands of the application and the domain. This is rendered difficult by the fact that most big applications are usually a collection of architectural patterns, and not a single pattern. Not only should the architect keep in mind the individual properties of each pattern, but their interaction effects as well.

7.7.2 Architectural Patterns Employed in Clobmas

It is a fallacy to think that any of the patterns mentioned are, by themselves sufficient to form an entire application. Typically, a complex system is a combination of architectural patterns. Clobmas is no exception. We realize our system as a hybrid of two MAPE-loop styles: Regional Planner and Information Sharing. Each BuyerAgent acts as a regional planner for each AbstractService in the application. The adaptation from BizInt's perspective is done via the BuyerAgents, with the ApplicationAgent determining high level policies like whether to stop adapting or to publish an adaptation event. SkyCompute, on the other hand, uses a loose structure of Information Sharing agents, in the form of SellerAgents and MarketAgents.
On a lower level of abstraction of message-passing and interface definitions, it uses the Publish-Subscribe and Broker patterns. Both patterns allow for dynamic addition of components, and allow for work in a distributed fashion. Since the primary aim of Clobmas is to enable service-oriented applications to self-adapt, it is only fitting that Clobmas should itself embrace the service-oriented paradigm, and use the notion of services to achieve its functionality. Clobmas combines Publish-Subscribe and Broker, in a Market-Based environment, to allow applications to self-adapt. The Market-Based environment provides the stimulus, and the enabling infrastructure, for the service providers to sell their services, and for service

consumers to buy these services at a competitive price. The Market-Based mechanism is distributed, scalable, robust to failure of individual agents, and allows for properties like confidentiality of private information amongst agents. The market metaphor also maps naturally to the real-world business equivalents of service providers, service consumers, negotiators, registries, etc.

7.8 Discussion of Issues and Limitations

7.8.1 Threats to validity

Current implementations of public clouds focus on providing scaled-up and scaled-down computing power and storage. Our mechanism assumes 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 can be self-adaptive by changing the web-service that it utilizes.

> An Agile Service Network (ASN) comprises large numbers of long-running, highly dynamic complex end-to-end service interactions reflecting asynchronous message flows that typically transcend several organizations and span geographical locations. The term complex end-to-end service interaction signifies a succession of automated business processes, which are involved in joint inter-company business conversations and transactions across a federation of cooperating organizations [63].

Agile Service Networks embody the kind of structure that this mechanism would be applicable in. Agile Service Networks are formed, when organizations outsource a part of their function to other organizations. These may, in turn, outsource a part of their function to yet other organizations. Thus, an application or a business process being instantiated by one organization, uses the services of many organizations. This has several benefits in terms of efficiency, agility in response to business fluctuations, etc.

Our mechanism takes the ASN to a higher level by enabling automated scaling and adaptation based on Quality of Service attributes. However, the current state of the art in ASN is quite

low. A high degree of standardization is required for our mechanism to be applied, usefully.

7.8.2 Identity and Reputation

Clobmas does not attempt to rate or retain reputation information about any of the services being provided. A third-party service could advertise high levels of QoS, but fail to provide it. In such a case, the application would typically reject that service and search for a new one. However, it is easily conceivable that there are many such malicious agents in the marketplace, that provide a defective service. The presence of these kinds of services, decreases the utility of the marketplace and automated service composition. Dealing with malicious agents requires Clobmas to maintain reputation scores for each of the service providers being hosted by it. While this is easy when all the services are created by Clobmas itself, it becomes a lot more difficult when third-parties start providing services. Issues of trust (should it believe the application that complains against a service? how to deal with collusion?), identity (how does one prevent the same service from re-appearing with different names?) and centralization (how to keep track of services from other markets?) are all part of reputation and identity management. This is a research area, that Clobmas does not venture into.

7.8.3 Monitoring of QoS Levels

Regardless of whether a QoS violation is malicious, accidental or a result of bad engineering, it is essential to have indisputable evidence of the violation. For all the stakeholders to agree on incidents of QoS violations, we propose that a third-party mechanism be used to ensure provenance of QoS compliance or violation. There are many approaches to QoS monitoring, but they are still a topic of research. Therefore, this thesis does not propose a unique mechanism for monitoring, rather it assumes that a third-party will be involved in the system.

7.8.4 Marketplace Modelling

The case study makes pessimistic estimates of the distribution of QoS attributes amongst sellers. In the real world, a competitive market would drive out sellers that have an average distribution of QoS with high prices. The set of services used to model the supply is deliberately seeded with services that probably will not be able to transact. This is done with a view to evaluating the mechanism's performance in an extreme situation. In real life, these services would be competed out of the market, with most services differing in their QoS offerings by a marginal amount (or the price would be appropriately low, for low QoS). With a more typical scenario, our mechanism's performance would be much better, since the number of applications that are able to successfully achieve their QoS would be higher than in Figure 7.5. Also, the simulation does not model seller-side pricing strategies like loss-leader. Markets will also evolve to occupy niches. One niche could be to have very low entry barriers to trade in the form of zero registration and transaction fees. Another could be to only allow services of high-quality (with high reliability and availability) and reputation to trade, with high registration and transaction fees for the buyers.

As both sellers and markets adapt to changing market conditions, buying agents will have to increase in sophistication to improve their market-selection and bidding strategies.

Since the self-adaptive mechanism merely attempts to satisfy the application's QoS targets, it does not try to achieve optimal set of ConcreteServices, even if available. This is a tradeoff that occurs between simplicity and robustness on one hand, and optimality on the other.

7.9 Conclusion

The market mechanism does not achieve the optimum, neither from the individual perspective nor from the aggregate perspective. Given the limited computation that we demand of our agents, we aim only to achieve a satisficing result, i.e., find a solution that satisfies the QoS constraints of an application. Also, since the performance of the multiple-double-auction markets is better than that of a posted-offer, we show that our mechanism is better than the

currently used mechanism. We tested the mechanism for scalability on various axes, and find:

1. The time-cost of adaptation does not increase exponentially, in any case
2. The cost of decentralization is lesser than the cost of increasing the number of Abstract-Services in the Workflow, and the cost of increasing the number of CandidateServices.
3. However, the cost of increasing the number of QoS attributes per Application, increases the time-cost more than increasing the number of Markets.

An architect for BizInt should take these into consideration, when deciding on the various parameters on which the application will self-adapt. SkyCompute's architects should also take these into account, while deciding on whether to adopt all of Clobmas' adaptation axes, or only allow adaptation on certain chosen axes.

CHAPTER 8

Conclusion and Future Work

The real voyage of discovery consists not in seeking
new landscapes, but in having new eyes
Marcel Proust

As our journey ends, we now pause to reflect on the path that we took, and the road ahead. This thesis was a result of our quest to find a self-adapting mechanism, that would enable applications to change according to their circumstances. The scientific method consists of asking questions, and trying to systematically work towards the answer. Therefore, in the beginning of the thesis, we asked ourselves three questions:

1. In a society of service-based applications, each of which is self-adapting, what kind of a mechanism will allow satisfaction of QoS constraints?

2. How do we systematically measure the goodness of this mechanism?

3. How do we systematically measure the scalability of this mechanism?

We started our search with mechanisms that are currently used by cloud-providers to allow for applications to exhibit changing QoS. When we concluded that there weren't any, we looked at various mechanisms of self-adaptation that have been previously studied and, identified that we needed a decentralized mechanism. We also realized, that to fit these mechanisms into the context of the cloud, scalability of the mechanism needed to be an important property. To this end, we looked at different ways of service composition and their notions of scalability. When we were unsatisfied with the approaches taken in literature, we used **requirements engineering** to systematically identify the various axes of scalability, that a good mechanism would need to be measured against. We then proposed a multiple-auction based mechanism, to satisfy both adaptive applications as well as cloud-providers. Finally, we evaluated it against multiple market-conditions, tested it for scalability, and took a critical look at its architecture and design.

8.1 Summary

We now summarize the results of the research carried out, through this thesis. The major conclusions that we can draw from this research, are the following:

1. There is a pressing need for service selection mechanisms to systematically test for scalability

2. A market-based mechanism can be used as a pattern, for creating a decentralized, self-adaptive mechanism for service selection in the cloud

3. The *multiple double auction* method proposed in this thesis performs better than the currently-used posted-offer method for service selection.

4. Amongst the four axes on which we measured for scalability, the mechanism is most sensitive to number of QoS. That is, increasing the number of QoS per ConcreteService, increases the time taken to reach a satisfactory level of adaptation.

5. The mechanism is least sensitive to growth in number of CandidateService, as time taken to adapt increases linearly with growth.

6. The sensitivity of axes (in increasing order) is:

   (a) Number of CandidateServices

   (b) Number of Markets

   (c) Number of AbstractServices

   (d) Number of QoS per ConcreteService

8.2 Thesis Contributions

In proposing our self-adaptive mechanism, we brought together ideas from economics, requirements engineering, multiple criteria decision-making, and machine learning. We took these ideas and coalesced them into a mechanism. That journey of gathering ideas, filtering them for suitability, mixing, kneading and baking them, enabled a few contributions to the state-of-the-art in software engineering. We list them below, in the order they appear in the thesis.

1. **A review of cloud-based resource allocation**: We reviewed existing work on resource allocation in IaaS clouds. The review draws from existing approaches, and provides new insights that can assist the problem of QoS-aware dynamic selection for cloud-based applications, while minimizing their QoS violations.

2. **A systematic literature review of the current dynamic service composition mechanisms**: A systematic literature review (SLR) is an important step in mapping out the research landscape, and identifying gaps in current research. Through the SLR, we discovered that there are no generally agreed principles on which the scalability of dynamic service-selection methods is currently evaluated. We also recommended variables (and their range of values), that all dynamic service-selection methods should report on.

3. **Requirements Engineering for scalability analysis**: Building upon ideas from GORE, we systematically analyzed the needs of a dynamic service-selection mechanism, and how a multi-agent mechanism would solve it. In this way, we also identified the axes on which the scalability of service compositions methods should be evaluated, and scalability goals that they should aim for.

4. **Design and Architecture of Clobmas**: Borrowing from economics, we presented a multi-agent system based on multiple double auctions to enable decentralized self-adaptation. The local selection of Asks utilized ideas from multiple criteria decision making, and allowed local BuyerAgents to make intelligent decisions without consulting a central authority. The BuyerAgent's ability to change its bid-price, based on the prices in the market is based on ideas of gradient-descent in machine learning.

5. **Systematic scalability testing**: We tested Clobmas on all the axes that were identified through the requirements engineering process. We showed how to systematically evaluate a service selection mechanism in an empirical way, and objectively evaluate whether the scalability goals have been met, or not.

The process of science is to create a metaphorical map of the world, as we discover various paths through the forest of ignorance. The intent of science is therefore, to shine a light and

understand where it is advantageous to go, and where it is not. Therefore, it is important to not only highlight the advantages and benefits of a particular path, but also to identify the paths not taken.

8.3 Future Work

This thesis is a description of a path in the direction of scalable, decentralized self-adaptive systems. The path does not end with this thesis, and we can foresee several directions that research can take.

Short-term: In the immediate future, we foresee the creation of a middleware that can work with multiple clouds, and observe its behaviour with real-world applications. All simulations are approximations of reality, and an exposure to real-world demands of QoS change would be instructive.

Medium-term: Notions like reputation-management are critical to establish trust amongst services. We foresee that trust in multi-agent systems, specially ones that participate in human economies, will be an important aspect of such systems. How does a market ensure that the SellerAgent actually provides the services promised? How does the market ensure that bad services are kept out of the auction space? Also, investigation of other auction types like package auctions will provide more insight into mechanism design. We have not modelled complex seller-side behaviour. Specifically, actions like deliberate violation of QoS to free up resources or mis-reporting of QoS available, are both strategies that seller agents could indulge in.

Long-term: While it is difficult to analyze CDAs theoretically, it would still be of immense value to model the self-adaptive process in a formal way. This would allow better predictive

ability and a greater understanding of the strengths and limitations of the market-based self-adaptation. 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.

8.3.1 Impact of this Thesis

We believe that this thesis makes a novel, and important contribution to the state-of-the-art in service selection, for the SaaS cloud. The SaaS model of the cloud is in its infancy, and research of this kind is extremely important in helping it mature. We believe that multi-agent based mechanisms for self-adaptation, will be used in many more domains. This thesis provides a template for the creation of similar multi-agent systems, and their systematic evaluation.

Concluding Remarks:

Thus dear reader, we approach the end of the journey that we began many moons ago. It is our hope that this thesis, which is a snapshot of that journey, has been able to convey the essence of path we took, in a convincing manner.

※ ※ ※ ※ ※

References

[1] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient qos-aware service composition. *Proceedings of the 18th international conference on World wide web - WWW '09*, page 881, 2009. 42, 45, 46, 49

[2] Danilo Ardagna and Raffaela Mirandola. Per-flow optimal service selection for Web services based processes. *Journal of Systems and Software*, 83(8):1512–1523, August 2010. xi, 42, 46, 47, 48, 51

[3] Danilo Ardagna and Barbara Pernici. Adaptive Service Composition in Flexible Processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, June 2007. 46

[4] Michael Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, and Others. Above the clouds: A berkeley view of cloud computing. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009. 13

[5] A. Auyoung, B. Chun, A. Snoeren, and A. Vahdat. Resource allocation in federated distributed computing infrastructures. 2004. 20

[6] Shengli Bao and Peter R. Wurman. A comparison of two algorithms for multi-unit k-double auctions. pages 47–52, 2003. 67

[7] 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. 26

[8] B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for Web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004. 42

[9] K.G. Binmore. *Fun and games: a text on game theory*. D.C. Heath, 1992. 21

[10] Clint Boulton. *Forrester's Advice to CFOs: Embrace Cloud Computing to Cut Costs*. eWeek.com, October 2008. 12

[11] Clint Boulton. Gartner sees great saas enterprise app growth despite downturn. *eWeek.com*, October 2008. 12

[12] 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. 22, 88

[13] Lars Braubach, Alexander Pokahr, Daniel Moldt, and Winfried Lamersdorf. *Goal representation for BDI agent systems*, pages 44–65. ProMAS'04. Springer-Verlag, Berlin, Heidelberg, 2005. 109

[14] Ivan Breskovic, Christian Haas, Simon Caton, and Ivona Brandic. Towards self-awareness in cloud markets: A monitoring methodology. pages 81 –88, dec. 2011. 128

[15] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996. 152

[16] 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. 21, 22

[17] Eun-Kyu Byun, Yang-Suk Kee, Jin-Soo Kim, and Seungryoul Maeng. Cost optimized provisioning of elastic resources for application workflows. *Future Generation Computer Systems*, 27(8):1011 – 1026, 2011. 20

[18] Scott Camazine, Jean L. Deneubourg, Nigel R. Franks, James Sneyd, Guy Theraulaz, and Eric Bonabeau. *Self-Organization in Biological Systems*. Princeton University Press, 2001. 53

[19] 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. 51

[20] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. QoS-aware replanning of composite Web services. *IEEE International Conference on Web Services (ICWS'05)*, pages 121–129 vol.1, 2005. 51

[21] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, and Raffaela Mirandola. A Framework for Optimal Service Selection in Broker-Based Architectures with Multiple QoS Classes. *2006 IEEE Services Computing Workshops*, pages 105–112, September 2006. 47

[22] Jorge Cardoso, Amit Sheth, and John Miller. Workflow quality of service. Technical report, University of Georgia, Athens, Georgia, USA, March 2002. 80, 116

[23] 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. xiii, 49, 81

[24] Zhi-yong Chen and Qing Yao. A Framework for QoS-aware Web Service Composition in Pervasive Computing Environments. *2008 Third International Conference on Pervasive Computing and Applications*, pages 1011–1016, October 2008. 42, 44

[25] Scott H. Clearwater, Rick Costanza, Mike Dixon, and Brian Schroeder. Saving energy using market-based control. pages 253–273, 1996. 62

[26] D Cliff. Simple bargaining agents for decentralized market-based control. *HP Laboratories Technical Report*, 1998. 62, 96

[27] CloudHarmony. Do slas really matter? 1-year Case Study. *http://blog.cloudharmony.com/2011/01/do-slas-really-matter-1-year-case-study.html*. 129

[28] Transaction Processing Performance Council. Tpc, tpc-w benchmark. Online. 19

[29] Rogério deLemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Software engineering for self-adaptive systems: A second research roadmap. *Workshop on Software Engineering for Self-Adaptive Systems, Dagstuhl*, May 2011. 152

[30] Thomas G. Dietterich. Machine learning for sequential data: A review. pages 15–30, 2002. 19

[31] 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. 41, 103

[32] 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. 67, 92

[33] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. *2008 Grid Computing Environments Workshop*, pages 1–10, November 2008. 12

[34] Daniel Freidman. The double auction market institution: A survey. 1993. 62

[35] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. Sharp: an architecture for secure resource peering. *SIGOPS Oper. Syst. Rev.*, 37:133–148, October 2003. 20

[36] Aiqiang Gao, Dongqing Yang, Shiwei Tang, and Ming Zhang. Web service composition using integer programming-based models. *IEEE International Conference on e-Business Engineering (ICEBE'05)*, pages 603–606, 2005. 42, 43

[37] S. Gjerstad and J. Dickhaut. Price formation in double auctions. *E-Commerce Agents*, pages 106–134, 2001. 113

[38] 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. 66, 68, 77

[39] Qing Gu and Patricia Lago. Exploring service-oriented system engineering challenges: a systematic literature review. *Service Oriented Computing and Applications*, 3(3):171–188, July 2009. 31, 49

[40] Alok Gupta and DO Stahl. The economics of network management. *Communications of the ACM*, 42(9):57–63, 1999. 62

[41] Kieran Harty and David Cheriton. A market approach to operating system memory allocation. pages 126–155, 1996. 62

[42] Minghua He and Nicholas R. Jennings. Southamptontac: An adaptive autonomous trading agent. *ACM Trans. Internet Technol.*, 3:218–235, August 2003. 62

[43] 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. 62, 64

[44] M.D. Hill. What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4):21, 1990. 41

[45] Y. C. Ho, L. Servi, and R. Suri. A class of center-free resource allocation algorithms. *Large Scale Systems*, 1:51, 1980. 62

[46] Pu Huang, Alan Scheller-Wolf, and Katia Sycara. A strategy-proof multiunit double auction mechanism. pages 166–167, 2002. 67

[47] IBM. An architectural blueprint for autonomic computing. June 2006. 55

[48] IBM. The IBM perspective on cloud computing, 2008. 12

[49] Amazon Inc. *Amazon Elastic Compute Cloud (Amazon EC2)*. Amazon Inc., http://aws.amazon.com/ec2/#pricing, 2008. 16

[50] David Irwin, Jeffrey Chase, Laura Grit, Aydan Yumerefendi, David Becker, and Kenneth G. Yocum. Sharing networked resources with brokered leases. pages 18–18, 2006. 20

[51] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155 – 162, 2012. 18

[52] Nick Jennings. Automated haggling: building artificial negotiators. pages 1–1, 2000. 62

[53] Mohd. Shahadatullah Khan. *Quality Adaptation in a Multisession Multimedia System: Model, Algorithms and Architecture.* PhD thesis, University of Victoria, 1998. 44

[54] Shahadat Khan, K F Li, E G Manning, and M D M Akbar. Solving the knapsack problem for adaptive multimedia systems. *Stud Inform Univ*, 2(1):157–178, 2002. 44

[55] Barbara A. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. *Evidence-Based Software Engineering*, 2(EBSE 2007-001), 2007. 27, 36

[56] Barbara A. Kitchenham, T. Dyba, and M.a Jorgensen. Evidence-based software engineering. *Proceedings. 26th International Conference on Software Engineering*, pages 273–281, 2004. 30, 31

[57] Paul Klemperer. Auction theory: A guide to the literature. *JOURNAL OF ECONOMIC SURVEYS*, 13(3), 1999. 66

[58] J Ko, C Kim, and I Kwon. Quality-of-service oriented web service composition algorithm and planning architecture. *Journal of Systems and Software*, 81(11):2079–2090, 2008. xi, 42, 45

[59] Jeff Kramer and Jeff Magee. A Rigorous Architectural Approach to Adaptive Software Engineering. *Journal of Computer Science and Technology*, 24(2):183–188, April 2009. 53

[60] Vijay Krishna. *Auction Theory*. Academic Press, 1 edition, March 2002. 64

[61] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1:169–182, August 2005. 20

[62] Jing Li, Yongwang Zhao, Jiawen Ren, and Dianfu Ma. Towards adaptive web services QoS prediction. In *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8. IEEE, December 2010. 42, 46

[63] Benedikt Liegener. Agile service network, July 2011. 155

[64] Richard Lipsey. *An Introduction to Positive Economics*. Weidenfeld and Nicolson, London, United Kingdom, fourth edition, 1975. 97

[65] Robert E. Marks. Breeding hybrid strategies: optimal behavior for oligopolists. *Journal of Evolutionary Economics*, 2:17–38, 1992. 68

[66] Andreu Mas-Colell, Michael Dennis Whinston, and Jerry R. Green. *Microeconomic Theory*. Oxford University Press, illustrated edition, 1995. 63

[67] 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. 129

[68] 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. 98

[69] Amit Nathani, Sanjay Chaudhary, and Gaurav Somani. Policy based resource allocation in iaas cloud. *Future Generation Computer Systems*, 28(1):94 – 103, 2012. xi, 16, 17, 20

[70] J. Nicolaisen, V. Petrov, and L. Tesfatsion. Market power and efficiency in a computational electricity market with discriminatory double-auction pricing. *Evolutionary Computation, IEEE Transactions on*, 5(5):504 –523, oct 2001. 68

[71] 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. 62

[72] 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. 62

[73] Jinzhong Niu, Kai Cai, Simon Parsons, and Elizabeth Sklar. Reducing price fluctuation in continuous double auctions through pricing policy and shout improvement. pages 1143–1150, 2006. 67

[74] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. pages 124–131, 2009. 16

[75] University of Chicago. Haizea. http://haizea.cs.uchicago.edu/. 17

[76] University of Chicago. The Nimbus Project: An open-source EC2-compatible iaas cloud. http://www.nimbusproject.org/. 16

[77] Peyman Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE*, 14(3):54–62, 1999. 56

[78] Manish Parashar and Salim Hariri. Autonomic computing: An overview. *Unconventional Programming Paradigms*, pages 257–269, 2005. 56, 57

[79] David C. Parkes. Iterative combinatorial auctions: Achieving economic and computational efficiency. May 2001. 72

[80] Simon Parsons and Mark Klein. Towards robust multi-agent systems: Handling communication exceptions in double auctions. pages 1482–1483, 2004. 67

[81] Dewayne E Perry, Adam A Porter, and Lawrence G Votta. Empirical studies of software engineering: a roadmap. *Proceedings of the conference on The future of Software engineering ICSE 00*, pages 345–355, 2000. 49

[82] S. Phelps. Evolutionary mechanism design. *University of Liverpool, Diss*, (July), 2007. 120

[83] Alvin E. Roth and Ido 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. 62, 113

[84] Yuko Sakurai and Makoto Yokoo. An average-case budget-non-negative double auction protocol. pages 104–111, 2002. 67

[85] Yuko Sakurai and Makoto Yokoo. A false-name-proof double auction protocol for arbitrary evaluation values. pages 329–336, 2003. 67

[86] Adam Smith. An inquiry into the nature and causes of the wealth of nations. 1776. 53

[87] Borja Sotomayor, Rubén S. Montero, Ignacio M. Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13:14–22, September 2009. 16

[88] Perdita Stevens and Rob Pooley. *Using UML: Software Engineering with Objects and Components*. Object Technology Series. Addison-Wesley, second edition, 2006. 114

[89] 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. 62

[90] Dawei Sun, Guiran Chang, C Wang, Yu Xiong, and Xingwei Wang. Efficient Nash equilibrium based cloud resource allocation by using a continuous double auction. *Computer Design and Applications ICCDA 2010 International Conference on*, 1(Iccda):V1–94, 2010. 21

[91] Gerald Tesauro and Rajarshi Das. High-performance bidding agents for the continuous double auction. pages 206–209, 2001. 67

[92] Leigh Tesfatsion. Agent-based computational economics: Growing economies from the bottom up. *Artificial Life*, 8:55–82, 2002. 67, 68

[93] Paul Tucker and Francine Berman. On market mechanisms as a software technique. *Science*, pages 1–38, 1996. 62

[94] Axel van Lamsweerde. *Formal Methods for Software Architectures*, chapter From System Goals to Software Architecture. Springer-Verlag, 2003. 102, 103

[95] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng.*, 26:978–1005, October 2000. 105

[96] Christian Vecchiola, Rodrigo N. Calheiros, Dileban Karunamoorthy, and Rajkumar Buyya. Deadline-driven provisioning of resources for scientific applications in hybrid clouds with aneka. *Future Generation Comp. Syst.*, 28(1):58–65, 2012. 18

[97] Perukrishnen Vytelingum. *The Structure and Behaviour of the Continuous Double Auction*. PhD thesis, 2006. 67, 78, 113

[98] Carl 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. 62

[99] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: flexible proportional-share resource management. page 1, 1994. 62

[100] M.P. Wellman. Market-oriented programming: Some early lessons. *Market-based control: a paradigm for distributed resource allocation*, pages 74–95, 1996. 63

[101] Danny Weyns. *Architecture-Based Design of Multi-Agent Systems*. Number XVII. Springer, first edition, 2010. 109

[102] Bernard Widrow and Ted Hoff. Least mean squares adaptive filters. 2003. 96

[103] Tom De Wolf and Tom Holvoet. A catalogue of decentralised coordination mechanisms for designing self-organising emergent applications. pages 40–61, 2006. 57

[104] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. SLA-Based Resource Allocation for Software as a Service Provider (SaaS) in Cloud Computing Environments. *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 195–204, May 2011. 21, 22

[105] Peter R. Wurman. A Parametrization of the Auction Design Space. *Games and Economic Behavior*, 35(1-2):304–338, April 2001. 62

[106] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. The michigan internet auctionbot: a configurable auction server for human and software agents. pages 301–308, 1998. 64

[107] Yan Yang, Shengqun Tang, Youwei Xu, Wentao Zhang, and Lina Fang. An Approach to QoS-aware Service Selection in Dynamic Web Service Composition. *International Conference on Networking and Services (ICNS '07)*, pages 18–18, June 2007. 42, 43, 48

[108] Y.O. Yazir, C. Matthews, R. Farahbod, S. Neville, A. Guitouni, S. Ganti, and Y. Coady. Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis. pages 91 –98, july 2010. 22

[109] Paul K. Yoon, Ching-Lai Hwang, and Kwangsun Yoon. *Multiple Attribute Decision Making: An Introduction (Quantitative Applications in the Social Sciences)*. Sage Publication Inc., 1995. 42

[110] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 1(1):6–32, May 2007. 27, 42, 44, 45

[111] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12:317–370, July 2003. 64, 109

[112] Liangzhao Zeng, Hui Lei, and Henry Chang. Monitoring the qos for web services. pages 132–144, 2007. 98

Appendices

APPENDIX A

Mechanism Design Appendices

A.1 A Segue into PROMETHEE

In the field of decision-making, where the alternatives have multiple dimensions, there are three kinds of methods that aid in making a decision:

1. **Aggregation Methods**: These involve using utility functions, for each of the dimensions and performing calculations on each, to choose the best. In our situation, while the ApplicationAgent is well-placed to calculate utility from a certain set of services, the BuyerAgent has only local knowledge. This makes the creation of utility functions difficult. Also, utility functions suffer from the problem of having to be re-created every time, a particular dimension changes.

2. **Interactive Methods**: Since our mechanism works with agents, and we wish to automate as much as possible, interactive methods of decision-making are not suitable.

3. **Outranking Methods**: These methods rely on the construction of an outranking relation between attributes, and then use this to eliminate possibilities. One of the advantages of an outranking method is that it provides a clear, easily-understood rationale for why a particular choice is better than another.

PROMETHEE is a multiple criteria decision-making procedure that provides an outranking mechanism to choose amongst alternatives, especially when each alternative can be compared on multiple criterion. PROMETHEE performs a pairwise comparison of the options, over each criterion. The preference for one option is formulated as a function, which takes into account the difference between the two options, over each criterion. Each criterion is represented as a pair $(c_i, P_i(a,b))$, where c_i is the criterion, and $P_i(a,b)$ represents the preference of option a over option b. The preference function $P(a,b)$ can be defined as:

$$P(a,b) = \begin{cases} 0 & \text{if } f(a) \leq f(b), \\ p[f(a), f(b)] & \text{if } f(a) > f(b) \end{cases} \tag{A.1}$$

Now, $p(\cdot)$ is chosen to be of the following type:

$$p[f(a), f(b)] = p[f(a) - f(b)] \tag{A.2}$$

Since there might be areas in the option space, where the decision-maker is indifferent to option a or option b, we need to denote the magnitude of the difference, between the two options on a certain criterion, c.

$$x = f(a) - f(b) \tag{A.3}$$

This difference can then be graphically represented using the function $H(x)$, such that:

$$H(x) = \begin{cases} P(a,b), & \text{if } x \geq 0, \\ P(b,a) & \text{if } x < 0 \end{cases} \tag{A.4}$$

PROMETHEE defines the following six types of criteria:

1. *Usual Criterion:* In this case, there is indifference between a and b only when $f(a) = f(b)$. As soon as these values are different, the decision-maker has a definite preference for one of the options (figure A.1a).

2. *Quasi-Criterion:* In this case, there is indifference between a and b so long as the difference between $f(a)$ and $f(b)$ does not exceed k (figure A.1b).

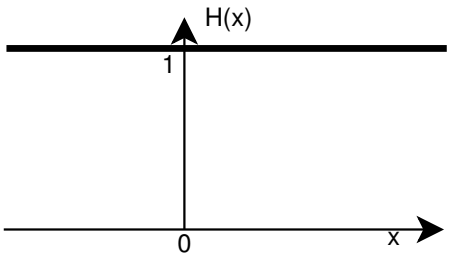
3. *Criterion with Linear Preference:* This criterion allows the decision-maker to progressively prefer option a over option b, for progressive deviations between $f(a)$ and $f(b)$. The intensity of the preference increases linearly until the deviation equals m. After this, the preference is strict (figure A.1c).

4. *Level Criterion:* In this case, the decision-maker is indifferent to a and b, until the deviation between $f(a)$ between $f(b)$ remains less than q. When the deviation is between q and $p+q$, the preference between one and the other is weak $(1/2)$, after which the preference becomes strict (figure A.1d).

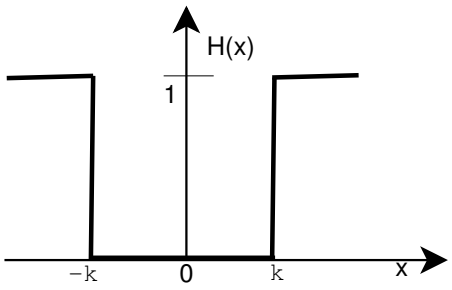
5. *Criterion with Linear Preference and Indifference Area:* Here, the decision-maker is indifferent to a and b, as long as $f(a) - f(b)$ does not exceed s. Above s, the preference increases progressively until $s+r$, beyond which there is again a strict preference (figure A.1e).

6. *Gaussian Criterion:* In this case, the preference of the decision-maker grows in a gaussian manner, with x (figure A.1f).

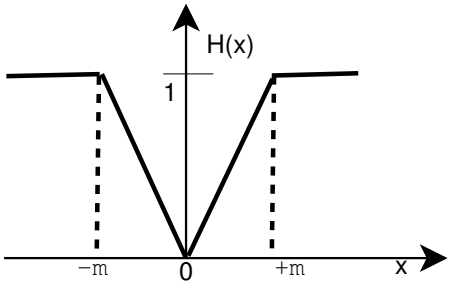
Graphically, the criteria can be shown in figure A.1



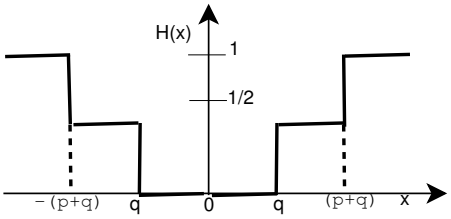
(a) Usual Criterion



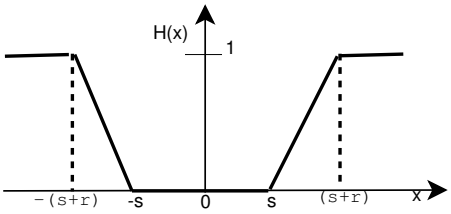
(b) Quasi-Criterion



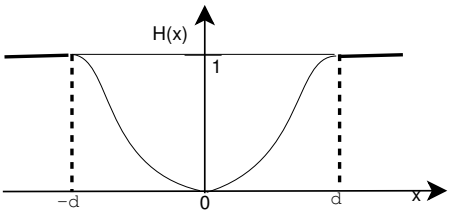
(c) Criterion with Linear Preference



(d) Level-Criterion



(e) Criterion with Linear Preference and Indifference Area



(f) Gaussian Criterion

Figure A.1: Six criteria defined by PROMETHEE

APPENDIX B

Requirements and Design Appendices

B.1 KAOS

B.1.1 KAOS Concepts and Terminology

Goals, agents and operations : A *goal* is a prescriptive statement of intent about some system (under construction or already developed). Satisfaction of all the goals of the system is the intent of architecting it. Satisfaction of a goal requires the cooperation of some of the agents in the system. An *agent* is a human, device, legacy software or software component, that plays a role in the system. The word *system* refers to the software-under-consideration **and** its environment. The environment is described using *domain properties*. These are descriptive statements, that affect the functioning of the software, generally by adding constraints. Typically, they refer to organizational policy, physical laws, etc. An agent is responsible for carrying out operations that affect the state of some objects in the system. An *object* is an entity, association, event or another agent, that is characterized by attributes and domain properties. An *operation* is thus, an input-output relation over an object. The state of a system is the aggregation of the states of all its objects.
Realizing a goal is the responsibility of an agent. However, a goal may not be realizable by the assigned agent for various reasons:

1. The goal is too high-level to easily judge whether it is realized or not
2. The goal refers to variables that are not monitored or controlled by the agent

To alleviate this problem, we perform *goal refinement*. *Goal refinement* refers to breaking down higher-level strategic goals into lower-level sub-goals. This is done by identifying sub-goals that, either independently or in coordination with other sub-goals, achieve the higher-level goal. In KAOS, this is done using AND/OR *refinement-abstraction structures*. An

AND refinement links a goal to a set of sub-goals, that are together defined to be sufficient for satisfying the higher-level goal. An OR refinement links a goal to a set of alternative sub-goals, any of which is defined to be sufficient for satisfying the higher-level goal. In figure B.1, we see a simple example of an *AND refinement* of a goal called `Meeting Scheduled Correctly`. Goals are specified using semi-formal words like `Achieve, Maintain, Avoid`. In figure B.1, both the sub-goals (`[Participants Constraints Known]` and `[Meeting Notified]`) must be satisfied for the higher-level goal to be achieved.

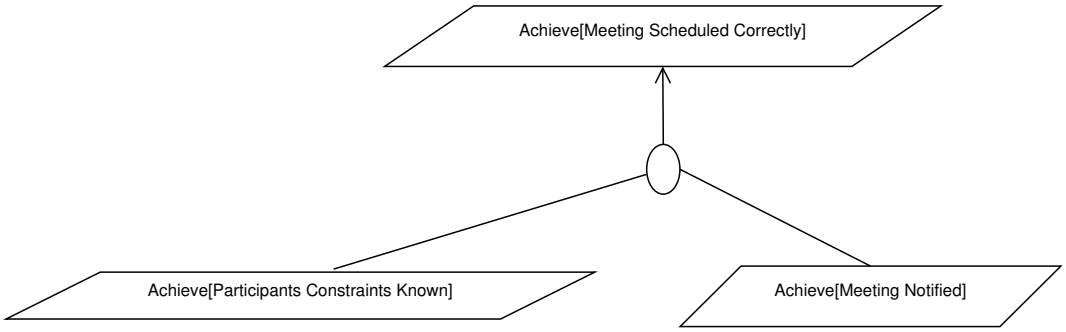


Figure B.1: A simple AND refinement

8.1.2 KAOS-based goals for our Multi-Agent System

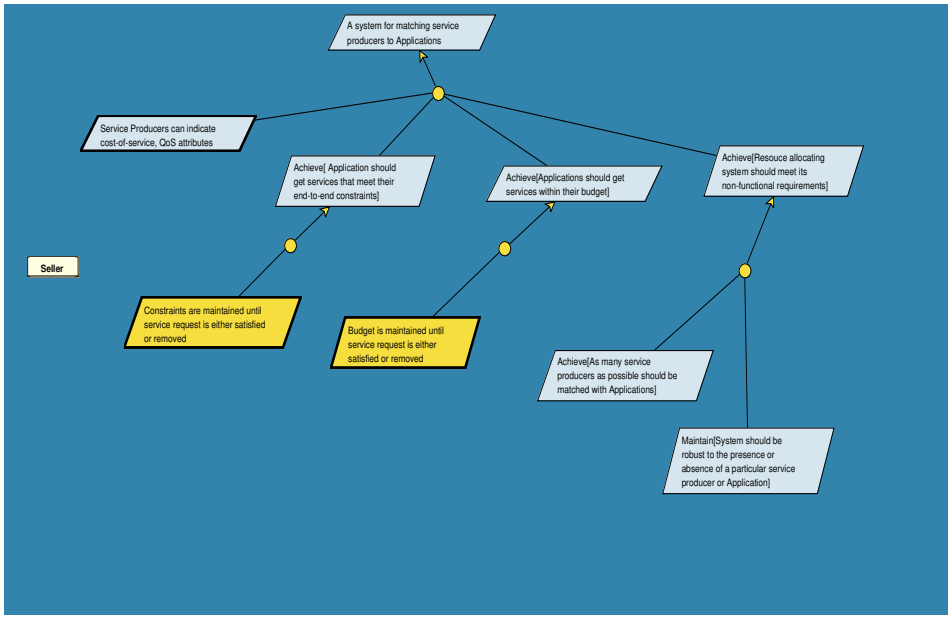


Figure 8.2: Goals for a multi-agent system in the cloud

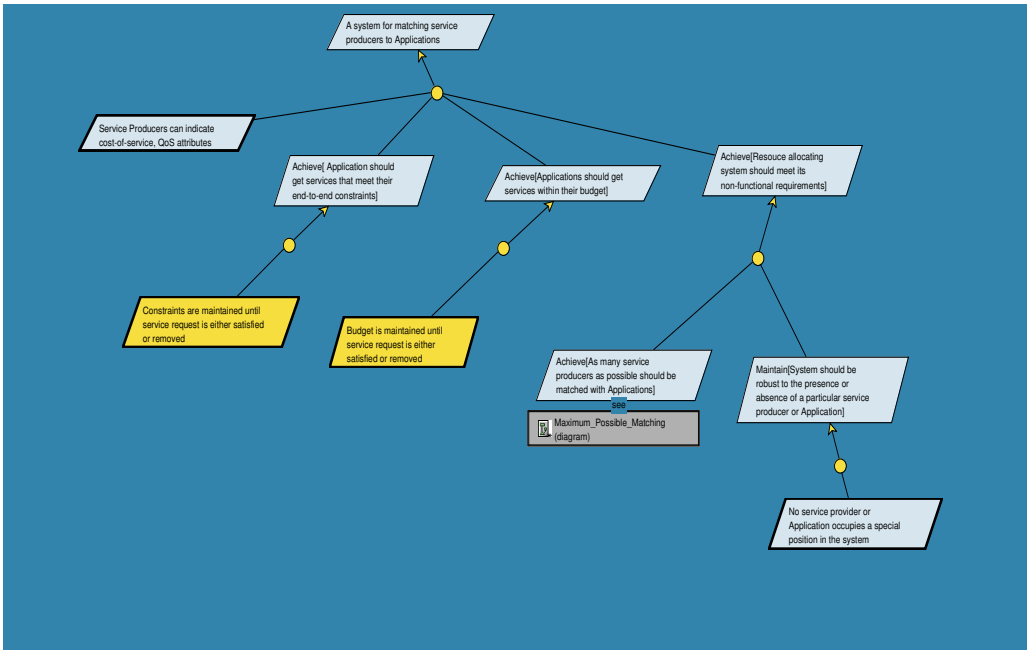


Figure 8.3: Top-level goals from the service consumer's perspective