Social Grid Agents

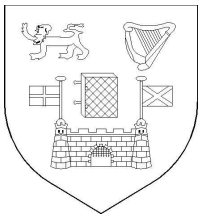
by

Gabriele Pierantoni

A Thesis submitted to
The University of Dublin
for the degree of

Doctor of Philosophy

Department of Computer Science,
University of Dublin,
Trinity College



September, 2008

Declaration

This thesis has not been submitted as an exercise for a degree at any other University. Except where otherwise stated, the work described herein has been carried out by the author alone. This thesis may be borrowed or copied upon request with the permission of the Librarian, University of Dublin, Trinity College. The copyright belongs jointly to the University of Dublin and Gabriele Pierantoni.

Signature of Author .....

Gabriele Pierantoni September, 2008

Abstract

The problem of resource allocation in Grid computing has been actively tackled by the scientific community for some years; its complexity is in meeting the expectations of different actors with different concepts of optimality within an environment divided into geographically dispersed and different administrative domains and thus unsuited to supporting centralized management systems. The complexity of the problem is further increased by the fact that the scientific community produced different Grids with different standards, focuses and technologies.

This problem closely resembles that of an economy and has consequently been investigated with economic paradigms, leading to models that are often tailored to one or more specific situation.

This thesis speculates that an approach reflecting the social structures that support economic models could allow different models to co-exist as they do in the real world; this thesis also speculates that a successful solution should be able to encompass different Grids both present and future.

In order to achieve this, the modelling of the different actors and their behaviour is inspired by some behaviours that are common in societies; to encompass different grids, the environment of the actors is modelled as a *metagrid*: a conceptual space divided into three regions: the different existing middlewares, the metagrid region and border regions at the intersection of the two where interoperability issues are tackled by a combination of *abstraction* and *translation*.

This thesis proposes that resource allocation be modelled as the intersection of different topologies where actors interact with each other to enforce different allocation philosophies. These actors are modelled with agents termed *Social Grid Agents*.

The proposed model is based on four different kinds of topologies: a *Production Topology* where services are performed, a *Social Topology* where agents that control the production process engage in social and economic processes, a *Control Topology* where the agents belonging to the Social Topology control those belonging to the Production Topology, and a *Value and Price Topology* that connects social and production agents to determine the value and price of a resource.

Social Grid Agents communicate with messages and their behaviour is defined by sets of policies that determine how they relate to each other. The need for an efficient and expressible native language with which the agents express policies and messages led to the decision to use a functional language as a base for the agent's native language.

An architecture is defined at an abstract and concrete level and a prototype that can encompass different social relationships has been implemented and tested in a series of experiments to evaluate the efficiency, scalability and behaviour of the Social Grid Agents.

Acknowledgements

I would like to dedicate this thesis to "no frills" flying which allowed me to pursue this research in Trinity College Dublin while still being able to bask (at times) under the plentiful light of my native sky becoming what, for long, I longed for: a European citizen. But, much more importantly and much more gratefully, I dedicate this work to Gaia that, in these years, flew "without frills" over and over and over with the loving and rocky stubbornness of her people to join me and gaze at the stars under Dobbblin's cloudless sky.

The person I own more for this research is Dr.Brian Coghlan, my supervisor, whom I thank most sincerely not only for being always of help and guidance in the often hazy routes of this research but especially for demonstrating me that truly admirable work ethics are not just abstract ideas but can be found in real people. His support and help, always generously offered even at the latest hours and from the remotest corners of this little, blue planet, has been a balsm for a cynical and mildly disillusioned Italian such as myself.

I would like to thank sincerely John Walsh who desperately tried (unfortunately often failing) to teach me the very basics of system administration and always volunteered his time to help me in both practical and theoretical matters. But I especially want to thank John for two more things: for being a friend in these years and for his most peculiar musical inclinations which strengthened my tolerance and acceptance of diversity.

I want to thank Dr.Eamonn Kenny, the research manager, for his help in mathematics and for the unpredictable paths of our conversations that, oscillating between graph theory to theology, rendered much less boring waiting at airport gates. I want to thank Dr.Stephen Childs, Dr.David O Callaghan, Dr.John Ryan and Dr.Geoffrey Quigley for their help in setting up the experimental testbed. I want to thank Dr.Keith Rochford for the time, always profitable and entertaining, in which we carried out the joint part of this research. I want to thank all the present and past people of the Computer Architecture Group: Kathryn Cassidy, Dr. Soha Maad, Peter Lavin, Simon O'Neil, Dr. Stuart Kenny, Stephan Dudzinky, Ronan Watson and Oliver Lyttleton who where always kind to me, helped whenever was possible and even went through the pains of throwing me a heartening surprise party.

I want to thank my family that had the wiseness (or madness) to support me practically and emotionally in going back to be a student when I had a stable job.

I want to thank HEA, ISF and the Irish Government for funding this research. I finally want to thank Trinity College Dublin and all the people that made it alive for these years that, despite the despicable weather and the soaring price of a pint, have been some of the most pleasant of my life.

Contents

List of Figures

List of Tables

List of Acronyms

CE	Computing Element
CPU	Central Processor Unit
EERM	Economic Enhanced Resource Manager
GA	Grid Agent
GMD	Grid Market Directory
GRAM	Grid Resource Allocation Manager
GT4	Globus Toolkit 4
JVM	Java Virtual Machine
OGSA	Open Grid Service Architecture
OGSI	Open Grid Service Infrastructure
OO	Object Oriented
OOA	Object Oriented Architecture
JVM	Java Virtual Machine
PGA	Production Grid Agent
RB	Resource Broker
RMS	Resource Management System
RUS	Resource Usage Service
SE	Storage Element
SGA	Social Grid Agent
SLA	Service Level Agreeement
SOAP	Simple Object Access Protocol
SOA	Service Oriented Architecture
WMS	Workload Management System

Chapter 1

Introduction

Figure 1-1: In the beginning, It's hard to see all the bumps.

The secret of a good sermon is to have a good beginning and a good ending, then having the two as close together as possible.

George Burns

1.1 Foreword

One of the first scientific essays I have read describes in its foreword the sense of awe and, at times, even despair that the author felt in discovering the limitless vastitude of its field of enquiry: a cubic millimeter of a turtle's retina, a truly minuscule entity. Nevertheless, the investigation of that microscopic cube unravelled a universe of unforeseen complexity, a treacherous ford that the author managed at the very end to cross, but just.

Only now can I really understand those words. This enquiry does not even have a cubic millimeter of wonderful, menacing matter to stand upon, nevertheless, traversing its maze was difficult.

This voyage starts from the simple wish to improve resource allocation in Grid infrastructures, sets sail to the distant shores of social and economic behaviour and returns at last to the concrete implementation with a still slightly blurred idea of how to make things a bit better.

Is this not, after all, the purpose of every journey ?

1.2 Grids, Stakeholders and Societies

Grid Computing [39, 38] is a rapidly developing field of research both in the academia and the industry. Although there are many different definitions of Grid Computing ranging from very technical to more descriptive, as a first approximation, we can say that Grids are a way to decouple the administrative domains of the resource and the jobs, or, more precisely: Grids are a solution that allows resource owners and job owners to belong to different administrative domains. This apparently simple decoupling opens great freedoms and, concurrently, great challenges. The process is similar in nature to the evolution of societies from little, isolated self-sufficient

structures to more complex and interconnected ones.

If users and resources are under the same administrative domain, security requirements and allocation mechanisms can be defined and enforced directly by the administrator. This arrangement has its strengths in its simplicity but its weakness resides in the fact that the resources of the group must suffice for all possible demands of its members. Using again a social example, we can model a department of a university (or a company) under a single administrative domain as a self-sufficient tribe which has to provide all the needed resources; these are shared among their members under the rule of an chieftain. The chieftain of the tribe is analogous to the system administrator (or whoever the system administrator takes orders from) and the resources are not food or shelter but digital resources such as computational time or storage space.

A self-sufficient system can either acquire vast amounts of resources (if it can at all) to cope with all foreseen peaks in demand or else accept that some requirements may not be fulfilled (possibly even threatening survival). An obvious way to overcome such limitations is to share resources among different groups allowing a more stable distribution. One way to do this is by trading resources. Trade allows acquisition of resources but also allows specialization. If actors can trade, they need not bother any more to provide themselves all the possible goods and services because they can be found elsewhere. Trading actors can now concentrate at doing what they are best at and trade for all the rest.

In this model a Grid is a trading and sharing platform among tribes. As a matter of fact, Grids define *"Virtual Organizations"* (that encompass different administrative domains) where resources can be shared under a common security framework. Within a Virtual Organization (often referred to as a VO for the sake of brevity), resources are usually distributed following arrangements among the administrators of the various domains. This might be modelled as a form of centralized trade based on the negotiation of the administrators of the different societies, i.e. an alliance of societies where the various leaders agree on sharing mechanisms under the aegis of a trade organization. This form of trade allows more flexibility, resilience and

specialization.

Farmers might produce corn and exchange it with fish caught by those living on the shores; fish and corn might be exchanged with game hunted by the tribes of the hills. The department of Astrophysics might have to bother no longer with cumbersome tasks of system administration which might be dealt with by the experts of the Computer Science group. The leaders of the Astrophysics and the Computer Scientists might meet and arrange compensation for the Computer Scientists in exchange for the provided services.

In the simplest case, in a Grid there are three main kind of stakeholders: those who own the resources, those who want to use them and those who are in control of the Virtual Organization that encompasses the first two. The stakeholders owning resources are usually named *Resource Owners*, the stakeholders that use the resources are usually named *Clients* or *Users* while the last are usually named *VO Managers*. All these stakeholders (or actors) have different and, at times, conflicting goals. In most cases and in absence of other constraints, the clients will be mainly interested in having their jobs run in the fastest possible time, the resource owners will try to have their resources idle for the least possible time and the VO Managers will try to satisfy the requirements of both the resource and job owners that belong to the Virtual Organization. This consideration leads to a first important distinction between the *functionalities* that are performed and the *policies* with which they are executed. A request of the execution of a job on a grid specifies as functionalities the executable, inputs, outputs and other parameters while it may specify as policies additional constraints to define time and/or budget for the execution.

Grids traditionally support three different topologies for the scheduling systems that are in charge for the distribution of computational resources. There are *centralized*, *decentralized* and *hierarchical* schedulers. Each of these solutions are particularly suited to fulfill the requirements of users or resource owners. In a centralized scheduling system, all execution requests are submitted to a single service that, in turn, forwards the jobs to the resource that suits best its selection algorithm. In this case, the preferences of the users to have their jobs fulfilled are easy to meet. In a

de-centralized scheduling system, the execution requests are submitted to different services, each controlled by a resource owner, these systems are best suited to fulfill the preferences of the resource's users. By combining different systems in multi-layered architectures, it is possible to create hierarchical structures that allow the resource owners to define local allocation policies and the users to avail themselves of the advantages of a centralized system.

The different scheduling approaches reflect the fact that having multiple actors negotiating the allocation and sharing of resources poses a major problem: any one of these actors will have its own (quite often selfish) idea of *"how"* resources should be shared, a concept of *"optimality"* that is frequently clashing with that of others.

This is the very heart of an economy in its broader understanding: *"The administration of the concerns and resources of any community or establishment with a view to orderly conduct and productiveness"*[1], or better still: *"The science which studies human behaviour as a relationship between ends and scarce means which have alternative uses"* [2].

This view of the resource allocation problem in Grid Computing is nothing new. The scientific community already investigated and still investigates many ways in which economics could be used to optimize and regulate the way actors with different ideas of "optimality" share their resources. Economy can be used to describe the theoretical basis of sharing mechanisms and can also be used to investigate how effective a sharing mechanism is. In recent years the scientific community produced both theoretical investigations on the best economic models for the Grid problem and also real systems that use economic principles for resource allocation.

In this thesis I speculate that this economic approach, while profitable and invaluable, has a shortcoming. Most of these economically inspired systems are tailored to a particular economic perspective: some have centralized planners, others rely entirely on free markets, others use economically driven optimization for scheduling. I believe that, as it is in the real world, there is no "optimal" economic model able to encompass efficiently all the bewildering variety of relations and allocation schemes

[1] Oxford English Dictionary - http://dictionary.oed.com/cgi/entry/50071995?

[2] Lionel Robbins, *An Essay on the Nature and Significance of Economic Science*

of a Grid. It is my belief that a high-level allocation system in Grid computing should allow a variety of allocation philosophies to co-exist and allow transactions that range from competitive to cooperative, from selfish to empathic.

As a matter of fact, the *exchange* and *sharing* of computational resources already encompass many different economic philosophies. From charitable empathy-based computation platforms to business-oriented distributed computing (often referred to as *Cloud Computing*); one may argue that the full spectrum and complexity of society and economics is already present in Grid Computing.

The second argument is slightly more subtle. I argue that economies are a product of societies and that they could not exist at all without all the other characteristics of a society. I speculate that economically based solutions will eventually have to face this truth and "open" themselves both to a larger variety of allocation philosophies and build stronger links to all the other aspects of Grid computing such as user, provider, charitable and miscreant behaviours that constitute the "other tiles" of the Grid social canvas. In fact, a system capable of encompassing different allocation philosophies must be capable of selecting the type of allocation philosophy depending to its social relations.

For these reasons I define a *Social Paradigm* to analyze the problem of resource allocation and as an abstract theoretical ground to design prototypical solutions capable of testing, in the experimental sense, different sharing philosophies in different Grid environments.

To tackle the problem posed by the existence of different middlewares I define the environment as a *metagrid*, a conceptual space encompassing different middlewares. A metagrid space is divided into three different regions.

- *Grid and Workflow regions* that host different existing middlewares
- *The metagrid region* where social and economic interactions take place
- *Border regions* at the intersection of the two where interoperability issues are tackled by a combination of *abstraction* and *translation*.

In this, this thesis is a beginning and a work in progress.

1.3 Description of this work

The structure of this thesis reflects the path from these abstract concepts to a proposed prototypical implementation and the experiments thereon.

Chapter 1, *Introduction*, introduces the basic concepts and the aims of this research. It also describes the architecture of the thesis and summarizes its achievements and contributions.

Chapter 2, *A look at the landscape*, briefly overviews the work of the scientific community relevant to this research: the concept of resource allocation in Grid computing from an abstract and concrete perspective and the different approaches to the problem of interoperability. The chapter introduces the concept of *Social Grid Agents*, an agent-based system for high-level brokerage in Grid computing and it explains how this paradigm relates to economic concepts such as *production* and *exchange* and to social concepts such as *selfishness* and *empathy*.

Chapter 3, *Methods*, is devoted to the description of the methods followed in this thesis. Firstly it details the social and economic paradigm, then it describes the different topologies in which the agents can be arranged and how different social and economic models can be described in such a way. The problem of interoperability is tackled in a section devoted to the concept of *metagrid* a candidate environment for SGAs. The architecture of the agents is described in a dedicated section and, finally, a section explains the information flows that regulate the behaviour of the agents and how a functional language, *ClassAd*, is used for this purpose.

Chapter 4, *Use Cases and Implementation*, details the implementation of the Social Grid Agents prototype from different perspectives: the implemented topologies, the implemented agents and an example of policies. The chapter is concluded with a section devoted to experiments that were performed to assess the reliability, scalability and correctness of the agent's behaviour.

Chapter 5, *Conclusions and Future Work*, concludes this enquiry and briefly explores possible future research directions.

Chapters 3 and 4 are divided into sections that cover the *topologies* in which the agents are arranged, the *architecture* of the agents and the nature of the *language*

used by the agents both internally and externally. The two chapters describe each of these topics with an increasing level of detail starting from an abstract perspective to implementation examples.

1.4 Main Contributions

The main aim of this research is a resource allocation system for Grid computing capable of encompassing different allocation philosophies across different middlewares. To design a system capable of encompassing different allocation philosophies I used a social paradigm to try to mimic, in the Grid, some of the social behaviours that, in the real world, allow to humans to act according to different economic philosophies that depend on their social context.

To design a system capable of harnessing different middlewares I used a concept of a *metagrid* that embraces three different domains: the existing middlewares, border regions where agents act as intermediators and a native metagrid region where agents that belong to different topologies, engage in social and economic relations. The need to deal with both functionalities and policies in a distinct albeit closely intertwined fashion, the impossibility to know at design time which agent will be making a final decision and the infeasibility of the assumption that all the information will be available to all actors suggested to base the native language of the agents on a functional language to allow delayed evaluation. A functional language widely used in Grid computing called *ClassAd* was selected as a basis to define a native language that agents use both internally and to communicate among each other. The prototype allowed to test this solution in an environments encompassing three middlewares and to experiment with agents capable of enforcing different, co-existing allocation mechanisms.

The main contribution of Social Grid Agents can be summarized in three broad, different areas

- Social Grid Agents can be seen as *economy-agnostic* (and this, in fact, has been one of my strongest beliefs in these years) and offer a system capable of

implementing different, and possibly co-existing, allocation mechanisms that can range from cooperation to competition, from unregulated economic-based systems to centralized planning. This agnostic view of the allocation problem allows Social Grid Agents to offers the possibility to instruct agents to expose different behaviours depending on their social context.

- Thanks to the metagrid approach, Social Grid Agents, are capable of harnessing different middlewares; the definition of border regions where interoperability issues that regard both functionalities and policies are tackled by a combination of translation and abstraction performed by border agents allows for the design of stable and flexible interoperability infrastructure

- The use of a functional language such as ClassAd as a basis for the agent's language allows the definition of functionalities and policies that can be evaluated in different, potentially unknown a priori, steps and this is a useful feature as it allows the definition of policies that can be enforced in a structure that is not known a priori.

- Finally, the architecture allows the implementation of agents whose behaviour can be easily defined and modified by writing behavioural instructions based on the same functional approach and language.

Chapter 2

A look at the landscape

Figure 2-1: Hoy, hoy, hoy ! Digital land ahead !

It is not from the benevolence of the butcher, the brewer, or the baker that we expect our dinner, but from their regard to their self-love, and never talk to them of our own necessities but of their advantages

Adam Smith

If you knew what I know about the power of giving, you would not let a single meal pass without sharing it in some way

Buddha

The reason that the invisible hand often seems invisible is that it is often not there

Joseph E. Stiglitz

The real voyage of discovery consists not in seeking new landscapes but in having new eyes.

Marcel Proust

2.1 Introduction

This chapter overviews the related work on the topics investigated in this thesis: the problem of resource allocation and interoperability in Grid computing followed by the motivations and the beliefs that underline this research.

2.2 Resource Allocation Mechanisms

The problem of resource allocation in Grid computing has been actively investigated by the scientific community and consequently the literature regarding this issue is both extremely vast and rich in content.

In its simplest form, this problem can be described as the dilemma faced by an actor looking for the resources it needs in an environment under a given set of constraints. In a Grid environment the agent looking for resources (often referred to as a *Client*, *User*, or *Requester*) uses different services (either directly offered by one or more Grid middlewares) to discover the available resources (thus exploring its environment), connect to them and eventually use them. The resources encompass hardware resources, access to simple Grid services (such as job submission and data

storage), software services (optionally accessed through licenses) and more complex services such as workflow engines. The environment comprises one or more Grid middlewares and in future might be expected to include a plethora of additional services (indexes, markets and banks) that should eventually connect the user to the resources it needs.

This problem is further complicated by the scarcity of resources (and the consequent need of their optimal use) and by the need to meet the user's satisfaction. This leads to different and often clashing optimization concepts that have to co-exist in the same environment.

Although the various solutions[29],[40] envisaged by the scientific community to tackle the complexities of resource brokerage and allocation in Grids encompass genetic algorithms, simulated annealing and hybrid solutions[20], we focus this survey on classical resource allocation approaches as well as on those that are economy-based, ranging from competitive liberalism to cooperation, from uncontrolled markets[29][85] to centralized controlled economies[42][35].

The first part of this literature survey encompasses some *"classic"* allocation systems; the second, some based on *economic paradigms*.

2.2.1 Classical Resource Allocation Mechanisms

A proposal for a taxonomy of resource allocation systems in Grid Computing is described in [49]. This paper concisely describes issues, context and functions of resource allocation systems and offers a taxonomy based on the concepts of grid type, namespace organization, resource dissemination protocols, resource discovery, scheduling model, state estimation and scheduling policy.

Another taxonomy is proposed in [52]. This analysis is based on the definition of an abstract model for a *Resource Management System* (RMS) and also uses actors that cover the roles of resource requesters, resource providers and resource controllers. RMSs are categorized along different dimensions:

- *Machine Organization*: the topology in which the various actors are arranged. The following categories are proposed: flat, cells and hierarchical

- *Resources*: how the RMS is related to the resources it manages. This dimension is further decomposed in the following sub-dimensions:
  - *Resource Model*: that can be either based on *Schemas* (where the description of the resource is expressed in a description language) or *Object Models* (where the description of the resource also comprises the actions that can be performed on the resource), both of which can be either fixed or extensible.
  - *Organization of the Resource's namespace*: that can be relational, hierarchical or graph-based.
  - *Quality of Service*: that can be either be absent, *soft* (where attributes of the request can be expressed but not enforced directly by the RMS) and *hard* (where the attributes of the request are guaranteed to be met).
  - *Organization of the store for the resource information*: that can either be implemented through *Network Directories* (such as LDAP [18]) or *Distributed Objects* (such as CORBA [6]).
  - *Resource discovery*: that is performed either through queries or agents.
  - *Resource dissemination*: that can be either periodic or on demand.
- *Scheduling philosophy*: that can be centralized, hierarchical or distributed.
- *State estimation*: that can be either predictive or non predictive.

gLite

In the gLite middleware, the action of matching jobs and resources is performed by the *Workload Management System* (WMS) [23, 27] where the information on the available resources is matched against the job description.

The gLite middleware encompasses two main types of resources: *Computational Elements* (CEs) [27] and *Storage Elements* (SEs) [27]. The middleware allows a form of combined matching where it is possible for a user to specify a preference for the Storage Element that is most close to a defined Computational Element. In gLite the

ClassAd language [80, 79, 81, 56] is used to define a *Job Description Language* named *JDL* [68, 69] that is employed to specify both the requirements of the job owners and the preferences of the resource owners. The description of the jobs and those of the resources are matched in a component of the Workload Management System called the *Resource Broker* [27, 43].

While the description of the jobs is directly determined by the user, the description of the resources and their status is the result of a gathering and filtering process performed by a multi-layered hierarchical architecture of elements known as *BDII* [27]. Mutual trust mechanisms between those responsible for the resources and those responsible for the various BDII servers minimize the likelihood of malicious tampering with the information.

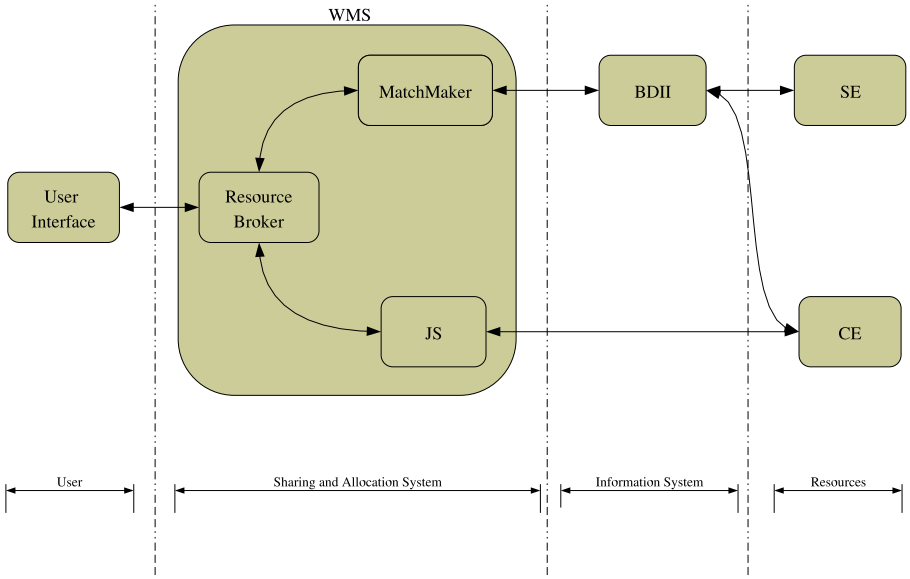


Figure 2-2: A simplified view of the resource allocation architecture in the gLite middleware.

To access resources users must possess a valid *User Certificate* [27] issued by a *Certificate Authority* [27] that is accepted by the resource providers. With this certificate a user can issue a *Proxy Certificate* [27] that will be accepted by the resources that are enabled to do so.

The user can define the requirements of its job as a Boolean function (known as *Requirements*) that must evaluate to *true* for the resource allocation to be successful.

The user can also specify a *Rank* function to rate different resources. The same can be done by the resource's owners. Hence gLite can choose resources and jobs that respect the requirements (expressed with the *Requirements* function) and the preferences (expressed with the *Rank* functions) of both the resource and job owners.

The user can query the gLite services for a list of available resources through a *matchmaking* operation and then target specifically that resource by specifying it in the requirements field or it can ask the WMS system to find and use an "optimal" resource by invoking a one-step *submit* operation.

It is worth noting that the Requirements function defines a *sub-space* in the *space* of the parameters where the requirements of both the user and the resources are met while the Rank function defines an *ordering* in the parameter space.

gLite also allows a certain degree of a site-level, VO-based *fair-share scheduling* as those responsible for the schedulers can implement fair-share policies on systems such as MAUI [15].

Fair-share scheduling [35, 42] entails that all users belonging to the VOs encompassed in a Grid can receive the services that were agreed upon when they joined the Grid. Although simple in principle this concept is difficult to implement because of the unpredictable and volatile nature of the workloads submitted by the users. An a priori allocation will fail in presence of heavy concurrent requests of resources. Let's imagine an allocation system where all users are granted the right to consume a given amount of computation over a period of time, if all users concentrate their requests in the same time frame, then it is possible, if not probable at all, that a system will not be able to cope with the demand peak and, consequently, the users will not be able to avail themselves of the share of resources they were granted. The implementation of fair-share policies is further complicated in Grids by the autonomy of the resource owners in the definition and enforcement of local allocation policies.

In gLite, MAUI implements a fair-share policy by setting the amount of *waiting time* that is then propagated through the BDII hierarchy to the WMS.

gLite's approach to resource allocation gives the user a significant latitude: on one hand, the description of the job may be minimal, omitting any resource spec-

ification (thus leaving the WMS to take into account only the requirements of the resource owners). On the other hand the job description may contain very detailed requirements and even target a specific resource.

The taxonomy of [52] describes the gLite middleware with the following characteristics: a hierarchical architecture, a hierarchical namespace with an extensible schema model, a network directory store, no explicit QoS support, discovery via distributed queries, periodical push dissemination of the information, and a hierarchical scheduler with extensible policies.

Globus

The Globus GRAM [91] (part of the Globus Toolkit [92]) has a different approach to resource allocation: it offers a sophisticated level of abstraction capable of harnessing different schedulers with an integrated staging system for the job sandboxes[1] but it offers no direct brokerage facility.

Jobs are described with the RSL language [93] where the requirements are directly expressed as key-value pairs for parameters. This is a significant difference from the gLite approach where the parameters are indirectly defined as the sub-space where the requirements function evaluates to *true*.

The Globus approach explicitly defines a point in the space of the parameters:

$$p_1 = v_1 \tag{2.1}$$

$$p_n = v_n \tag{2.2}$$

$$p_N = v_N \tag{2.3}$$

The gLite approach is to define a subspace:

$$Requirements = f(p_1, p_n, p_N) \tag{2.4}$$

[1]While the gLite system requires that all the sandbox files be present on the User Interface machine, the Globus GRAM allows for the files to be staged automatically. The locations of the source and destination are directly specified in the job description.

The two solutions coincide if the gLite Requirements is composed of equivalences, such as:

$$Requirements \quad = \quad (p_1 == v_1)\&\&(p_n == v_n)\&\&(p_N == v_N) \tag{2.5}$$

The taxonomy of [52] describes the Globus middleware with the following characteristics: a cell architecture, a hierarchical namespace with an extensible schema model, a network directory store, soft QoS support, periodical *push* dissemination of the information and a decentralized scheduling infrastructure where the schedulers are provided externally.

Condor

The *Condor* middleware has an approach that is similar to that of gLite. In fact the ClassAd language used by gLite was originally developed for Condor. One difference between the Condor and gLite middleware is the way in which the information is gathered and published by the resources and the users. While the gLite middleware delegates both the steps of matching and binding the requester and the provider to the WMS component, the Condor architecture is more flexible, allowing the two steps to be separated. This is achieved by specifying different protocols.

A simple allocation scenario in a Condor system is described in Figure 2-3. Firstly the *Provider* and the *Requestor advertise* the description of requests and resources to the *Matchmaker*. This communication is regulated by a specific *advertisement protocol*. Once the matchmaker has found a compatible pair of ClassAd advertisements it communicates the match to the requestor and the provider with the *notification protocol*. Finally, the requestor and the provider may engage in the final binding step through a third protocol known as the *binding protocol*.

Because of this flexible architecture and the functional nature of the ClassAd language that allows partial evaluations, this solution can be extended to cope with situations where requests can only be satisfied by bundles of resources such as CPUs and licenses. This extension to the match-making architecture, called *gang-matching*, is represented in Figure 2-4 and is the focus of a Ph.D thesis [81].

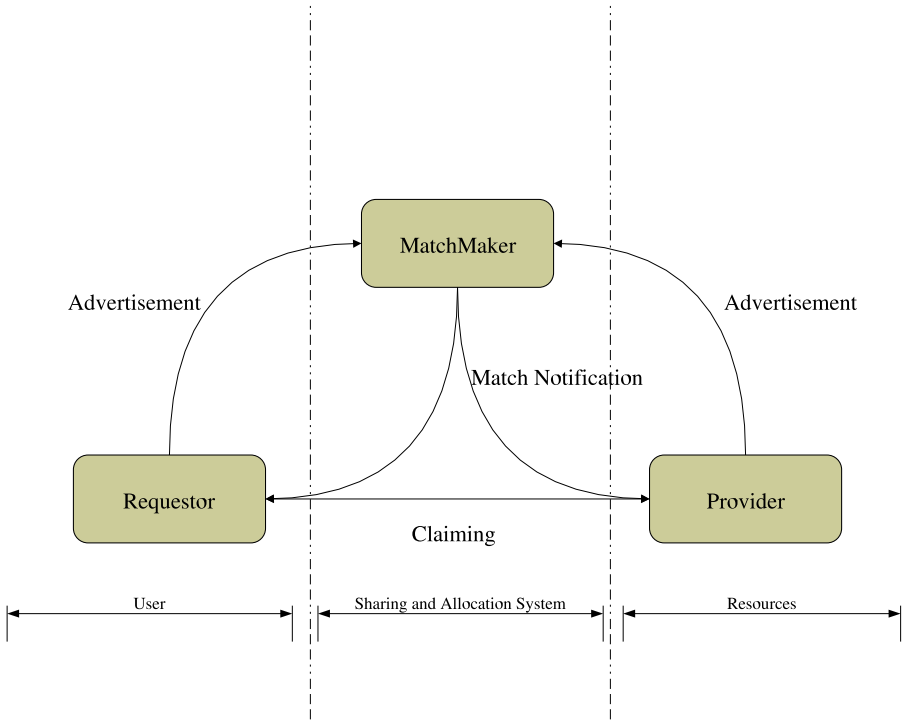


Figure 2-3: Simple Resource Allocation architecture in the Condor middleware.

The taxonomy of [52] describes the Condor middleware with the following characteristics: a flat architecture, a hybrid namespace with an extensible schema model, a network directory store, no QoS support, discovery via distributed queries, periodic push dissemination of the information and a centralized scheduler.

2.2.2 Socially and Economically Inspired Resource Allocation Systems

Chapter 1 introduced the concept that the problem of resource allocation in Grid computing closely resembles the definition of an economy. In fact economic-inspired solutions both at theoretical and practical level are actively investigates and proposed by the Grid community.

Although not specifically focused on Grids, Sharpe's *The Economics of Computers* [84] offers a comprehensive application of the *General Equilibrium Theory* [61] to the domain of computers. The book was written in 1972 and, consequently, it deals with a scenario that has some significant differences from that of these days;

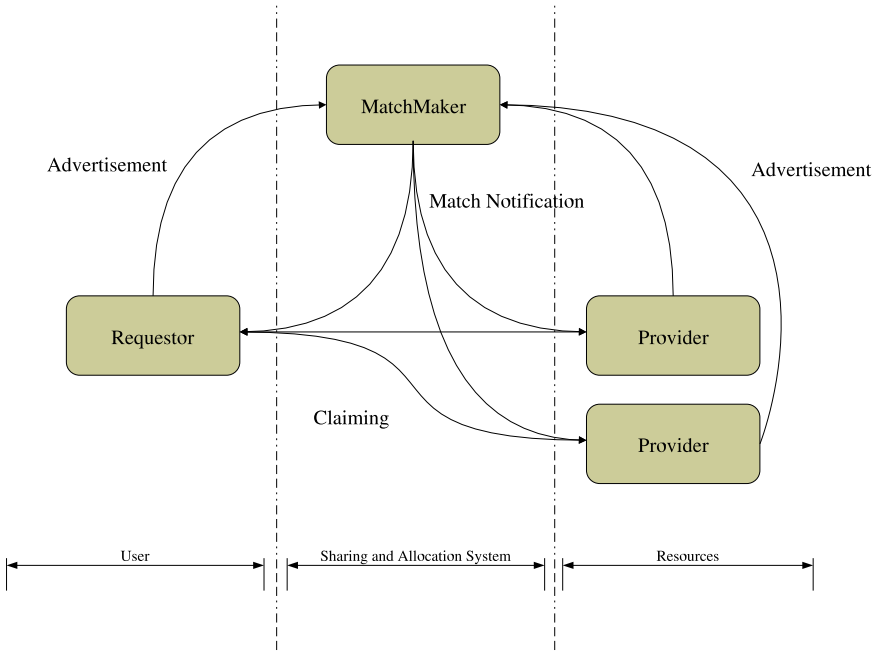


Figure 2-4: Multiple Resource Allocation architecture in the Condor middleware.

nevertheless it offers a theoretical and practical approach to understanding the economics of computers and, although written in a period in which computers were large mainframes, the discussion on pricing models could be extended to the modern scenarios.

In [96] and [97] there is an attempt to analyze the problem of economic-based resources (only cpu and disk space are taken in account) allocation in Grid computing (at times referred to as *Computational Economies*) both at a theoretical and practical level through simulations. A set of assumptions to use Computational Economics are defined and two main pricing mechanisms: *Commodity Market* and *Auctions* are analyzed in different market conditions. The conclusion of the authors is that, although auction-based systems are widely used (the authors suggest that this is due to the relative simplicity of the implementation of auction-based systems), the commodity market models performs better in most market conditions.

Although most of the economic-based solutions and analysis [28], [29], [33] and [36] of the problem of resource allocation in Grid computing rely, implicitly or explicitly to the believes that the General Equilibrium Theory is useful to describe real life economics and that, by extension, it is useful to tackle similar problems in Grid com-

puting, the study conducted in [63] contradicts these believes on different counts. The main problems raised by the authors are of different kinds: first General Equilibrium Theory does not offer any explanation for cases in which the markets fail, secondly they highlight the differences between real and computational markets that may result in an even higher probability of failure, lastly the authors point out that General Equilibrium Theory does not take into account aspects such as the role of government policies, the necessity of trust between the agents that are relevant in Grid computing.

[28] and [29] survey the market models that could be possibly be applied to Grid computing. These are deemed to be:

- *Commodity Market Model*: In such a model the price of a resource is set directly and solely by the owner of the resource. This model encompasses different strategies and algorithms for the definition of the price that can reflect the amount of resources used (that can be frequently adjusted in time to reflect demand and supply or that can be flat and thus remain constant for a period of time).

- *Posted Price Model*: This model is similar to the Commodity Market with the extension of special offers that a resource provider can post to attract new costumers. Both models are characterized by the fact that the price is set by only one party: the resource owner. In the real world this model is found in supermarkets where the price of each item is defined by the management of the supermarket and remains constant over a reasonable length of time.

- *Bargaining Model*: In this model the price is defined by a negotiation of two parties: the resource owner and the party requiring the resource (e.g. a user or a broker). The two parties engage in a negotiation based on their object functions. If the negotiation protocol is effective then the resulting price is a compromise of the expectations of both parties. In the real world such a model can be found in a bazaar where the price is defined on a *per purchase* basis by the seller and the client.

- *Tendering/Contract-Net Model*: In this model the party requesting a resource publishes its requirements and collects offers from the resource owners. It can then rank the offers and choose the one that is most suitable. This model is used in the real world as a way to find and choose contractors.

- *Auction Model*: The auction model defines a one-to-many negotiation where a party offers resources to many possible buyers. The price that is set during the auction depends on the utility functions of the different actors involved but also depends heavily on the *auction rules* that are defined and enforced by a third party: the *auctioneer*. The rules define the auction protocols. Among the most common auction models are:

  - *English Auction*: where all buyers can out-bid each other until only one offers remains.
  - *First price sealed bid*: where all buyers submit their offer just once, without knowing those of the others, and the highest offer is selected.
  - *Vickery*: where all buyers submit their offer just once without knowing those of the others; the winner is the bidder who offered the highest price but the price is the one offered by the second highest bidder.
  - *Dutch Auction*: where a high starting price is set by the auctioneer, who lowers the price until one of the bidders accepts it.

- *Community and Coalition Models*: In such models the resources are shared among the members of a community from a common pool.

- *Bid-based proportional sharing models*: In such models the percentage of a resource that each requester can get is a proportion of the price it offered during the bidding process.

- *Monopoly, Oligopoly*: Where the price is set by one or few resource owners that can define prices without competition.

DGAS

gLite includes a distributed accounting system called DGAS [21] that is based on a network of accounting units that keep information on groups of users and resources. Classic records (CPU time and memory) are managed by the system, which also supports a limited form of economic accounting for economic brokering.

DGAS implements Resource Usage Service (RUS) [74, 94], which is an OGF [17] initiative to define an accounting system capable of encompassing different middlewares (gLite using DGAS, SweGrid using SGAS [35, 42], and Unicore[77]) to achieve accounting interoperability.

An extensive survey of accounting in Grid Computing can be found in [70].

SweGrid and SGAS

Globus can be extended with economy-based resources allocation using the *SGAS* [35, 42] system developed at the Department of Computing Science and HPC2N of Umea University. The SGAS system focuses on interoperability and fair-share allocation. Interoperability is achieved by offering plugin-points for adapters specific to different environments.

SGAS is somehow inspired by a *socialist* perspective where great importance is attributed to the fairness of the allocation. To achieve this, SGAS implements a centralized economic model where a central authority grants different amounts of credits to the various users of the system.

This solution allows a fair allocation of resources (provided that the central authority is fair) in the sense that all users may be given a fair amount of credits; unfortunately, in the absence of a correcting mechanism, there is no guarantee that the credits can be really converted into services.

SGAS successfully tackles this problem by time-stamping the allocation credits and by reducing their validity to a certain time-frame. This discourages clients from hoarding credits and the possibility that they are used all at once.

A key component of SGAS is a highly-scalable banking system. This is achieved by virtualizing the bank and by partitioning it across different servers (named branch

servers).

SORMA

The SORMA project [19, 64, 65] attempts to adopt self-organization. First it postulates three different reasons that prevent effective implementation of market mechanisms [19]: insufficiencies in the design of Grid markets, poor client support and inadequate connection with existing middleware. The project proposes an economic model called *Decentralized Local Greedy Mechanism* inspired by the neo-liberist belief that a population of agents acting in the pursuit of their exclusive self-interest will eventually reach an equilibrium of economic efficiency that no single agent planned for at the beginning.

The model is further enriched in [78] to take SLA agreements into account. An *Economic Enhanced Resource Manager* (EERM) uses a predictive model to foresee the impacts on the resources it manages of accepting a certain task. The model uses the concepts of *Revenue* (the amount of credits that are to be paid if SLA terms are met), *Penalty* (the amount of credits that are to be paid if SLA terms are not met) and *Gain* (the difference between Revenue and Penalty). The EERM will decide whether to violate or not SLA terms, basing its decisions on the foreseen effects on resources and the financial outcome.

SORMA proposes an architecture where a Grid application will connect to a set of *intelligent tools*. These tools will use the economic knowledge provided by the *Open Grid Market* (contracts management, billing, security and monitoring) to optimize the Grid resources offered by an *Economic Grid Middleware* that connects the intelligent tools with the existing resources and enriches the information with economic-specific data.

SORMA implements an economic model that adjusts the amount to be paid according to the effective Quality of Service received by the user. In fact, each time a job is rescheduled, an amount of credits is paid to compensate for the jobs that are delayed.

The connections of SORMA's approach with the concepts of *Service Oriented*

Architecture and *Autonomic Computing* link this project philosophy with that of CATNETS discussed in 2.2.2.

ASSESS Grid

The ASSESS Grid project also focuses on solving the issues that arise in the implementation of SLAs in Grids. In [2] the shortcomings of SLAs (unwillingness of providers to agree on SLAs knowing that a penalty will have to be paid in case of violation and the uncertainty of users on the real enforcement of SLAs) is mitigated and taken into account in the decision mechanism with the concept of the *Probability of Failure* (PoF) that describes the likelihood of a SLA to be violated. This information is published with the SLAs and allows for more complex and effective decision-making mechanisms by all the parties involved: users, resource owners and resource brokers.

CATNETS

CATNETS [4, 37] offers a theoretical analysis and simulation experiments of a market model based on the concept of a self-organizing population of agents. This approach is tightly linked to the concept of Service Oriented Architectures (SOA).

The agents used in the simulation are:

- *Resource Agents* that act as proxies for aggregation of Grid resources.
- *Basic Service Agents* that provide the *Complex Service Agents* with the basic service they need.
- *Complex Service Agents* that provide unspecialized entry points to Grids (they accept any request for a Complex Service) that compose the basic services into ones of higher complexity.

CATNETS implements a two-step economic model whereby a call-for-proposals is followed by a negotiation.

The simulation also encompasses two different types of markets for resources and services. This approach allows the evaluation of a free market allocation topology where no central broker exists.

GESA

The book *"A Networking approach to Grid Computing"* [57] devotes a chapter to the description of an OGSA-based architecture that supports Grid economic services: the Grid Economic Services Architecture (GESA). The proposal is not tailored to any specific economic model but offers an OGSA-compliant architecture to support different economic models and to build effective, standardized economic-based applications in Grids. The architecture is harmonized with the architectural proposals of several OGF [8] working groups on Resource Usage Services, Grid Resource Allocation Agreement Protocol and Usage Record.

GESA proposes a set of standardized metadata built on top of the OGSA Grid Service interface that describe *Service Data Elements* (service-specific advertising elements) that enrich the description of a Grid service with economic information such as pricing, currency and payment methods.

GESA architecture is flexible and comprehensive, it describes resources as well as complex payment issues such as compensation, refunding and support for multiple currencies.

The GridBus Project

The GridBus [12] project is the continuation of the *Grid Economy Project* [9] that produced an economic-aware infrastructure sitting on top of existing middlewares.

The overall architecture is composed of five different layers

- A *Grid application layer* that hosts the software applications that are executed on the GridBus middleware.
- A *User-level middleware layer* that hosts tools such as workflow engines and economy-aware brokers used by the applications to discover, negotiate, select and manage resources.

- A *Core Grid middleware layer* that hosts existing Grid middlewares such as Globus or Unicore alongside economy-specific services such as markets and banks.

- A *Grid-Fabric Software layer* that hosts Operating Systems (OSs), schedulers such as Condor [5] and the native Libra [14], and other technologies such as the Java Virtual Machine (JVM).

- A *Grid-Fabric Hardware layer* that hosts physical resources.

Native, economy-aware components are present in all layers except the bottom (Grid-Fabric Hardware) and top (Grid application layer). These are:

- *Nimrod-G*[16] that allows the economy-aware management of the execution of parameter-sweep applications across distributed computational resources.

- *An economy-aware broker*[11] that allows the discovery and selection of resources on the basis of *budget constraint* and *time-constraint* policies.

- *A Grid Market Directory (GMD)*[10] that allows the providers to publish their services and the related costs to the public. It permits the retrieval of information through its portal, a SOAP interface or specific APIs.

- *A Grid bank* that allows accounting, payments and authentication.

- *Libra*[14] that allows quality-of-service-aware and economic-aware scheduling, execution and monitoring of sequential and parallel jobs on a homogeneous Linux cluster. The job description is enriched by information regarding the allocated budget and the deadline required. This additional information allows the Libra scheduler to implement economy-driven time and budget optimization algorithms whilst respecting the QoS requested by the user.

The architecture of GridBus allows for the implementation of economic models that are mainly QoS, budget and time driven at every level from low-level scheduling with Libra to high level interactions with markets, banks and workflow engines.

It is worth noticing that, although the current implementation highlights time and budget constrained allocation modalities, its paradigm and the philosophy underlying its architecture is inspired by a broader analysis of many different economic models, see [28] and [29].

ArguGRID

Although it relies on Peer to Peer computing-oriented resources rather that Grid middleware, the ArguGrid [1, 90] (funded by the European Commission under Framework Programme 6) proposes an interesting agent-based approach. Its design is inspired by Service Oriented Architecture and is based on a set of agents capable of *argumentation* to cope with incomplete knowledge and conflicting information for decision-making.

The proposed architecture contains five layers: the topmost consists of user applications that connect with dynamically formed Virtual Organizations of agents hosted in the fourth level. The third level contains middleware and web-services to connect to the second level, which contains the P2P computing networks. The level at the bottom hosts the network.

The most innovative aspect of ArguGrid is in the fourth level where intelligent agents use argumentation to negotiate with each other to produce contracts that describe task allocations and workflow descriptions.

A Trust-aware Economic-based Scheduling Model

The relation between trust and economic models is analyzed in [98] in the domain of economic models for scheduling. The relevance of the problem of trust in the sharing of resources is highlighted and a formal model is proposed to allow trust issues to be taken into account in economic models. A mathematical abstraction of trust is defined as a function of the relation between two parties and the *reputation* that the party has among its peers. This mathematical figure is taken into account in a cost optimization algorithm that considers constraints in time and budget. Theoretical analysis and simulations show how the additional information regarding trust improves efficiency.

Price Elasticity

[55] speculates that time-division based pricing mechanisms, where resources are priced depending on the average use through the day, are not used by the Grid users to their full advantage. To overcome this an economic model is proposed that affects both the behaviour of the user and the provider by introducing a function based on the *price-elasticity* concept that is used to adjust resource prices. An architecture based on OGSA is then proposed, where components hosted on the user and on the provider adjust pricing mechanisms and object-functions to maximize the user's satisfaction.

2.3 Interoperability

The scientific community has long sought ways to cope with the challenges of using heterogeneous hardware, operating systems and application-specific software. One of the solutions was the concept of *Metacomputing*, ultimately leading to *Grid Computing*.

In this chapter I have detailed how the Grid concept has given rise to many different implementations. Each of these different approaches has particular strengths and weaknesses. It is becoming increasingly important in the Grid community to find solutions capable of harnessing the different capabilities of multiple implementations. Each Grid middleware offers solutions based on different philosophies for basic common services such as job submission, file storage, information systems and security whilst some offer extended services such as sophisticated resource brokerage systems or innovative workflow execution systems capable of executing complex workflows based on different computational models. There are a number of workflow engines that are being developed independently of Grid middlewares and are already able to interoperate with one or more Grid middlewares. In addition each Grid and/or workflow engine is managed with different policies and rules that must be respected.

Ironically the proliferation of differing Grid middlewares is nowadays posing the same set of problems and challenges it intended to solve, just at a higher level. In the

never-ending game of coping with the complexities generated by previous solutions, the Grid community is trying to address these new issues with a field of research, known as *Grid Interoperability*, that focuses on enabling different Grid middlewares to inter-operate with each other, and enabling combined usage of heterogeneous grid middleware.

For one Grid middleware to be able to use the capabilities of another, it must be extended to interface either directly with the other middleware, or with third-party middleware. While it is easier to just support the common services, it may actually be those very Grid-specific extended services that make interoperability desirable.

2.3.1 Major current interoperability approaches

P-Grade

The Hungarian P-GRADE (Parallel Grid Run-time and Application Development Environment) [54, 22, 26] offers an extensive toolset for creating workflows, and with the associated PROVE [75] provides for debugging and monitoring the execution and performance of applications. P-GRADE is based on GridSphere, a grid-enabled portal construction framework from Poznan. In recent times the support of P-GRADE for different Grid technologies has extended to include Condor, Globus and LCG/EGEE. One of the design goals here was to make it possible for a user to develop a workflow for the Grid without knowing the particulars of the underlying Grid technology and to be able to move the work to other Grids without altering the workflow. The focus is on allowing a user to work at a workflow level and interoperate with multiple middlewares.

GridLab

A European project led by Poznan produced GridLab [83], a spiritual descendant of Cactus, a set of application-oriented Grid services and toolkits providing capabilities such as dynamic resource brokering, monitoring, data management, security, information, adaptive services and more. Services are accessed using the Grid Application Toolkit (GAT). The GAT provides applications with access to various services, re-

sources, specific libraries, tools, etc. Applications use the GAT through a fixed GAT API. The GAT is designed in a modular plug-and-play manner, such that tools developed anywhere which conform to the GAT API will be inter-operable. In this context, GAT is a third-party middleware through which other middlewares may interoperate.

WLCG

For each pair of Grids, the worldwide WLCG [89] identifies common elements in the software stacks, and develops extra glue components to connect compatible elements. Currently, interoperability between EGEE and OSG is the most developed: bi-directional job submission is possible, OSG sites appear in the EGEE information system and are tested using EGEE test suites. It should be noted that many of the issues involved in providing ongoing interoperation between Grids are administrative rather than technical.

While pragmatic, WLCG's approach is not scalable, since for N Grids it leads to $\approx N^2$ extra glue components, i.e. $\approx N^2$ interoperability solutions.

Grid Interoperability Now (GIN)

OGF's Grid Interoperability Now (GIN) [86] has identified four interoperability services that they would like to realise in the short term: authentication/authorization, a resource schema, job submission, and data management. In the longer term, workflow, co-scheduling and accounting are also envisaged. The GIN group aims to tackle *interoperation*, which they define as 'what needs to be done to get grids working together', using existing software only. In contrast they consider *interoperability* to be the more general ability of software and hardware from various vendors to communicate.

2.4 Social Grid Agents

Chapter 1 postulated similarities between Grids and societies, in this thesis I explore a high level resource allocation system based on social behaviour and I do so by proposing a social and economic paradigm as an abstract platform upon which an experimental implementation can be constructed.

Grids allow computational resources to be shared in a relatively secure environment. In this respect Grids allow users and resource owners to belong to different administrative domains. The separation of these domains poses many challenges: firstly, there must be a security infrastructure that connects the domains; secondly, (and more importantly for this research) there must be a way to find suitable resources across domains. Better still, there ought to be a way to find "optimal" resources and, conversely, an "optimal" way to use resources. When resources and users are in the same administrative domain it is relatively easy to define allocation algorithms aiming at the optimization of some metrics but in a system where these domains are disjoint it is even difficult to define what optimal means. These difficulties stem from different reasons: firstly, if the preferences of all the actors are to be taken into account by one or more actors when they make their decisions, there must be a homogeneous way of defining them, secondly all the information to be available to all the relevant decision makers and this is hard to achieve on a both practical and theoretical accounts. Information can be outdated when it reaches the relevant decision maker, the amount of information can result far too conspicuous to be handles and, finally, it cannot be assumed that all the involved actors will always be willing to share all their preferences to all the other actors.

Societies and economies are the solutions that mankind devised to solve similar problems: the allocation of limited resources among actors with incomplete knowledge, different aims and goals and the need of a relatively stable structure. Societies allow groups of individuals to co-exist in relative stability and provide grounds to implement different allocation philosophies for scarce resources. I hope that, by reflecting social behaviours in a Grid, I will be able to provide similar levels of flexibility in the allocation of resources. In fact, both allocation problems have many points

in common and it is reasonable to assume that solutions devised for one might also prove profitable for the other. This consideration is at the very base of the desired paradigm.

This thesis speculates that Grids might be seen as *"societies in a digital environment"* where goods and services are exchanged and where actors, either directly controlled by humans or of entirely automatic behaviour, expose social behaviours.

I see the potential similarities between human and grid societies as multi-faceted: *optimization*, *diversity* and *complexity*.

2.4.1 Optimization

Mankind has developed societies supporting a variety of philosophies of sharing and allocation. How the different actors concur in the definition of the allocation mechanism appears to depend on the nature of the society itself: in some all the members contribute almost equally to decisions on allocation parameters, while in more "autocratic" ones only few or even only one of the members will decide (sometimes to their own advantage). The author cannot claim any special expertise in the subject, but to cite John Kenneth Galbraith in his "History of Economics" [50] and "The Affluent Society" [41], the allocation mechanisms appear to be based on the current *"common wisdom"* of that society. Although the research of important pundits (as for example Stiglitz [87, 88]) points out many failures and shortcomings, in one "common wisdom" it is believed that *"free market"* economics can approach optimal allocations in many (if not most or even all) scenarios.

The debate on the validity and shortcomings of different economic models is also present in Grids as presented in Section 2.2.2, there is no strong consensus on the best model for resource allocation. The theory of General Equilibrium Analysis fails to explain failure in the market system and ignores aspects such as trust and government intervention that are particularly important in Grids.

While competitive, free-market allocation mechanisms appear to be able to reach allocations that are a reasonable compromise among many members of a society it also appears that co-operative behaviours are best suited for others. Finally, in

many scenarios, it seems that free-market allocation cannot even approach optimality without the intervention of a regulatory entity often referred to as the *government*. Governments are also key players for their capacity in investing in infrastructure that are of common benefit for the entire society. This behaviour of governments, often referred to as *Keynesian Investments*, will inspire one social topology investigated in this thesis. Investment in public infrastructures is a particularly important topic because many Grid infrastructures are publicly funded.

Human societies appear to have the capability of simultaneously encompassing allocation behaviours based on different (and possibly even contradictory) philosophies and being relatively stable and resilient at the same time.

For the time being, Grids are, fortunately, much simpler than human societies; they nevertheless face a very similar problem: the need to find a resource allocation philosophy that takes into account the different criteria of optimality of the various members. A job owner's idea of "optimal execution" will probably differ from that of a resource owner and a Grid that consistently ignores the needs of users or resource owners would soon be deserted by one of the two parties and collapse. As it is in human scenarios, "free market" economics might approach optimal allocations in some scenarios while co-operative behaviours might be best suited for others. For these reasons I believe that in a Grid actors should be able to concurrently implement a large variety of allocation philosophies in a stable environment.

2.4.2 Diversity and Interactions

The social paradigm can also help us in understanding yet another grid characteristics: *diversity*. Human societies differ from each other in many aspects: behaviours, philosophies, values and consequently priorities. History is both nursery and graveyard to a myriad of concepts and implementations of the abstract concept of society. Although with much less beauty and complexity, the different grids that have been proposed and implemented show the same tendency to variety and originality. All these many different solutions to the same problem are in competition but also coexist and could potentially interact with each other as human societies do.

2.4.3 Complexity

There are cases of human societies where the emergence of a stable environment for a significant time coincided with an explosive increase in the complexity of the society itself and its products. In its beautiful book Jared Diamond, *Arms, Germs and Steel* [34], postulates causality: that whenever humans can rely on sufficient resources and social stability, they have the potential to engage in a momentous increase in the complexity of their social structure and of the complexity of their products.

An analogous coincidence is now happening in Grids: stable Grid environments have emerged, and now they are offering services of ever-increasing complexity. The first Grid middlewares offered limited services for job execution and data storage, then workflow engines followed, and currently highly sophisticated services even include semantic descriptions of themselves and complex metadata management.

Moreover, the possibility of harnessing multiple different middlewares (e.g. using P-Grade) allows the creation of even more complex tasks or workflows. The CrossGrid [46] flooding management system [47] is just one enlightening example of a complex and sophisticated system in Grid computing. Examples such as this encompass and orchestrate many different services that can be arranged in hierarchies of different complexities.

A society-based approach to grid interoperability that yields the concept of a *metagrid* is detailed in Section 3.10.

2.5 Motivations

In Grid computing, the economic perspectives and beliefs used to investigate the resource allocation problem are as vast and diverse as those used to interpret real life. This consideration suggests an *economic-agnostic* approach based on a social paradigm capable of encompassing different economic philosophies to give allocation systems the flexibility to change its behaviour depending on the social context.

The fact that many different implementations of the concept of Grid are already

present and the belief that full interoperability defined through standardization will not achieved in a short time combined with the increasing complexity of the Grids suggests an *technological-agnostic* approach based on a metagrid, a conceptual view of interoperability based on the concept of border agents that act as translators and mediators at a technological and logical level.

I propose a solution (detailed in Chapter 3 and Chapter 4 capable of enforcing different allocation philosophies across different Grid middlewares).

Chapter 3

Methods

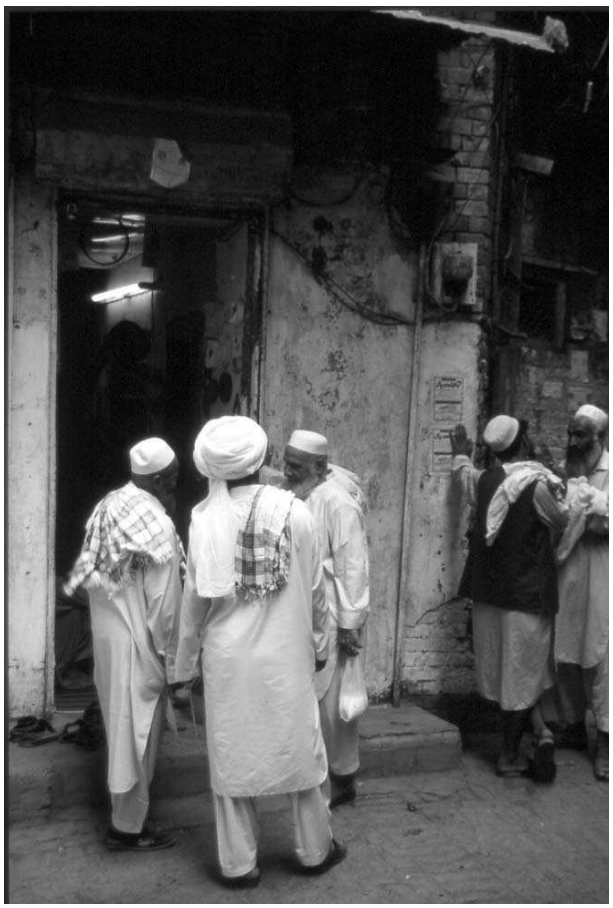


Figure 3-1: Social Non-Grid Agents exchanging information.

Geometry is not true, it is advantageous

Jules H. Poincare *I prefer drawing to talking. Drawing is faster, and leaves less room for lies.*

Charles-douard Jeanneret-Gris dit "Le Corbusier"

3.1 Introduction

This chapter begins with the explanation of the main concepts on which Social Grid Agents are based, it then describes the abstract architecture at three different levels: the topologies I used to model the different allocation philosophies, the architecture of the agents and the agent's native language.

3.2 Social Grid Agents Main Concepts

3.2.1 Production

Production in grids can be described using the concept of a *production chain* that goes from the basic goods, or *factors*, to *intermediate commodities* and, in the end, to the final service, or *produced commodity*. This conception of grid jobs and workflows as supply chains is illustrated in Fig. 3-2 while Fig. 3-3 shows a more detailed view in which the factors are divided into three main categories:

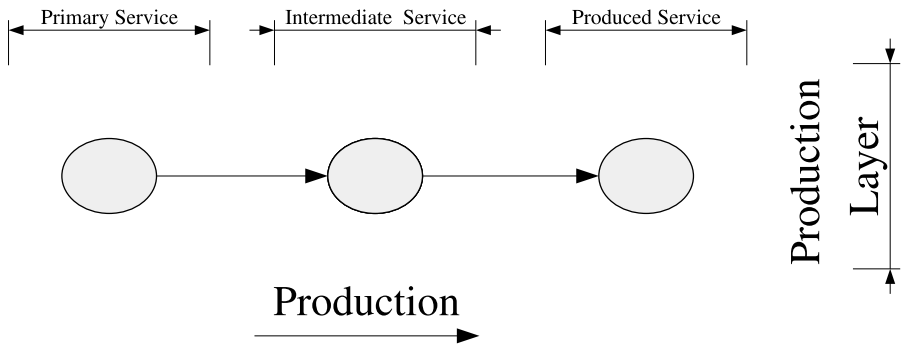


Figure 3-2: Representation of a grid service as a micro-economic supply chain

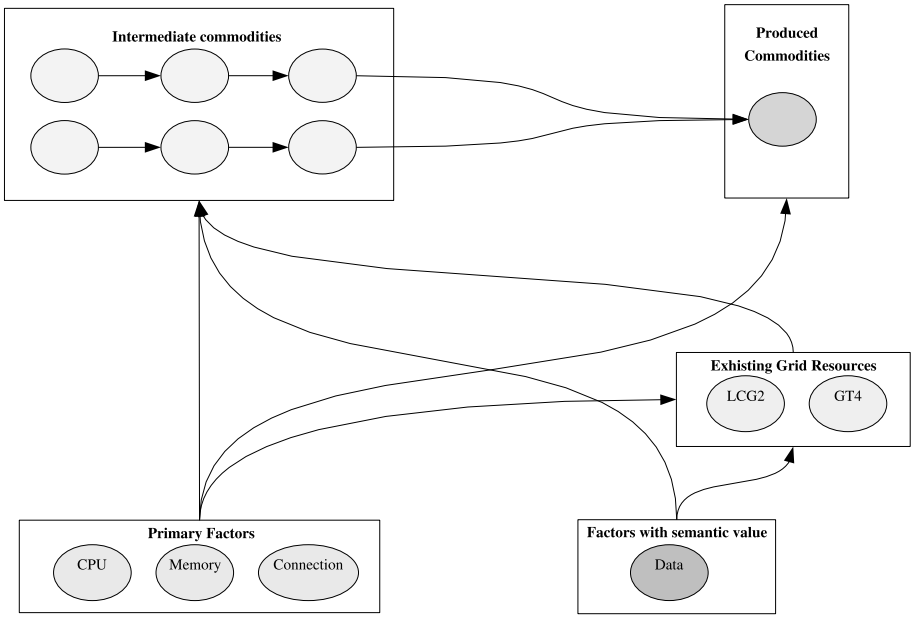


Figure 3-3: Detail of a grid service production chain

- *primary factors with no semantic value*: CPU time, memory storage and network connection offered directly (e.g. through a scheduler);
- *primary factors having semantic value*: data;
- *existing grid resources*: such as job submission, information systems and file management services; these are, in reality, produced commodities but are modelled as factors because their internal production chain is not explicitly considered even though the price at which they are sold may reflect the consumptions of primary factors.

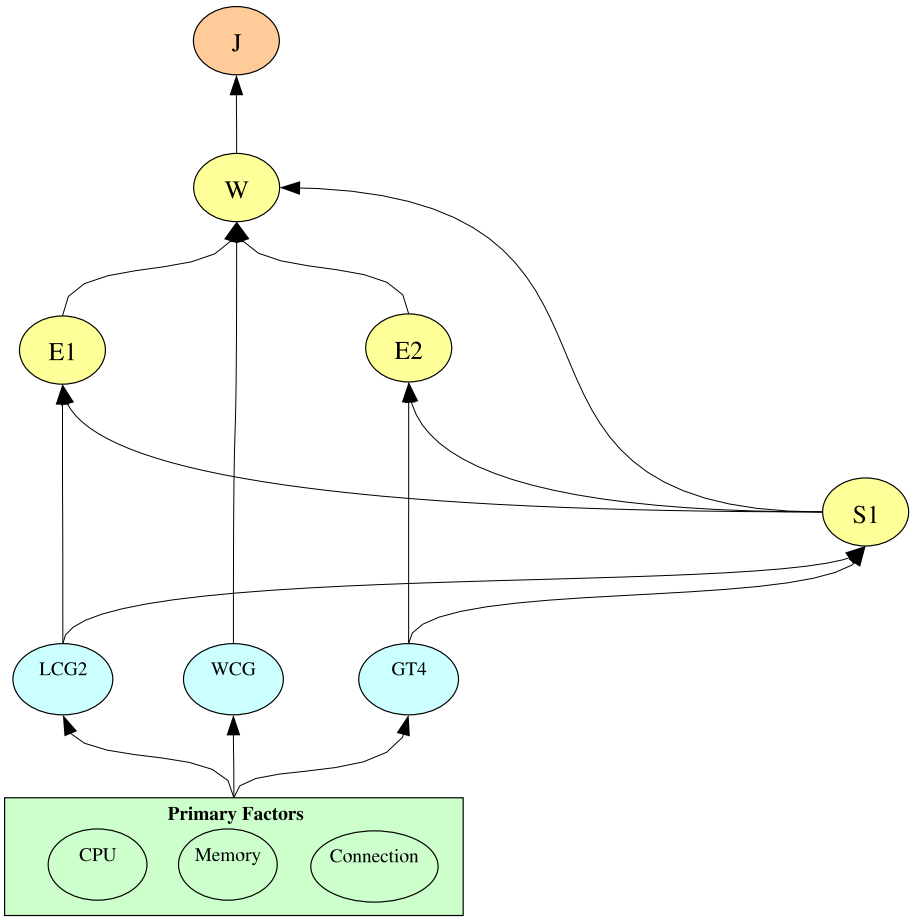


Figure 3-4: Example of the production paradigm

The distinction between data and primary factors, although not directly used in this first approach to the problem, may turn out to be quite important as the real economic value of a data set can be represented by its meaning or semantic.

At each step of the production chain services are combined into higher level ones by one or more actors. Let us call these actors *Grid Production Agents* or, more concisely, *Production Agents*, reflecting society's conception of an agent, *viz* a free agent, without implying any sense of Agent Technology [25] as understood in Computer Science. In this way we can represent grid services ranging from simple to complex. For example, a job spawning multiple grids and necessitating different grid services can be decomposed and described into lower level factors, that can be further decomposed until a service provider capable of performing them is found.

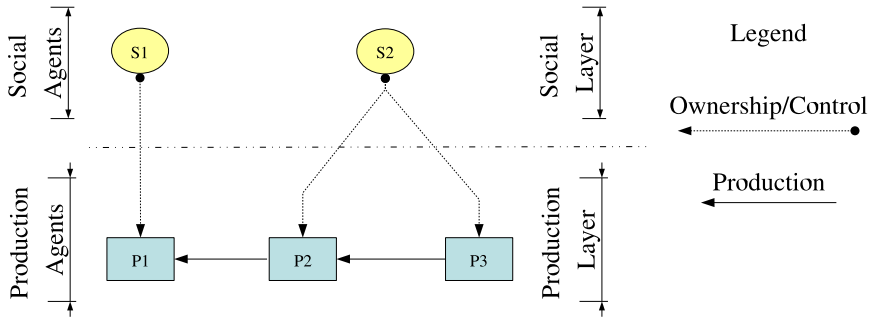


Figure 3-5: Example of the production paradigm

Fig. 3-4 illustrates an example with a job encompassing different middlewares for execution and storage. The final service consists of a job that needs different services orchestrated in a workflow. As a first step we can see that the only service needed to execute the job J is a workflow engine W able to understand the language in which the workflow is described. The workflow engine, in turn, will need to execute jobs and save and retrieve data using the execution services *E1* and *E2* and the data storage service *S1*. When unfolding the production chain of the job in an abstract fashion there is thus far no consideration of which real services in the grid would be capable of fulfilling the requirements or whether they are available on the market.

The next step consists of finding real and available Grid services capable of executing the needed services. In the example, the workflow can be executed by a *WebCom* middleware [60], the job executions are compatible with the *gLite* [27] and *GT4* [92] middlewares while the storage needs can be fulfilled both by the gLite and GT4 middlewares. Eventually, all these real Grid services will consume primary factors such as CPU time, storage space and bandwidth.

This paradigm considers the composition of the various grid services as a supply chain. Not all scenarios can be dealt with in such a reductionist fashion. It does not deal with problems such as ownership of the services or their compatibility with each other; therefore it can only be useful and implementable in the real grid world if the job's owner can access all the needed services and if all the needed services can interoperate with each other. As this is very often not the case, the paradigm needs to be further expanded.

3.2.2 Ownership and Control

To describe the concepts of ownership and control let us add a new layer: the *social layer*. Now the paradigm is composed of two interacting layers, one where production takes place and another that hosts agents owning and/or controlling the production entities. Each layer can be complex or even hierarchical, but the production layer does not have social properties, whereas the social layer does. Henceforth let us call the agents that reside in the social layer *Social Grid Agents* or, for the sake of brevity, *Social Agents*. This is shown in Fig. 3-5.

Agents in the social layer can own or control or use none, one or more production agents. A social agent should also be able to exchange and trade the use of, or the services produced by, the production agents it controls. This allows our social and economic paradigm to exhibit behaviour such as the *purchase of services*, the *rent of production agents* and the *exchange of ownership* of one or more related production agents. Let us propose that these relations between social and production agents can be of *ownership*, *control* or *usage*:

- *Ownership*: when social agents own production agents they have complete control over them; they can sell, rent or donate them; they can set access rights for controllers and users and they can set policies to regulate the production process.

- *Control*: when social agents control production agents, they have partial control over them (that may have been delegated by an owner); they can set access rights for users and set (possibly a sub-set) of production policies.

- *Usage*: when social agents use production agents they can access only the services that they were granted by a controller or an owner.

Multi-modal relations

To better understand, design and implement transactions such as rent, purchase and donations it is useful to propose different modalities in which the above mentioned relations can relate both to time and the space of the resources.

Relations might have different time modalities such as:

- *Absolute*: Either *true* or *false*, especially as a logical attribute of ownership relationships.
- *Token-based*: to allow the submission of a certain number of requests to a production agent, especially for simple service rental scenarios where a social agent buys (or receives) a given amount of service execution rights on a production agent that it does not control or own.
- *Deadline-based*: to allow the control of a production agent for a given amount of time, especially for more complex service rental scenarios where a social agent acquires temporary control of other production entities to engage in the execution of a complex service.
- *Time slot-based*: to allow the use of the services of another agent until the allowed amount of time has been used, especially for reserving processing and storage services.
- *Combined*: to allow modalities to be combined together: e.g. token-based and deadline-based modality might be combined to obtain a modality where tokens have expiry dates while time slot-based and deadline-based modalities might be combined to obtain a "perishable" reservation on computational and storage services.

Relations might also be allowed with regard of the controlled resources. These concepts relate to the *allocation modalities* examined further in section 3.2.6. They can be roughly divided into:

- *Complete*: give ownership and control over all resources controlled by the agent over the specified period of time.
- *Partial*: give ownership and control over a part of all the resources.

It is clear that some form of constraint has to be provided over the modalities with which each of the actors sets the rights of the others. If no such control is provided

there may be cases in which agents gain control of other agents in an unforeseen way. As an example, let us suppose that an owner O grants control to an agent C to one of his production agents P, with a deadline modality that expires at a date T. As the controller can set any kind of access rights for the user it is possible for the controller to set absolute rights to a user U. These rights, being absolute, will outlive the range of time in which C is allowed to control P. To avoid such problems one might constrain the setting of the access rights as follows.

- For owners: owners might be allowed to set any kind of access rights and they might be enforced directly.
- For controllers: the access rights that controllers enforce might depend on the access rights they have, for example:
  - Absolute modality: enforced directly.
  - Deadline modality: If the controller has Deadline modality access rights, its commands change in the following way.
    - Absolute modality → Deadline modality.
    - Deadline modality → Deadline modality with an expiry date that is the minimum between the controller's and the target's settings.
    - TimeSlot modality → TimeSlot and Deadline modality.
    - Token modality → Deadline and Token modality.
  - Token modality: token modality is a way for controllers to grant execution rights to users, and thus is usually granted to users and not controllers. Should the case arise of tokens being granted to controllers they are treated as if they are users: Token modality → Token modality.
- For users: users might not be able to set any access rights.

The above simple modalities are powerful enough to describe some interesting basic topologies of ownership and control.

3.2.3 Social Topologies

In *Production Topologies*, Social Agents control Production Agents. A similar process is possible among Social Agents, thereby forming *Social Topologies*, see Figure 3-10. Both topologies are explored further in Chapter 3.3. Although more complex, the possible control topologies amongst the social agents are similar in nature to those between social and production agents. Owners can set policies and access rights for controllers, controllers can set policies for clients. Such policies can encompass, among other information, the nature of the social behaviour of the agents such as the type of purchase to which an agent is entitled, e.g. purchase for free, special prices, full price or the maximum credit/debt it is entitled to. These forms of social control, along with the concept of exchange modalities explained in the following section (3.2.4) describe the social behaviour of the agents.

3.2.4 Exchange

At each production steps an actor (either human or program) assembles factors and, possibly, intermediate commodities to create a produced commodity. If at any given time an actor has not enough factors to produce the commodities he wants, he may engage in social or economic transactions so as to obtain the commodities he needs. To model this behaviour concepts and terminology may be borrowed from microeconomics to define *Endowments* as the set of commodities owned by the actor, *Needs* as the set of commodities needed by the actor, *Supplies* as the set of commodities that the actor has in abundance and can be traded or donated, *Demands* as the set of commodities that the actor cannot cover with its endowment and *Satisfied Needs* as the set of commodities that the actor can cover with its endowment. An actor owns an Endowment set of commodities and faces a Needs set of commodities. Demands and Supplies are obtained from Endowment and Needs as illustrated in Fig. 3-6.

In this case, the Endowment and the Needs sets partially overlap. Where they overlap, that part of the Needs has been satisfied by the Endowment. The actor will be able to trade or donate the remaining part of its Endowment, and will also need to trade for the unsatisfied part of the set of its Needs:

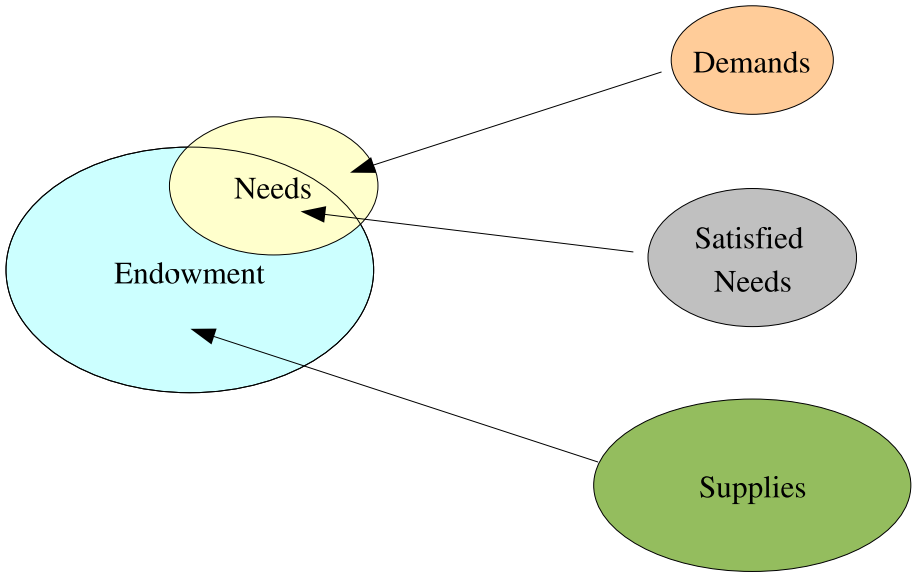


Figure 3-6: Demands and Supplies.

$$Supplies = Endowment \cap \overline{\text{Needs}} \tag{3.1}$$

$$Demands = Needs \cap \overline{\text{Endowment}} \tag{3.2}$$

$$SatisfiedNeeds = Endowment \cap Needs \tag{3.3}$$

Besides this general case there are some particular configurations worthy of interest:

- Disjoint sets : In this configuration, illustrated in Fig. 3-7 the Needs and Endowment sets are completely disjoint:

$$Endowment \cap Needs = \{\} \tag{3.4}$$

$$Supplies = Endowment \tag{3.5}$$

$$Demands = Needs \tag{3.6}$$

$$SatisfiedNeeds = \{\} \tag{3.7}$$

- Abundance : In this configuration, illustrated in Fig. 3-8 the Needs are completely satisfied by the Endowment:

$$Needs \subseteq Endowment \tag{3.8}$$

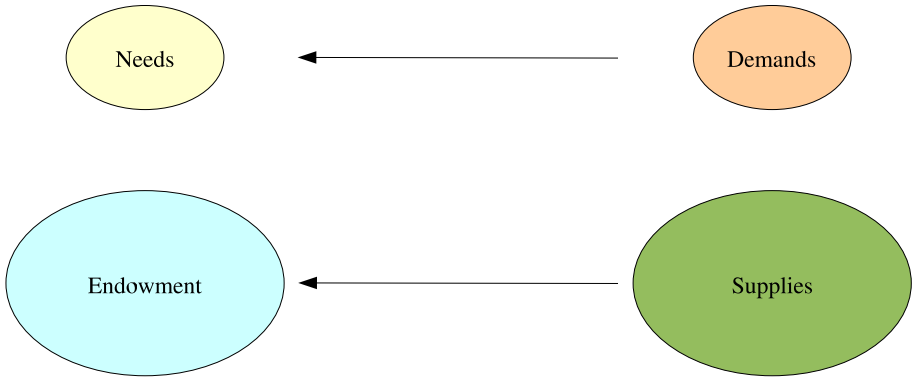


Figure 3-7: Demands and Supplies are disjoint sets

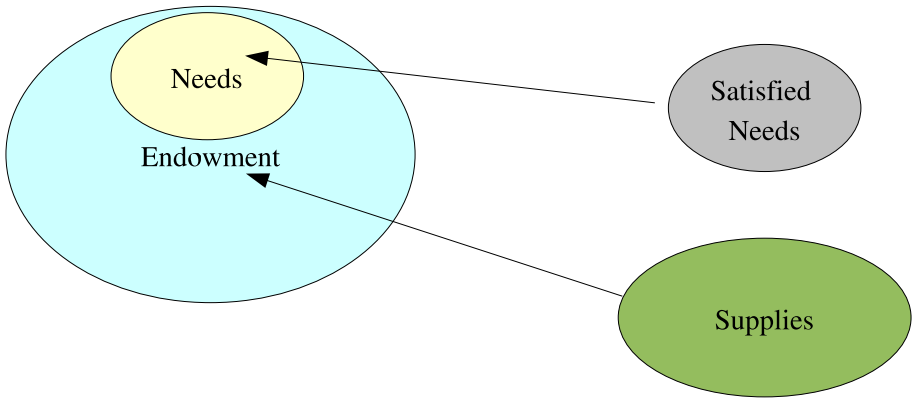


Figure 3-8: Abundance, Needs are completely covered

$$Supplies = Endowment \cap \overline{\text{Needs}} \tag{3.9}$$

$$Demands = \{\} \tag{3.10}$$

$$SatisfiedNeeds = Needs \tag{3.11}$$

- Scarcity : In this configuration, illustrated in Fig. 3-9 the Needs cannot be satisfied by the Endowment:

$$Endowment \subseteq Needs \tag{3.12}$$

$$Supplies = \{\} \tag{3.13}$$

$$Demands = Needs \cap \overline{\text{Endowment}} \tag{3.14}$$

$$SatisfiedNeeds = Endowment \tag{3.15}$$

If an actor owns an Endowment set of resources and faces a Needs set of resources, the result is two disjoint sets: Demands and Supplies, that will be traded, donated or

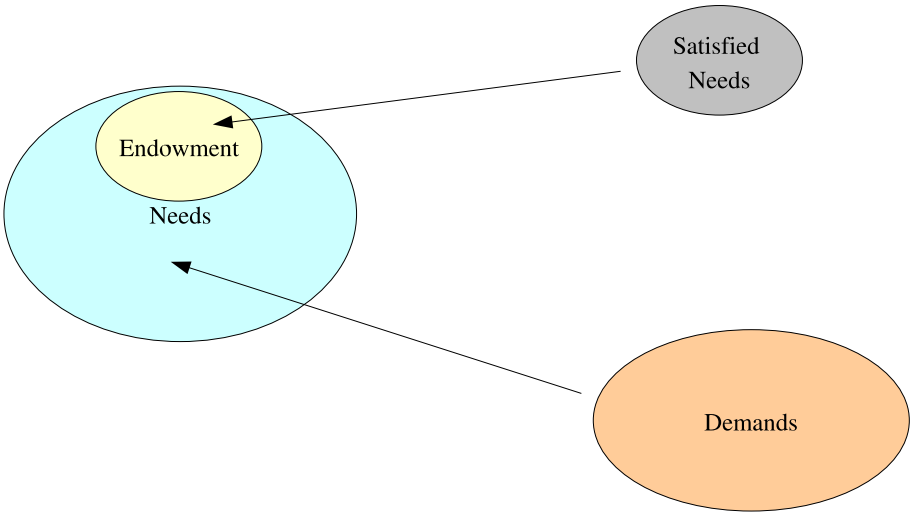


Figure 3-9: Scarcity, Needs cannot be covered

acquired in the societies that the actor is joining. In order to do this, actors engage in relations, both competitive and empathic, to fulfill their Needs sets. These types of relations could be:

- Co-operative relations: such relations involve access to resources without the need of a balancing transaction representing a payment. These relations can be of the following nature:
  - *donations*
  - *lending and borrowing*,
  - *Keynesian investments*[1]
  - *common goods*
- Self-interested non-monetary economic relations such as bartering.
- Monetary economic relations that involve payments and monetary transactions.

[1]We term *Keynesian investment*, in honour of the economist John Maynard Keynes, a relationship where an Institution invests in Grid resources binding their use, partially or fully, to a certain user or usage

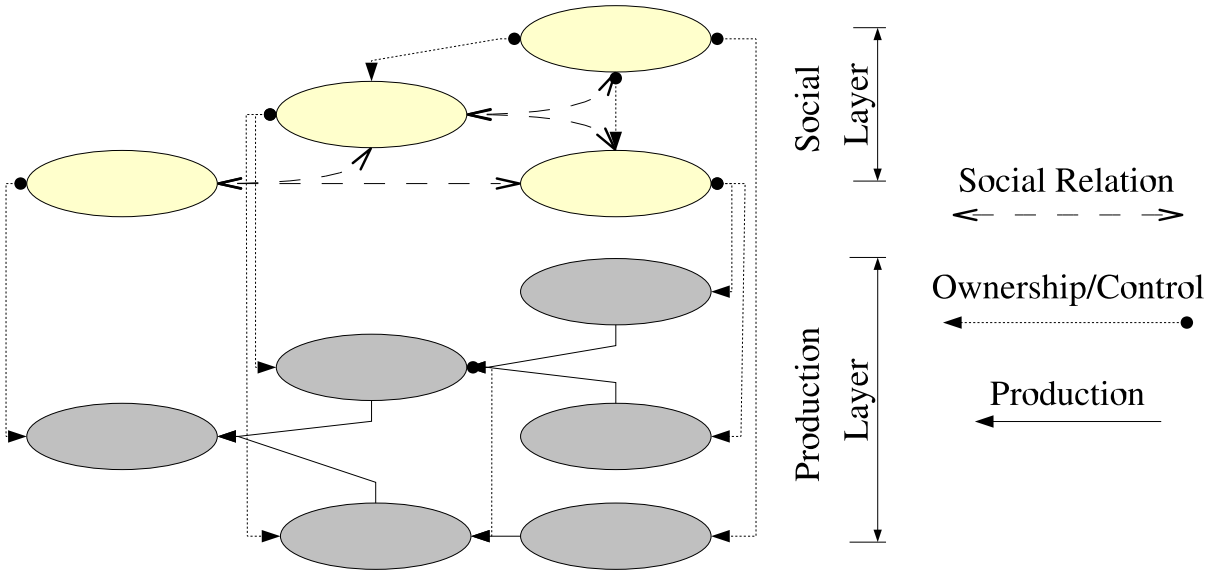


Figure 3-10: Agent's topologies

3.2.5 Value and Price

As yet there has been no definition of what kind of information the agents exchange with each other or on the paradigms that are at the base of their behaviour. Two basic concepts in the social and economic paradigm are *value* and *price*. These concepts can be very loosely borrowed from economics and used in a similar fashion but without the strictness and formality of economics.

Price is a mathematical entity expressed in a unit (a currency that has to be recognized by all parties involved) on which two parties agree for an economic transaction. Almost everything has a price, from tomatoes in a supermarket to human lives in an insurance company. It is easy to define the concept of price for digital goods. One hour of computation at a given time on that CPU at certain conditions will have a certain cost. One month of 1 megabyte of space accessible at given conditions will have another cost. As a matter of fact, digital services such as computation and storage are already sold in real markets against real currencies.

Unfortunately it is difficult to define strategies solely based on price when not entirely economic behaviours arise. In grid computing there are social behaviours that cannot be easily described with a mere economic explanation. Cooperation may require decisions entailing financial losses to maintain social relations. It is difficult

to give a monetary price to a social relation. Offering a pint to a friend in a pub is an action with a very clear financial outcome (loosing the price of a pint) but involves reasoning that is not easy to model in mere monetary terms. Giving a Euro away in charity, again, has a financial outcome (the amount of money given away) that can be clearly measured in a currency but a benefit (the moral well-being, or the temporary easing of the sense of guilt, depending on how inclined you are to cynicism) that can hardly be measured in financial terms. As informal cooperation and "charity" mechanisms are present in Grid computing it is necessary to find another, more private, metric to describe how an individual agent perceives the "worthiness" of a resource, a service or an action. Let us call this *value*, an entity to be used both to define prices and to help the decision-making process.

One useful simplification is to say that the production layer is only aware of value. The value of services and resources could be either inferred or directly set by the social layer. In the paradigm a production agent could be constrained to make decisions only on value. It would be possible, as an example, for a production agent to perform a service up to a certain value or to report to its owners or controllers the value of a performed service. The value of a service could be defined or it could be inferred from metrics through a defined function.

On the other hand, the social layer might not only define the value of the services it controls but might also infer a price from it. A social agent may map values into prices without considering the buyer's identity as done in super-markets or instead it might define a price depending on the value and the identity like a seller who decides to offer a discount to a good costumer or, finally, it might decide to donate services up to a given value because the social relation is deemed worthy of such financial sacrifice.

Fig. 3-11 illustrates this concept in more detail. Here the production agents P1, P2 and P3 define a value for their services based on metrics or on the control decisions made by their controllers S1 and S2. On the social layer, S1 and S2 engage in social exchanges. The social decisions made by S1 and S2 are made on the basis of both value and price. The economic transactions between S1 and S2 are defined only in

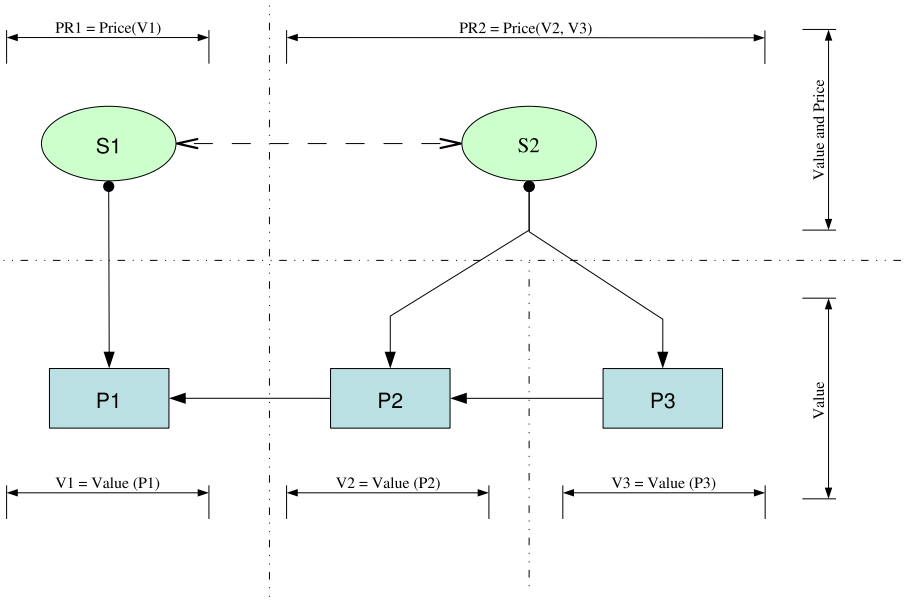


Figure 3-11: Value and price

prices as the unit must be recognized by both parties. Value is a private entity, price on the contrary is shared.

3.2.6 Policies and Modalities

So far, the paradigm is capable of describing production systems composed of various agents. This system is flexible because the production process can be re-arranged and it should be able to cope with disjoint administrative domains as the production agents can be owned and controlled by different agents. Finally, it should be able to cope with the social complexity of the grid as it allows for a reasonable variety of topologies. The paradigm describes a two-layered environment where agents act and define topologies of different kinds: production, control/ownership, social and value/price, as described in Fig. 3-12. The information must also be able to express both the *functionalities* required and the *policies* with which the functionalities must be performed. Functionalities and policies are closely intertwined as policies can constrain functionalities and different policies can be applied to different functionalities. Thus far there has been no consideration of the nature of these information flows.

Agents exchange information in the form of messages. The way in which the

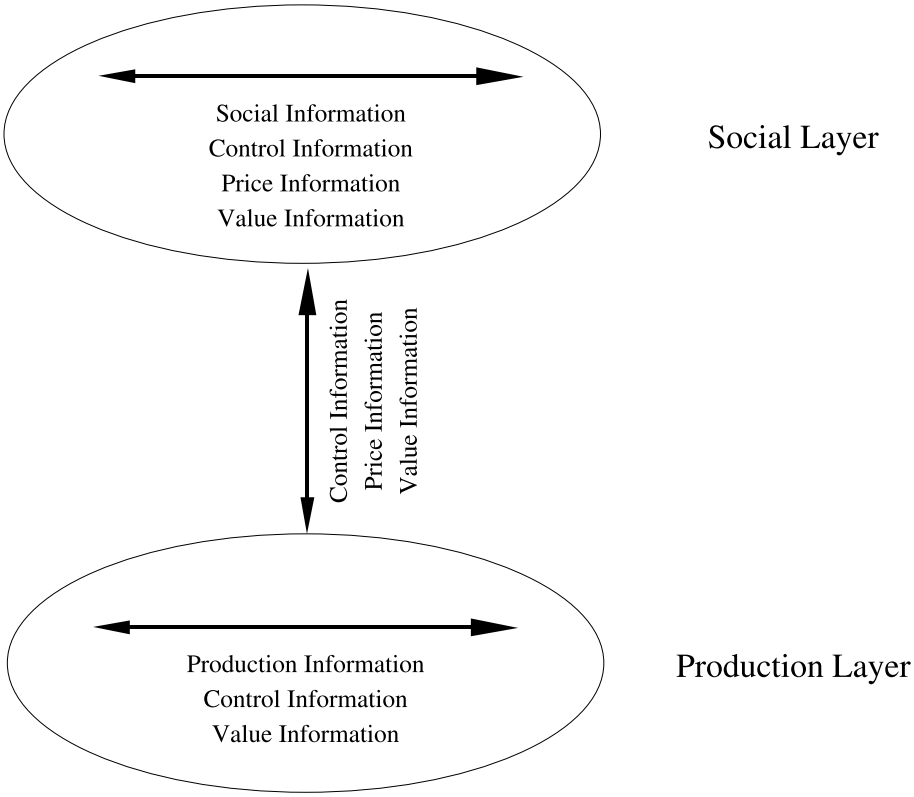


Figure 3-12: Information flows

messaging is conducted does not need to be considered yet. The important point is that these messages should be able to express concepts such as: *"Execute this task with these modalities"* or *"Is it possible to execute this job with these modalities in that environment ?"* or *"What would be the cost of executing this task with these modalities ?"*. An analysis of these messages enables one to see that the information of a message must at least contain:

- *A subject* that often is the sender of the message.
- *An action* describing what must be done. An action might be an execution, a request for information, the definition of behavioural patterns of the agent or the cancellation of some action.
- *An object*: describing the object of the action. An object could be an action itself in the case of complex actions.
- *A set of modalities*: describing how the action has to be carried out.
- Optionally *The identity of the beneficiary (i.e. the ultimate requester)*: describes the identity of the agent that initially requested the action.

The agents that process these messages must take decisions based on their relationship with the subject of the message. A request of the type "Execute job j with modalities m" should be accepted or rejected on the basis of the compatibility of the actions and their modalities with the status of the agents and especially how it wishes to handle such requests, i.e. its *policies*.

Generic Policies and Modalities

The "common wisdom" of a society often is embedded with a generic set of beliefs that govern policy, and an analogous provision would benefit the proposed paradigm. Hence generic policies and modalities could define behavioral parameters that are common both to the production and to the social layer and are defined by the relation that links two agents. An example is illustrated in Figure 3-13. Every agent supports a set of policies, the set *SP* which encompasses all the possible policies that can be

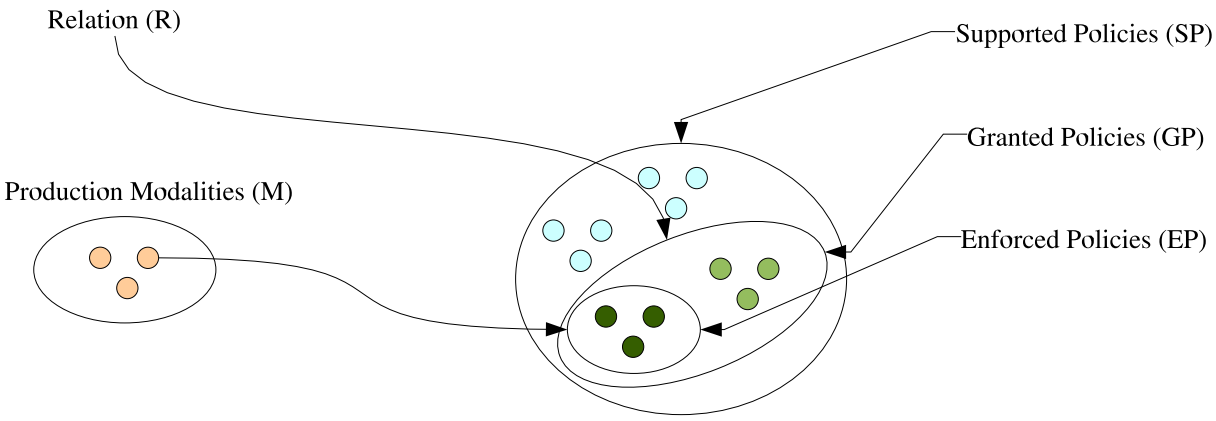


Figure 3-13: Policies and modalities

enforced during the execution of a certain action. The nature of the relation between the agents defines a subset *GP* of policies that can be granted. Finally, the modalities with which the action is to be executed define a final subset *EP* of policies that will be enforced. If the subset of enforced policies is empty the action cannot and will not be executed by the agent.

The policies could consist of:

- *Authentication policies*: that define how authentication is to be executed.
- *Authorization policies*: that define if and how the actions specified in a message are to be executed. It would not be unusual for authorization policies to be defined in terms of other policies.

Production Policies and Modalities

For the individual production agent, production policies and modalities could define different aspects of how a service is to be executed. They might cover:

- *Allocation policies* that describe allocation parameters that define the execution of a certain task.
- *Execution policies* that describe specific execution parameters such as timeouts.
- *Value policies* that describe what is the value of the services offered by the production agents.

- *Accounting policies* that describe if and how an action is to be accounted for.

Social Policies and Modalities

Similarly, for the individual social agent, social policies could define parameters of the social behaviour of the agents. They might cover.

- *Pricing policies* that describe how resources and services are priced with regard of the relationships with the other social agents.
- *Selection policies* that describe how resources and services must be selected.
- *Billing policies* that describe how credit transactions are to be performed and if they must be performed through a bank or a simple transfer of credit is sufficient.

3.2.7 Additional Social Dimensions

In paragraph 1.2 we have stressed our belief of a close integration of the concept of economies and societies. We have described how social interactions can encompass many different economic models and how economic models can be influenced by social relations. We now propose to describe this as the different *Social Dimensions* of a relation; this concept is tightly bound to those of *Optimization*, *Diversity* and *Interactions*. In fact, a *Social Dimension* is the equivalent of finding in the social layer a set of *Social Services* capable of fulfilling needs that an agent cannot meet alone. Even if in real life the complexity of the analogous concepts is far beyond the reach of this investigation, we can nevertheless try to isolate some of these dimensions within the paradigm proposed by this thesis. Let us consider an actor that has complete knowledge, perfect memory, perfect "honesty" and full mutual trust with other actors with the same characteristics. Then this actor will not need any market as it will know with absolute precision where and how to find all the needed services. It will not need any legal system or law for the protection of the consumer as it will be able to rely on the perfect honesty of all its social parties. Finally it will not need any bank as all transactions may be based on complete trust; having perfect memory,

each agent will remember how many credits it will own and will simply subtract or add it at every transaction[2].

Sadly, this is unlikely to be practical. Agent behaviours will be defined by their human programmers. Among humans mischievous behaviour is so common that even the tiniest of societies do not last long without rules and laws to control "wrong" behaviours (as an example, spam and hacking). Knowledge is limited at best, trust is a wonderfully rare (and lavishly squandered) social commodity and even the most gifted in memory will fail to keep track of all the economic transactions that happen even in a single day.

Faced with its own shortcomings an agent must seek in its social layer whatever it lacks. We try here to define three social dimensions that compensate for incomplete memory, incomplete knowledge and incomplete trust.

A *Banking Dimension* is a social dimension where actors delegate a second party (a Bank) to keep track of their endowment and economic transactions. This dimension is also where, in future, more sophisticated scenarios, the computation of the *overall amount of wealth* and therefore the *overall amount of credits* of a society will be computed either through the connection to the real economy or through a mint authority that issues credits depending on the amount of resources available.

An *Indexing Dimension* is a social dimension where actors rely to a second party (an Index, or a Market) to discover previously unknown parts of the society, the actors therein and the services they offer.

A *Trusting Dimension* is a social dimension where actors rely to other parties to determine how much and if at all a third party has to be trusted. Trust can be based on a combination of three different factors: experience, authority or shared knowledge. The first is based on the recognition of behavioural patterns through time, the second is based on the authority of a party and the third is based on the opinion of other other parties, whom we trust. Currently only the approach based on the authority has been explored to define trust-based topologies (for the definition of prices) in this thesis.

[2]We do not consider here the services that a bank offers regarding security, investment or interests

3.2.8 Limitations

The paradigm described in this chapter should be flexible and powerful enough to allow the Social Grid Agents to mimic in the Grid world some behaviours that constitute the bare basis of social and economic interactions. Agents can arrange in arbitrarily complex structures to engage in the production of services. These production structures are defined depending on the social interactions among the social agents. This will allow for the definition of allocation philosophies of different nature. This paradigm allows the implementation of testbeds for the evaluation of different allocation strategies. The paradigm also offers an initial support for the implementation of the concept of *value* and *price* and their use for decision making. This paradigm allows the description and implementation of a broad range of simple social and economic models and it also allows their coexistence for experimental purposes.

Although the paradigm is flexible enough to describe a large variety of social and economic behaviours, the current architecture and implementation of the agents (detailed in Chapters 3.12 and 4) is not rich enough to allow the description of very complex social and economic systems. In particular, behaviours that need complex self-aware actions such as stock and futures markets cannot be modelled with the current architecture and implementation. The flexibility of the paradigm, on the other hand, should allow encapsulation of such advanced behaviours in new but backwardly compatible agents that can be interfaced with the current ones in a relatively seamless fashion. We will return to this subject in Chapter 10.

3.3 Topologies

In Section 2.4 we have described the *Social Grid Agents* paradigm intended as the basis of flexible allocation systems in Grid Computing. These agents can be seen as having four types of topologies: *production*, *control/ownership*, *social* and *value/price*. This chapter is devoted to the description of these topologies

3.4 Production Topologies

3.4.1 Simple Producer

The simplest of these topologies is the *Simple Producer* shown in Figure 3-14, where a client contacts a Social Agent and negotiates the acquisition of a grid service. When the social or economic transaction is accepted by the two parties the Social Agent gives instructions to the Production Agent it controls to execute the required service. Then the service is performed and the results are sent back to the client. Even such a simple example can be implemented in many different ways. From a social and economic perspective there may be or not a payment required for the service, depending on the relationship between the agents. There can be different payment policies such as pay beforehand, pay after the service is executed, pay only if the service is successfully executed, etc. All these different ways are defined by the policies that control the agents behavior and by the modalities that define how services are to be executed.

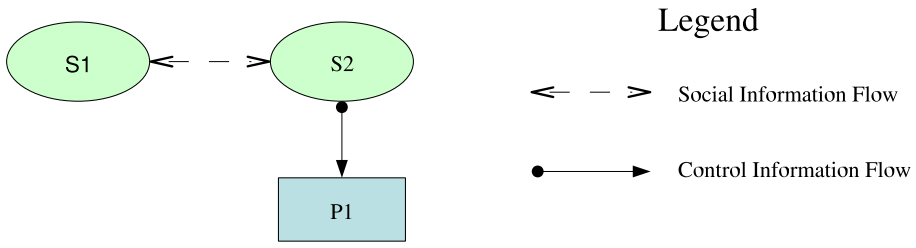


Figure 3-14: Simple Producer.

3.4.2 Service Rental

A slightly more complex control topology is the *Service Rental* as shown in Figure 3-15. Here two Social Agents exchange the services of one or more Production Agents. After the social or economic agreement, the grantor Social Agent *S1* instructs the Production Agent *P1* to accept requests from the grantee agent *S2* with a certain modality (e.g. token modality, time slot modality, deadline modality or combined modalities). The grantee is now capable of directly using the Production Agent.

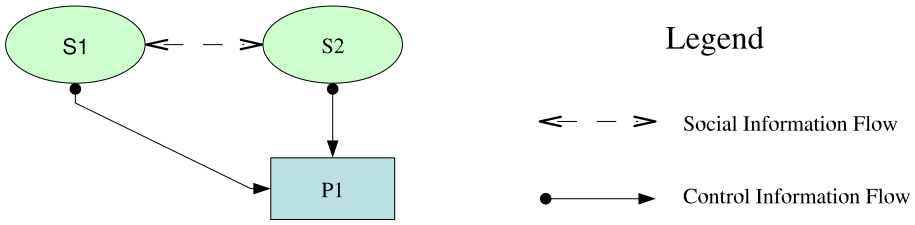


Figure 3-15: Service Rental.

3.4.3 The Company

We call a *Company* the control topology where a Social Agent controls one or more Production Agents. The control exerted by the Social Agent lets it define the topology of the production flow, as illustrated in Figure 3-16. As the ownership and the control of the Production Agents can be absolute (if described by the absolute modality) or relative to tokens or time (if described by other modalities), a company can avail itself of the services of Production Agents it does not control directly, provided that it can either obtain partial control of them or be able to purchase their services from their Social Agents. This second scenario, which we term *outsourcing* is described in Figure 3-17.

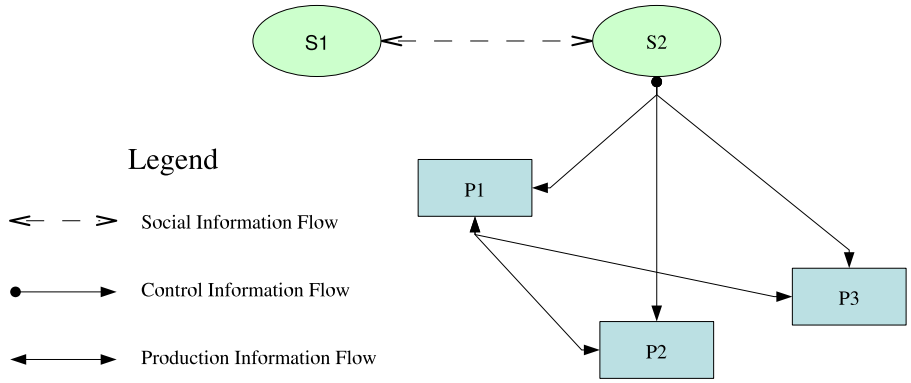


Figure 3-16: A simple company.

3.4.4 The Market

When the Production Agents are controlled by different Social Agents, their services and/or partial control can be exchanged in a market. In this case, already partially

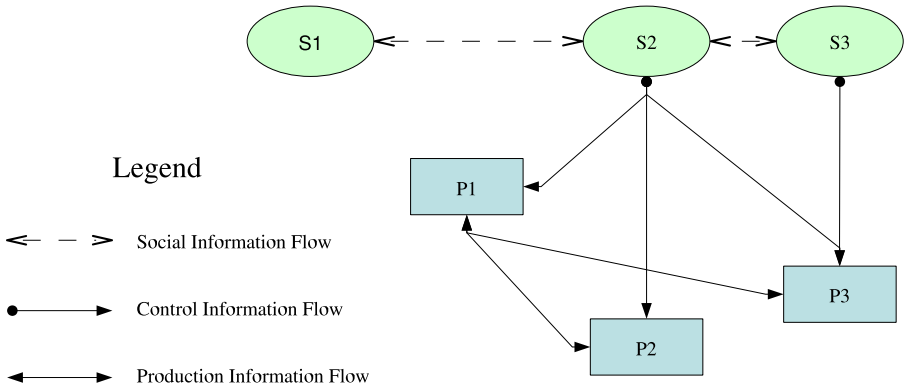


Figure 3-17: A company with outsourced services.

illustrated in Figure 3-17, a Social Agent engages in exchange with other Social Agents to create the production topology it needs for the services it wishes to produce. This control topology is illustrated in Fig. 3-18.

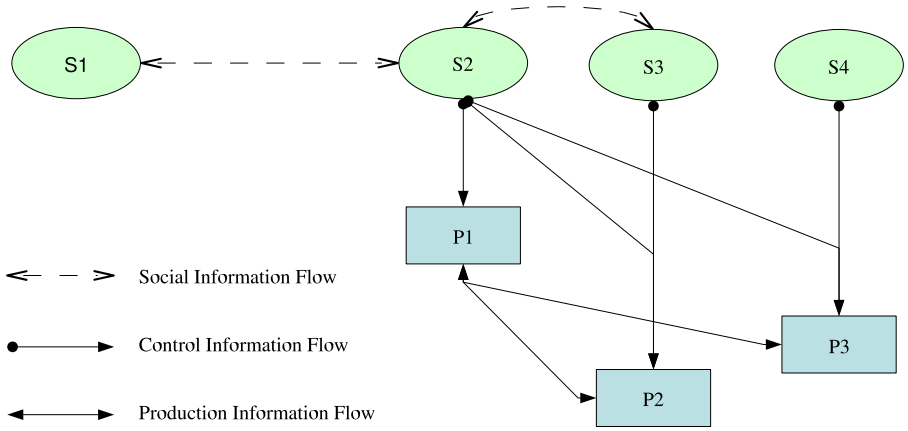


Figure 3-18: A Market.

3.4.5 Complex production topologies

Finally, the two control capabilities, absolute and relative, described in sections 3.4.1, 3.4.2, 3.4.3 and 3.4.4 can be combined in topologies of a higher complexity. In Fig 3-19 a client *S1* asks for a complex service, the execution of which requires different grid resources (*P1*, *P2*, *P3* and *P4*). These resources can represent different job submission systems and a work-flow engine or, in other cases, storage devices and computational power. The Social Agent interacting with the client directly controls

only grid resource P1 through one of its Production Agents. When the Social Agent receives the request from the client it checks if the other Social Agents, with which it has a relation, are able to fulfill the needed services. If so, a social arrangement is made to allow the Social Agent to gain control of the Production Agents for the necessary time. The Social Agent then instructs its Production Agent about the other different Production Agents it can use for the completion of the complex service. The service is then executed and the result is sent back to the client.

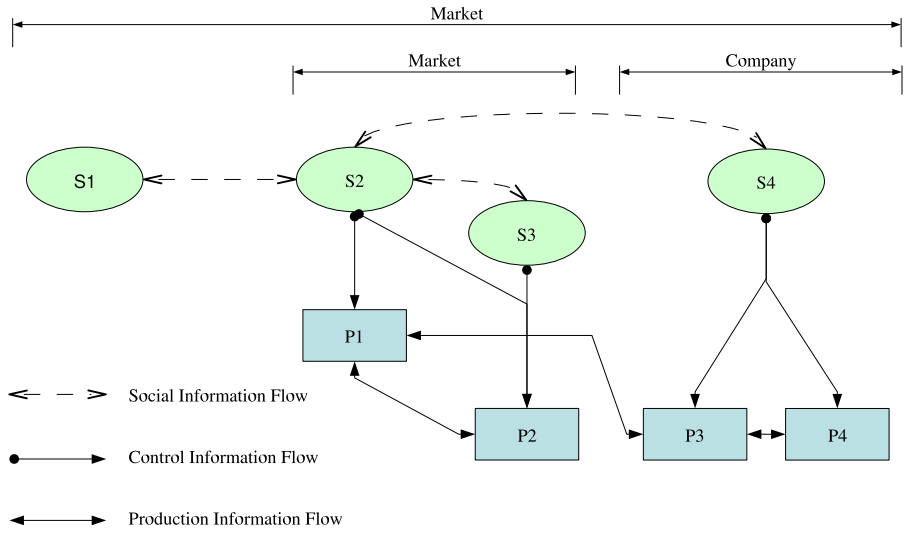


Figure 3-19: A complex production topology.

3.5 Social Topologies

In production topologies Social Agents control Production Agents. A similar process happens between Social Agents. We term these *Social Topologies*. Whereas in the production topologies above the issue of pricing does not arise, in social topologies it does.

3.5.1 Simple Relationship

In this social topology (also termed as *Simple Purchase*), see Figure 3-20, an owner or controller sets the pricing policies of a Social Agent. The Social Agent then responds to the requests of its clients accordingly.

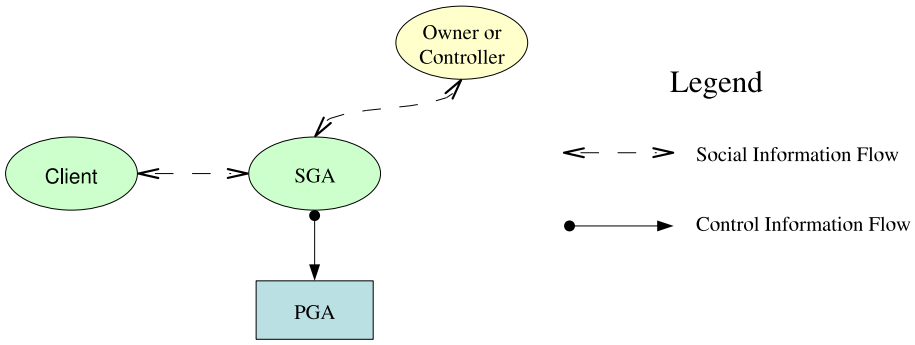


Figure 3-20: Simple Relationships.

This topology can be extended along the three additional dimensions described in 3.2.7. Payments between the two social agents can be performed through a bank, the requester may find the provider through an indexing agent and, finally, trust between the two can be achieved through the delegation of the price's definition to a third party, the arbitrator that is trusted by both as detailed in 3.7.

3.5.2 Tribe

In a *tribe*, shown in Figure 3-21, the Social Agents accept the pricing and access policies of the tribe (usually free or discounted prices for all the members of the tribe). Then all Social Agents can use part of the resources of the tribe to fulfill their needs. I call this topology a Tribe as its allocation mechanism is similar to that of a Tribe where all active members to the society share part of their resources with the other members in compliance of the rules dictated by an authority.

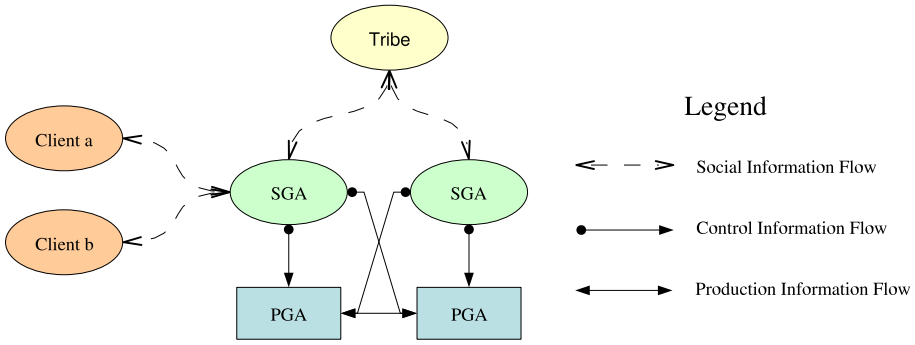


Figure 3-21: Tribe.

3.5.3 Pub

In a pub social topology, see Figure 3-22, two or more Social Agents agree to share part of their resources. This topology is similar to the tribe but it is not centralized as the agents can define private sharing modalities that do not need to be approved or defined by the other members. I call this topology a Pub Topology as its allocation mechanism is similar to that of the behaviour of friends that go regularly to a Pub. If one of the friends lacks the money the others will gladly pay for his consumption under the assumption that the same would be done for them. If one of the parties consistently fails to help the others for a certain period it is usually not granted help any more.

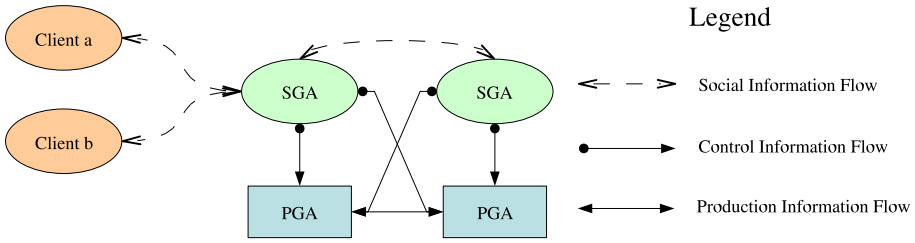


Figure 3-22: A pub model.

3.5.4 Keynesian Scenario

In a Keynesian scenario, such as that in Figure 3-23, the Social Agents accept pricing and access policies from an authority; these pricing and access policies usually refer to a subset of all the clients of the Social Agents. I call this topology a *Keynesian* scenario in a loose fashion without implying any direct link with the work or theory of John Maynard Keynes to describe a scenario where one of the actors (The Keynesian Authority)

3.6 Control and Ownership Topologies

Figures 3-24 and 3-25 describe two of the most common examples of ownership and control topologies. In Figure 3-24 social agent S_3 owns production agent P_3; following

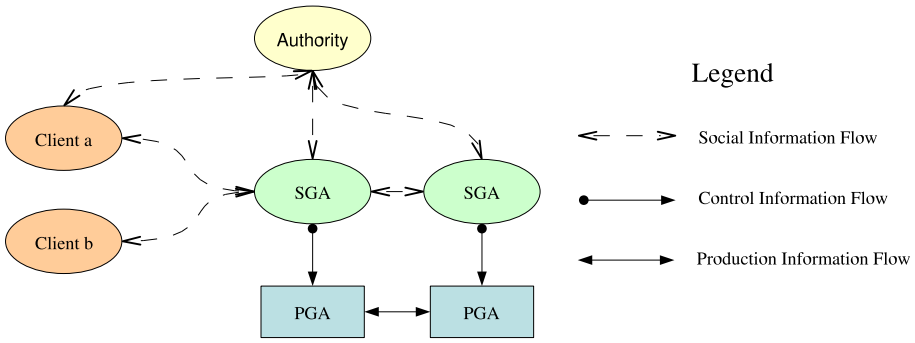


Figure 3-23: Keynesian Scenario.

a social transaction with social agent S_1, certain usage rights are transferred to social agent S_1 that is now capable of using P_3. Figure 3-25 describes a slightly more complex example where agent S_3 grants control rights on P_3 to S_2. Now S_2 can grant usage rights to S_1.

In the example of Figure 3-24, the owner S_3 will be able to set any modality to the usage relation granted to S_1 while in the example of Figure 3-25 the owner S_3 will be able to set any modality to the control relation granted to S_2. S_2 will, in turn, be able to grant usage modalities to S_3 that are compatible to its own control modalities.

3.7 Value and Price Topologies

The information flows required for *price* and *value* determination follow their own topology. Figure 3-26 illustrates where the concepts of value and price are used: price is used in the social layer, value is used both in the social and production layer while metrics encompass all layers. The value and price topologies define which agent determines those figures (and how). Figure 3-26 also describes the simplest of those topologies where all values and prices are determined by agents that belong to the same ownership and control topology.

This simple, *"vertical topology"* can be further divided into *top-down* and *bottom-up* depending on the main direction of the flow of information. Figure 3-27 describes these two different approaches. In the *top-down* approach described in the left part

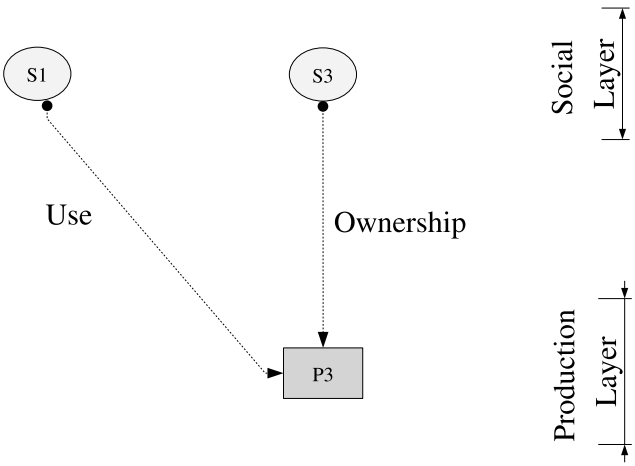


Figure 3-24: Example of control topology: simple use.

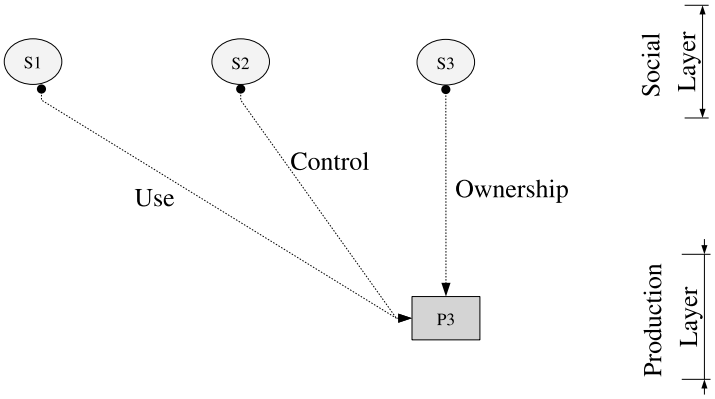


Figure 3-25: Example of control topology: rent and simple use.

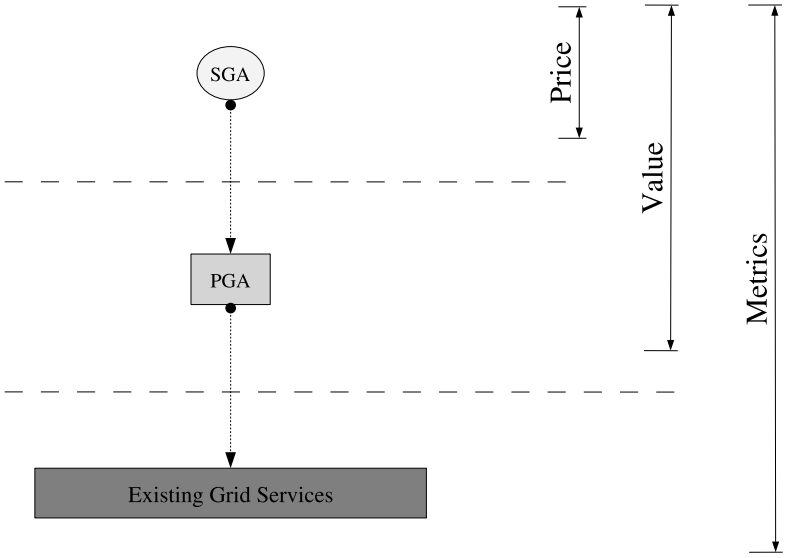


Figure 3-26: Metrics, Value and Price in the different layers.

of the figure, the Social Grid Agent determines arbitrarily the value of the production resources it controls and their price. This can be formalized as:

$$value = v_k \tag{3.16}$$

$$price = p_k \tag{3.17}$$

where v_k and p_k are fixed parameters determined by the Social Grid Agent.

The *bottom-up* approach is more complex and flexible. In it the Social Grid Agent determines the function to compute the price from the value and the metrics and the function to compute the value from the metrics in a *top-down* fashion but there then occurs an opposite *bottom-up* flow of information that provides the parameters of the functions. This is described in the right part of the figure and can be formalized as:

$$value = v_f(\{m\}) \tag{3.18}$$

$$price = p_f(v_f(\{m\}), \{m\}) \tag{3.19}$$

where $\{m\}$ is a set of metrics originated by the existing Grid resource and functions p_f and v_f are determined by the Social Agent.

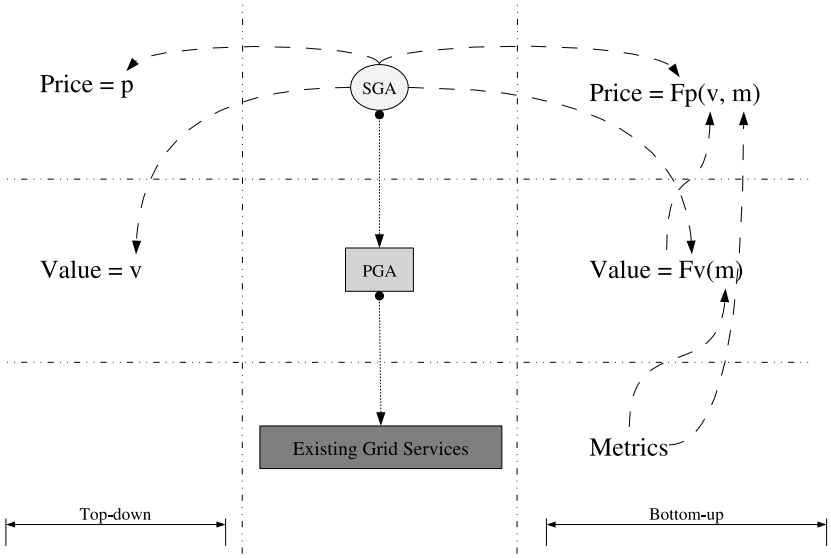


Figure 3-27: Different approaches to the computation of value and price.

A more flexible way to compute the value of a service is to define it as a function of two different sets of metrics:

- Service Metrics - These metrics provide information on the abstract service.
- Resource Metrics - These metrics provide information on the resources that have been used by a specific execution of a service.

Chapter 4 discusses how metrics can be combined to determine the value of a gLite job submission system.

In the simplest case, when trusted direct messaging exists between agents and resources, the functions are determined directly by the controlling agents, but more complex topologies arise when there is no trusted direct messaging between agents and resources, see 3.8.3 below.

3.8 Additional Social Dimensions

In section 3.2.7 we introduced the *Social Dimensions* of Banking, Indexing and Trusting. The following sections describe possible expansions in dimensionality to the topologies we have already introduced in this chapter.

3.8.1 Banking Dimensions

An exchange topology can be extended along the *banking dimension* if the involved parties delegate to a *Bank agent* the management of the financial part of their transaction. This extension is represented in Figure 3-28.

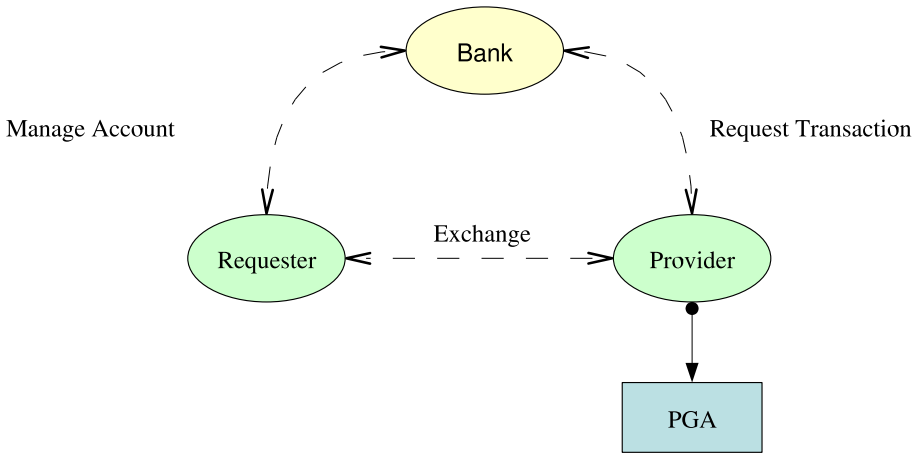


Figure 3-28: A banking extension of an exchange.

A bank extends a transaction between two agents by offering two main set of simple operations:

- *Account management* that allows agents to create accounts, and to define sets of agents that have the right to lodge and/or retrieve credits.
- *Transaction management* that allows agents (if they have the proper rights) to lodge and/or retrieve credits from an account.

This is a brutal simplification but it allows for the extension of exchange through simple steps:

- The *Requester* sets up an account and allows the provider to retrieve a certain sum of credits.
- The *Provider* retrieves the credits through a transaction.

3.8.2 Indexing Dimensions

A social topology can be extended along the *indexing dimension* if a party uses an *indexing agent* to discover which provider is capable of offering the needed services. This extension is represented in Figure 3-29.

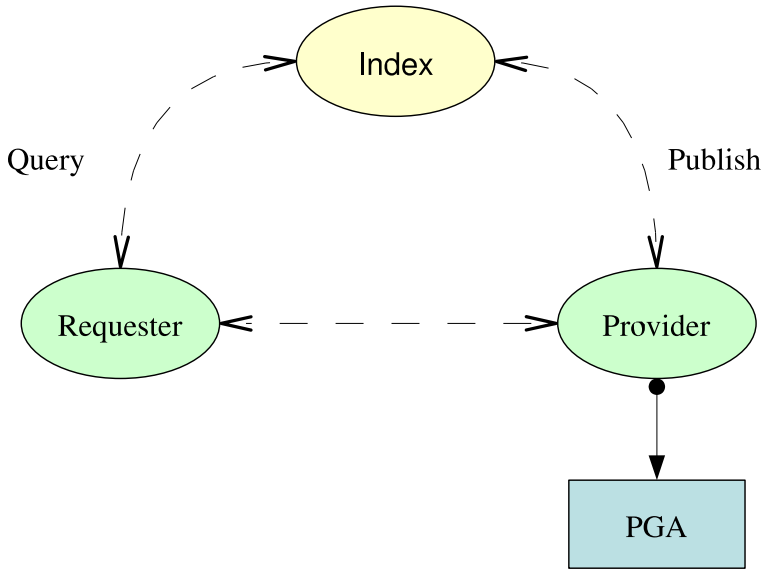


Figure 3-29: An indexing extension of an exchange.

An index can extend a relation between two agents by offering two main set of simple operations:

- *Publish* in which the Provider publishes part or all of its *description* (how providers describe themselves and their services is detailed in Section 3.18) in an index.
- *Query* that allows a Requester to find a suitable Provider.

3.8.3 Trusting Dimensions

A social topology can also be extended along the *trusting dimension* to supply trust where it is lacking. For example, in the specific case of value and price let us consider agents between which there is no direct trust link, then problems such as the perception of the *"fair price"* of a service or the need of arbitration in case of disputes arise. Clients may not be willing to pay for the execution of failed jobs while service providers may require that the resources being used are to be paid for in any case.

If a third party called the *arbitrator*, shown in Figure 3-30 is trusted by both the client and the service provider, then it can conduct an independent analysis (e.g. it can define value and prices and enquire into the status of jobs) to resolve the dispute. In this case, we say that there is an implicit trust link between the clients and the service providers through the arbitrator.

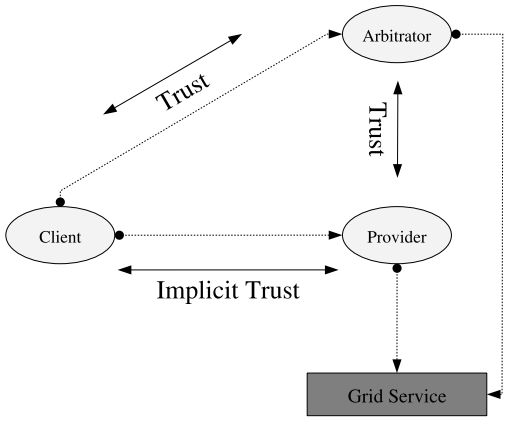


Figure 3-30: Different approaches to the computation of value and price.

3.9 Complex Topologies

By introducing the additional social dimensions detailed in 3.2.7 we allow the implementation of more *Complex Policies*. The first problem we must try to solve with

these complex policies concerns how to "close" the loop of a Grid economy shown in Figure 3-31. Some examples proposed so far are (excluding co-operative relationships), in fact, *open economic systems* where two different flows cross each other: that of services and that of credits. If in such an open economy the agents that provide services also consume services and thus *providers* are also *consumers* the original credits keep circulating in the topology. But if there are pure providers and pure consumers the credits will accumulate somewhere and, eventually, the entire system will cease to function.

Let us consider two possible solutions to this: a *Synthetic Macroeconomic Topology* (illustrated in Figure 3-32) and a *Connection Topology* (illustrated in Figure 3-33) that connects the Grid economy with the real one.

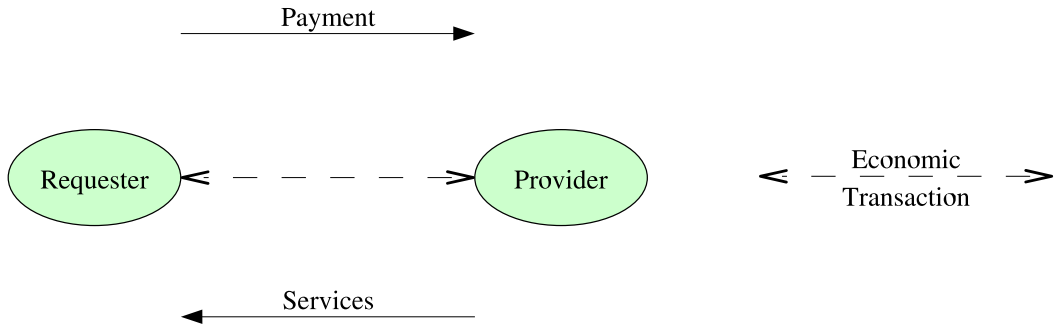


Figure 3-31: Open and closed economies.

In a synthetic economy the circulation of credits can be ensured by three social actors: an *index*, a *mint* and a *bank*. All production agents sign to an index and publish information on the resources they control. This information is used by the mint agent that issues and distributes credits in the different bank accounts. Finally, these accounts may be used for the micro-economic transactions.

Another possibility (as yet just a theoretical approach) to redistribute credits among the different agents is to link the Grid bank with the real economy, allowing exchange of real currency and Grid credits. In this case the resource owners will be able to exchange the Grid credits into real currency that will be in turn be granted to the requesters for exchange into Grid credits.

Of the two, this thesis focuses on the first as the ramifications, the feasibility and

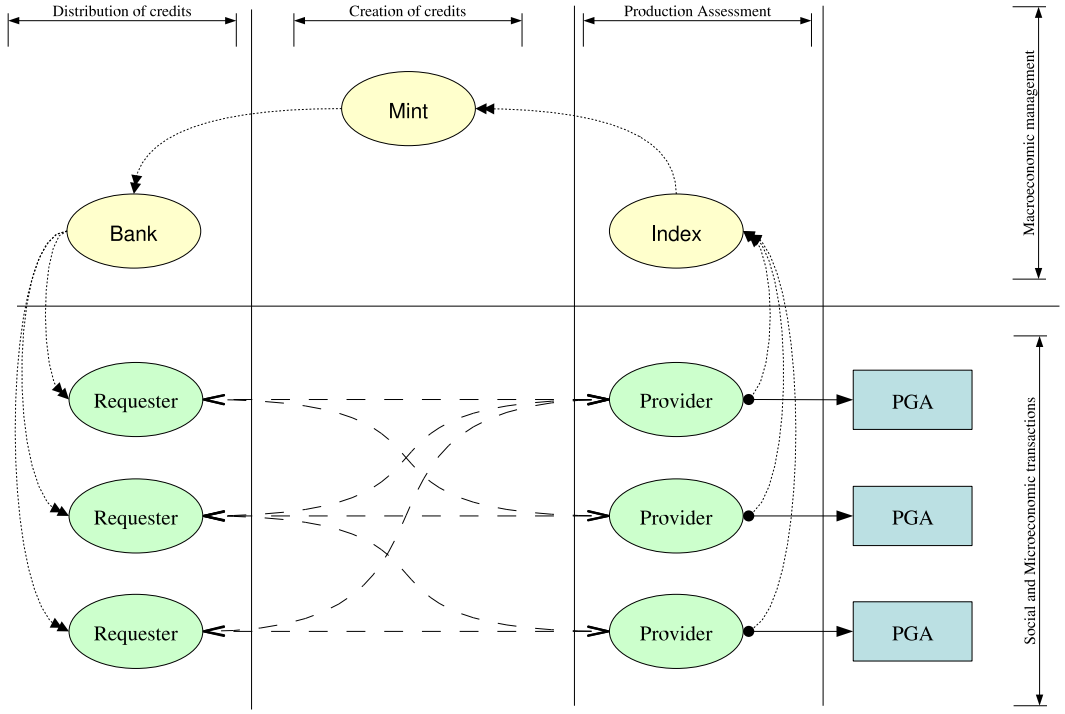


Figure 3-32: A closed, synthetic economy.

even desirability of directly[3] connecting a Grid economy to a real economy (and, if the scale become significant, even the reverse) are yet to be investigated.

3.10 A Metagrid Paradigm

Although the concept of interoperability could have different interpretation in the Grid Community, I use the term interoperability in two different senses. The first and more strict one is the capability of a system to interface and use different middlewares, in this sense the metagrid prototype implements this view of interoperability. A broader acception of interoperability as the possibility of different middlewares to use each other seamlessly is at the basis of the metagrid abstraction but is not fully implemented as of now.

This thesis explores a *metagrid* abstraction for interoperability, and exploits this as an example environment for SGAs.

[3]Grid economies and real economies are already *indirectly* but strongly connected as hardware, electricity, wages and rents have to be paid for in real currencies.

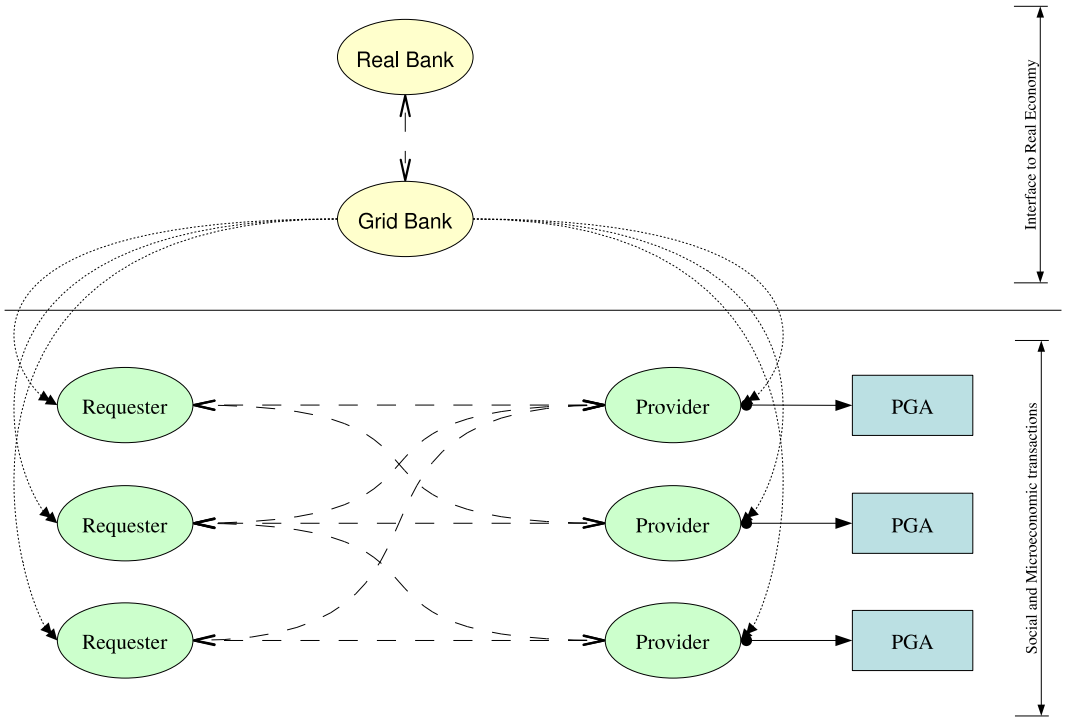


Figure 3-33: Connection between a synthetic and real economy.

If we imagine that the different Grids occupy segments of a space, then the remaining space can be conceived of as a generic interoperability space. Since some workflow engines are *users* of Grids, not an intrinsic element of any one Grid, it is useful to conceive of these as distinctly different entities, yielding three distinct spaces: Grids, workflow engines and an interoperability space. This chapter proposes a metagrid paradigm that is an attempt to put the user at the centre of such an interoperability space, the *metagrid space*. This means that each user should have access to all available Grids and workflow engines.

3.10.1 An abstract view

A metagrid may be simply represented as a Venn diagram as in Figure 3-34. Let us define a *region* as a set of resources available to the user. Let S be the set of all the possible resources available to the user where $R_m \subseteq S$ is the metagrid region. R_w is the family of workflow regions. R_a is the family of ancillary regions containing services such as banking or accounting that, although not necessarily directly linked

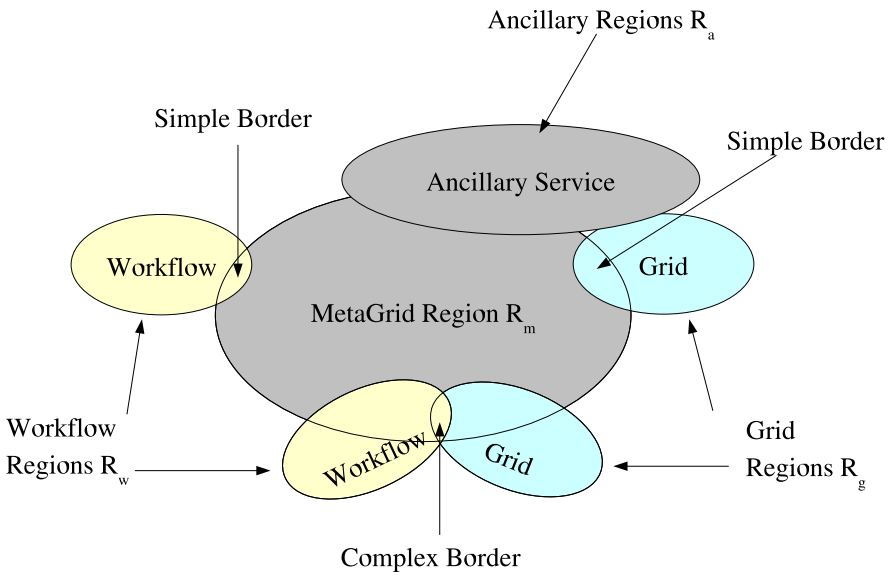


Figure 3-34: Abstract view of a metagrid.

to a specific grid middleware, might be needed by the metagrid services. And, finally, R_g is the family of Grid regions. Of course, unless so provided, none of these regions are capable of interoperation. One insight is that each Grid or workflow region may be extended with *dual-mandate* resources that interface to the metagrid region. Resources that serve one mandate only are within *internal* regions. Resources that serve two or more mandates are within *border* regions. While they need not be, it is helpful to think of these resources as machines. Thus an internal region might be a set of machines that host a homogeneous set of either metagrid, workflow or Grid technologies. R_m is a pure metagrid region where only metagrid technology exists. R_w is a pure workflow region (WebCom, for example), R_g is a pure Grid region (gLite, for example). Border regions not only separate technology domains but also policy domains; therefore border regions must also cope with policy compatibility in addition to technology compatibility.

Communication between the metagrid and a Grid or workflow middleware flows through the border regions, where two or more technologies coexist. Where only two technologies coexist these are called *simple borders*. Where three or more technologies coexist these are called *complex borders*.

The abstraction of border regions is very useful. For instance, when communicat-

ing between two technologies, each of which has simple borders to the metagrid, the communication is said to traverse an *extended border*, e.g. extended border supports traversal from the workflow region via a simple border to the metagrid region and thence via another simple border to the Grid region. On the other hand, with complex borders, communication between any two of the technologies hosted by one complex border is said to traverse a *collapsed border*, e.g. collapsed border supports traversal directly from the workflow region via a complex border to the metagrid region and to the Grid region. The advantage of the former is simplicity, while the advantage of the latter is traversal via only one complex border rather than two simple borders.

The metagrid region R_m could, in the first place, offer a friendly and flexible interface towards the user, and could allow relatively uniform access to different Grid and workflow middlewares and services. If the user I/O is moved from other spaces to the metagrid space, then the Grids could be enabled to focus on their strengths in pure computation and storage, and likewise the workflow engines to focus on their specific strengths. Each interface to a user could conceivably serve all Grids and workflow engines. The net effect would be that for N Grids, any particular I/O model would need just one implementation within a metagrid, rather than N implementations, one per Grid.

A set of *views* can encapsulate the underlying ideas and assumptions of a metagrid. The first of these is of the metagrid as a *"Grid of Grids and Services"*. This view is implicit to the discussion above.

A second view is that of a metagrid supporting different computational models. While it is not possible for the metagrid to change the computational model of Grid and workflow middleware, it should in principle be able to encompass middlewares that support different computational models and, especially, to be able to express the orchestration and interaction of middlewares with different workflow models. Special attention would need to be given to those workflow engines (such as WebCom [60]) that are able to execute complex workflows that embed multiple computational models.

A third view is of a metagrid as a bridge between the different standards employed

by the multiple Grid and workflow middlewares. The choice is a challenging and interesting dilemma: *abstract or translate*; to translate a standard into another or to create an abstraction layer able to understand all the other standards. Both approaches have advantages and disadvantages. A translation approach forces one to write a large number of modules to translate each standard into another while an abstraction layer is often an excessively complex generality. The solution this thesis proposes is an hybrid: an *exchange approach*, that adopts an *official standard* (that may already exist); *exchange services* can then translate from the official standard to the different supported standards. An optimization is clearly available if the middlewares are already interoperable.

The final view is that of a scalable metagrid. The exchange approach allows the number of translators to grow linearly rather than quadratically with the number of standards supported, and thereby simplifies the design and implementation of the abstraction layer. Similarly, moving the I/O to the metagrid space allows the number of I/O solutions to grow linearly rather than as a product $M \times N$ for M I/O models and N Grids. These two attributes guarantee that a metagrid is scalable.

3.10.2 A concrete view

Figure 3-35 illustrates a concrete example of a metagrid architecture that has been implemented as an experimental prototype.

The communication among all the metagrid components is provided by the Metagrid Transport Layer. This component of the architecture is composed from two sub-layers: the Control Transport Layer, which conveys all *lightweight* information, and the Data Transport Layer, which conveys all other information. We use the term *lightweight* to mean all the information that does not need a file to be communicated; an example of *lightweight* communication is the information passed between web services in a distributed architecture.

The Metagrid Transport Layer allows the communication between three main groups of metagrid services. The first group is the *Metagrid Border Services*, interfacing all the existing middlewares that are to be encompassed by the metagrid

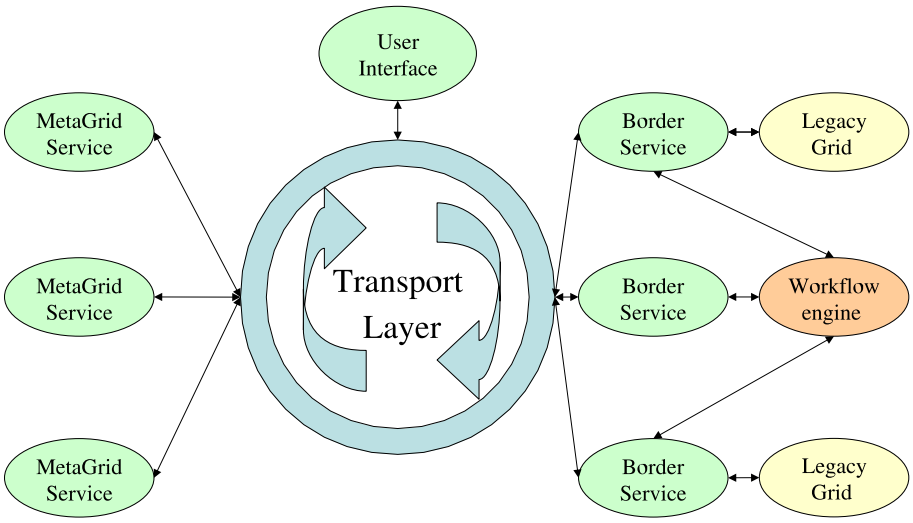


Figure 3-35: A concrete view of the metagrid

architecture. As in Section 3.10.1, these middlewares are divided into two regions: workflow middleware regions R_w such as WebCom, and Grid middleware regions R_g such as gLite and GT4. Each of these regions is extended with a border region R_{mw} or R_{mg} that hosts border services that allow communication with the metagrid transport services.

The second group is the *Native Metagrid Services* that comprises those services that were developed to tackle specific metagrid issues. Thus far only one of these services has been implemented: the metagrid Job Exchange (MJX) service that enables any supported middleware to target job-related actions to any other middleware supported by the metagrid environment.

The Transport Layer

The present Control Transport Layer is based on Globus GT4. Both extended and collapsed borders have been successfully created using GT4 technology.

The Job Exchange

The Metagrid Job Exchange (MJX) enables any supported middleware to target job-related actions to any other supported middleware. These actions include job submission and status requests. The actions are invoked via the appropriate border

region. It is the duty of this service to try to submit a job, described in the job description language of one middleware, to any other middleware. This is likely to involve a translation from one language and format to another. As an example, the MJX may be requested to submit a job described in gLite's JDL [68] to a GT4 Grid that requires a description in RSL [93]. As not all job description languages have the same functionalities some translations are impossible if all the information must be preserved. But a translation is possible provided that it is acceptable that some information is lost during the translation process. As an example, consider a job expressed in the JDL language that explicitly defines computational requirements and ranking criteria for finding an optimal resource. If it is acceptable that this ranking information is lost, this job description can be successfully translated into a less expressive job description language and the job executed in a grid middleware different from the one it was designed for.

3.11 Social Grid Agents and Metagrids

Social Grid Agents and metagrids are mutually beneficial paradigms: in a metagrid SGAs can, amongst other things, perform two major roles: *border agents* or *native metagrid agents*.

3.11.1 Border Agents

In borders the SGA capability of expressing and enforcing different policies helps to allow the co-existence of different middlewares. SGAs prove their usefulness in two main respects: they can act as standards bridges and they can act as policy enforcement points for those responsible for the existing middlewares. Border agents can translate different standards. While these translation capabilities of the agents are mostly useful at a production level, the social level of the agents is useful in describing the ownership and control topologies of the different middlewares allowing those responsible to define suitable policies to reflect the agreed co-operation levels. As an example, a middleware requiring complete traceability of the issuer of the job

will instruct its border agents with appropriate authorization and delegation policies whilst middlewares with a different security policy might allow border agents to submit jobs without bothering to fully trace back the delegation chain. In this case, a control topology of a production border agent can be used to trace areas in the metagrid space where different security policies co-exist.

3.11.2 Native Agents

On the other hand, inside the metagrid space, Social Grid Agents can be used to enforce the policies required by the border agents and to control ancillary services such as indexes, markets and exchanges. The metagrid space presents obvious opportunities for exploring the subject of this thesis.

3.11.3 Discussion

The use of a metagrid paradigm as a candidate environment for Social Grid Agents offers mutual benefits: as a way to tackle the problems of interoperability (that, although it is not the focus of this thesis, is a closely related issue) and as a meaningful and natural environment for testing the behaviour of Social Grid Agents.

The generic nature of SGAs finds a natural complement in a metagrid environment where its native language (see Section 3.18) can be translated into different dialects of grid middlewares by border agents. This capability allows for the definition of policies (and resulting behaviours) that are not tied to any particular middleware but that can be translated (sometimes accepting the loss of some information) into the specific languages. The adoption of a native language implies that the number of translating agents grows with N, where N is the number of supported middlewares, and not with N^2.

Social Grid Agents also allow the implementation of higher level functionalities in the metagrid space, both for pure interoperability purposes such as the Metagrid Job eXchange (see section 3.10.2), and for supporting different social and economic models across different middlewares.

An interesting feature of this view is that the production layer is a border region

with the existing middlewares, and Production Grid Agents are *wrappers* of the different services; this allows for scalable expansion of the encompassed middlewares as well as a single point of translation.

However, the essential point is that a metagrid is a good candidate environment for SGA prototyping.

This chapter is devoted to the description of the architecture of Social Grid Agents. The chapter starts with an Introduction (section 3.12) describing features and constraints; a taxonomy of the agents is in section 3.14, and a general description of the overall architecture of the agents is in section 3.15.

3.12 Architecture

In order to implement the features described in Section 2.4, the agents will have to compose services arranged in a production chain. This production process must be defined in a flexible way to allow the creation of different supply chains depending on different optimality criteria and social and economic relationships with other agents. Agents act as nodes that connect different information flows (production, control/ownership, social and price/value) and react to their changes; they must manage and compose services and they must interface with existing ancillary services such as accounting or banking systems. Agents must also manipulate and react to social information flows that describe their relationships.

As grids are developing apace and yet, despite the efforts of the scientific community, a clear framework for standardization is still to be defined, it is important for the agents to be able to inter-operate with other technologies, and even be capable (at least at a theoretical level) of inter-operation with services that are still to be developed or even designed.

As all but an initial set of agents have yet to be developed, tested and deployed, there also is a very strong need for a flexible and simple architecture and the possibility to develop the code in small, incremental steps to control the overall complexity of the solution. The necessity for simplicity rules out traditional software agent technology. These considerations, although not directly related to functional specifications,

play a significant role in the design of the architecture.

These constraints can be met by a lightweight architecture based on associative relations and agents that expose some plug and play behaviour. Agents should be able to manage information relating to the association with other agents, they should be able to find "optimal" configurations of services based on a flexible set of data and, finally, they should be able to have some degree of "self consciousness" by which to make decisions based on historical records.

3.13 Abstract Architecture

An abstract view of the architecture is as a set of *Message Transformers* that communicate and compose services. The message transformer is defined as an entity which exposes a minimal interface. Transformers can be further sub-divided into sub-entities:

- *Providers* that simply process a message; they constitute the simplest tile of our architecture. We use a provider interface each time we want to wrap the message-based SGA architecture around a service. An example of a provider can be a wrapper around a job submission system or a piece of code capable of performing some clearly defined operation.

- *Processors*: that map messages into sequences of messages that unfold in time; a simple example of a processor is a workflow engine. Using a very informal description we say that processors map *messages* to *messages* in time.

- *Managers*: that map *messages* to *providers* in space; managers are basically indexing services.

Let us define input and output messages as m_{in} and m_{out} and the provider's transforming function as $f_{pv}(...)$. We can describe a transformer as a transforming function such that :

$$m_{out} = f_{pv}(m_{in}) \tag{3.20}$$

If we define a sequence of messages in time as $M = \{m_0, m_1, ..., m_i\}$ and the processor's transforming function as $f_{pr}(...)$, we can describe a processor as a transforming function such that: $M = f_{pr}(m_{in})$.

$$M = f_{pr}(m_{in}) \tag{3.21}$$

Finally, if we define a transformer as t, then we can describe a manager as a transforming function such that:

$$t = f_m(m_{in}) \tag{3.22}$$

These three sub-entities: providers, processors and managers are composed to create structures of the required complexity and sophistication. A fourth concept, *the agent*, describes a set of message transformers that share an identity as described in Figure 3-36. Agents themselves are transformers and expose the same behaviours: processing agents behave as workflow engines orchestrating complex services, managing agents offer indexing services such as markets or yellow pages whilst agents wrapping existing services such as simple job submission are pure provider agents.

3.14 Types of agents

Social Grid Agents can be divided into three main groups along two dimensions. The first dimension relates to the connection with existing grid services, where there are two groups of agents: *Border Agents* and *Native Agents*. The second dimension relates to the nature and role of the agents; here we identify *Social Grid Agents* and *Production Grid Agents*. This two dimensional taxonomy of the agents is represented in Figure 3-37.

As illustrated in Figure 3-38, social agents of both border and native nature share the same architecture and are therefore grouped together. On the other hand, production agents are divided into different border and native architectures depending whether they produce a service based on existing grid services or not. Existing grid

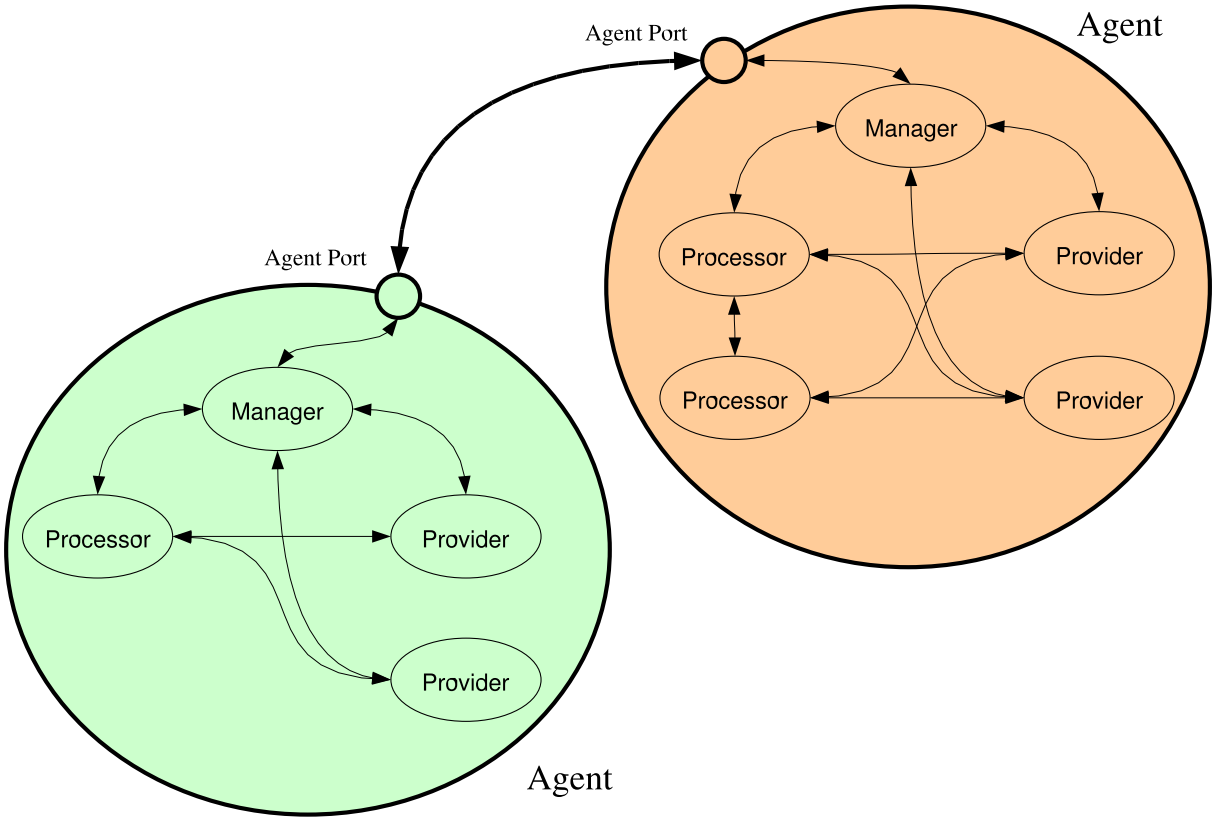


Figure 3-36: Abstract architecture of the agents.

services are interfaced with the agents via what is termed a *Service Provider* and constitute what we term a *Grid Body*.

3.15 Agent anatomy

A high level view of the architecture of the agents is described in Figure 3-39. It is composed of two or three main entities.

Agents are surrounded by a *Native Service* which enables them to be invoked through a *Native Service Port*. If a link exists between an agent and an existing grid service it usually consists of API invocations but can also be performed with system calls if an API is lacking. All other agents contain a *Native Skeleton* that consists of the services offered by the Native Service. Finally, an *Agent Behaviour Engine* implements the production or social behavior. Agent Behaviour Engines are software infrastructures that host *Agent Behaviour Policy*. The behaviour and the

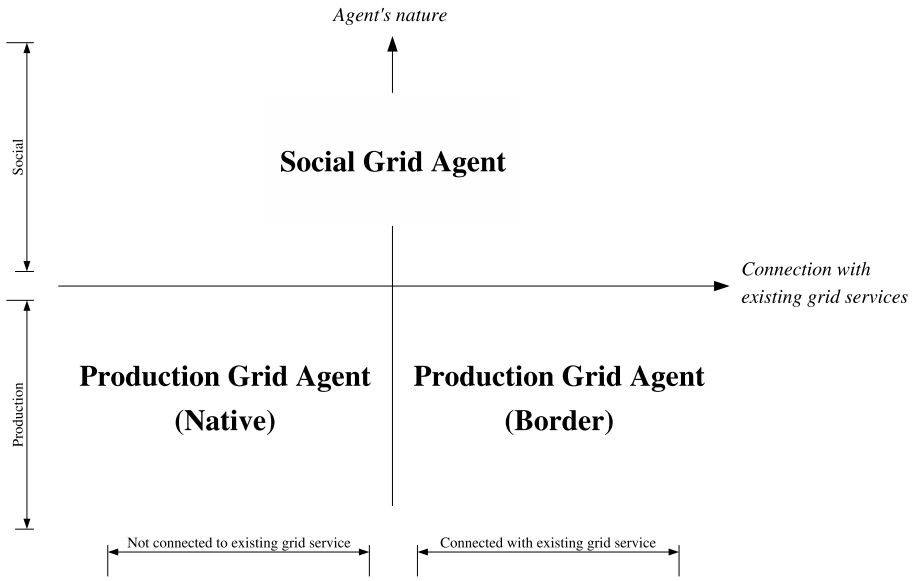


Figure 3-37: Taxonomy of grid agents.

decision mechanisms of the agents are expressed in a *Native Language* detailed in Chapter 3.18. Finally, all agents and service providers expose a minimal interface composed of a single method *process(message)* that returns another *message*. The agents are simple message transformers, where the transformation is defined in their native language. This design decision was made in order to reduce to an absolute minimum the complexity of the agent's skeleton and to allow all agents to expose the same *stub*.

This architectural feature also facilitates the outsourcing mechanism that is the very foundation of most of the agent's behaviour; so that an agent may use the grid services it directly controls (see Figure 3-39) or it may access them through another agent willing to allow this (see Figure 3-40). In the latter case agents a and b can use the services offered by both service providers provided that the agent in control allows so, and the very simplicity of the agent architecture expressly facilitates this.

3.15.1 Messages

Agents communicate through messages; in order to understand the agent's architecture it is important to understand that of the messages. Messages are classes that

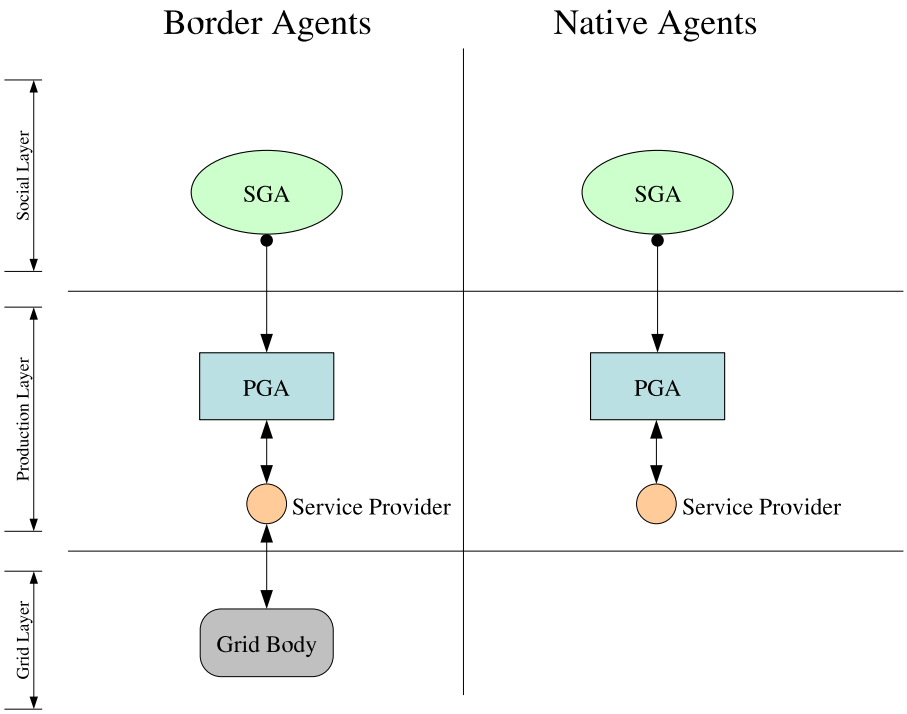


Figure 3-38: Social and production agents.

implement a *Serializable* interface in order to be sent and received through the *Native Service Port*. Messages reflect the division into layers as described in Figure 3-41. Social agents communicate with each other through *Social Messages*, production agents accept and send *Production Messages* and, finally, service providers communicate through *Service Provider Messages* or, for the sake of conciseness, *Execution Messages*. Messages contain a description expressed in the native language and possibly attached data in a service-specific format.

As production agents control service providers and social agents control production agents, the messages they exchange reflect this structure as illustrated in Figure 3-42.

Execution Messages can contain the information needed to execute the various services, either by native agent code or by existing grid services or can convey information regarding the Service Provider to the Production Layer. This information can be condensed, filtered and enriched by additional production information when it is exchanged by production agents. Finally, production messages can join social

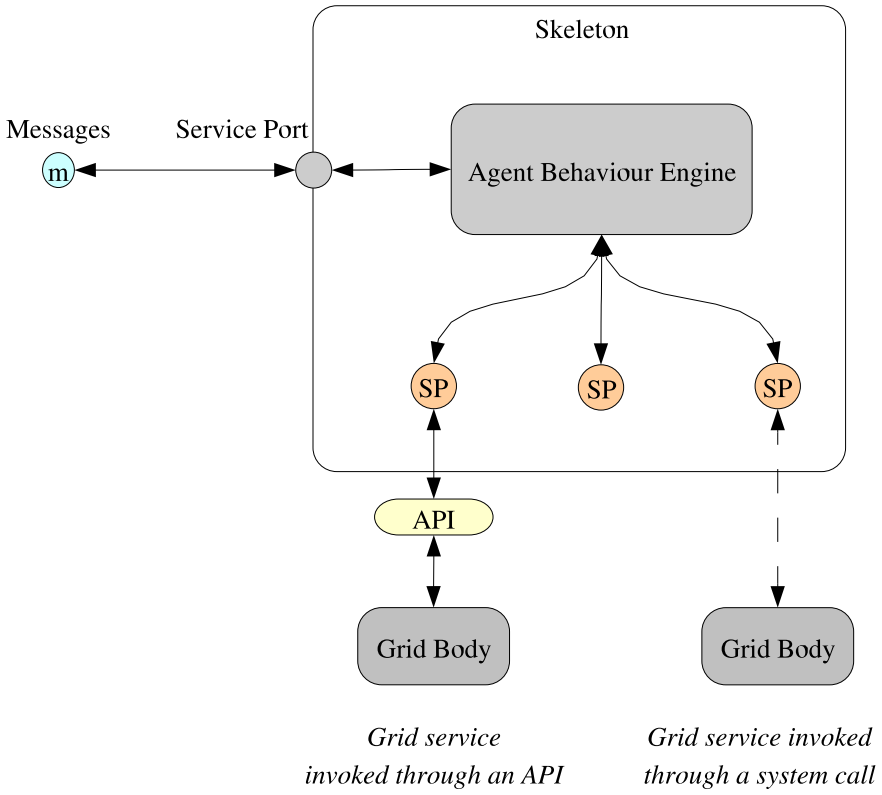


Figure 3-39: Agent Architecture.

information in social messages. In every layer there can also be messages that contain only information pertaining to that level like pure social messages, pure production messages and social messages containing pure production messages.

3.15.2 Service Providers

Service providers can be interfaces to existing grid services or to native services for the grid agents. In the first category there are interfaces to job submission systems such as the Workload Management System of gLite or the GT4 GRAM, or work-flow engines such as WebCom [60][58]. Messages received by the service providers are usually simple wrappers containing necessary information for the execution of a service. Service provider messages for a job submission system, for example, will contain information regarding the executable, input, output and other data regarding the job submission. On the other hand messages regarding work-flow submissions will contain the representation of the workflow to be executed.

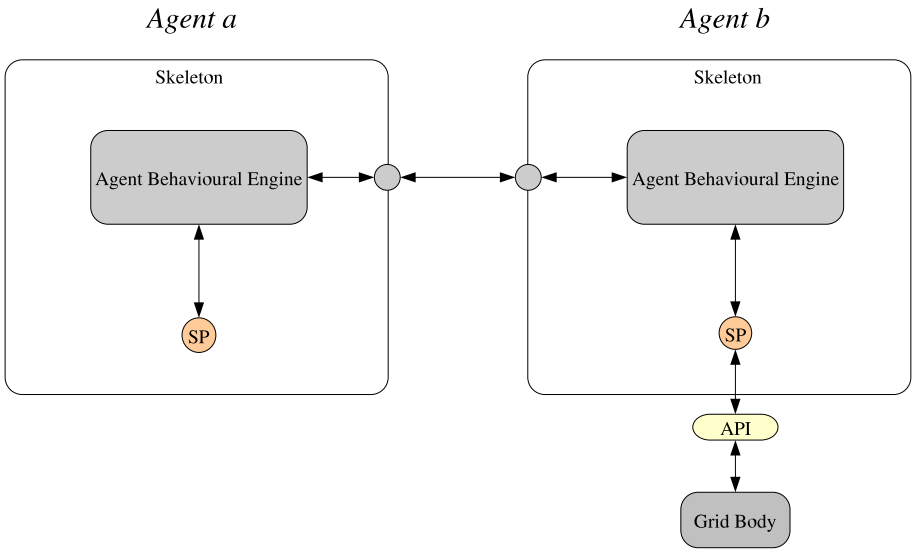


Figure 3-40: Architecture of the outsourcing mechanism.

Behaviour Engines

Service providers are managed by a component called the *Agent Behaviour Engine* described in detail in Figure 3-43. When a message is received by an Agent Behaviour Engine, its *Manager* checks whether there is any *Processor* (P_k in Figure 3-43) able to manage it. A processor is a class that contains the "knowledge" of what to do with one or more messages. The processor will execute a series of *actions* that can use other processors or service providers.

The manager is capable of match-making a processor to a message through a mapping function:

$Pr_k = F(M_{IN}, \{p\})$

where Pr_k is the k^{th} processor, $F(...)$ is the mapping function performed by the manager and $\{p\}$ are additional parameters.

The processor then analyzes the message and engages in a series of steps; when it needs external *providers*, it requests them from the *manager* (SP_A and SP_B in Figure 3-43) that instantiates them using the same mechanism as that which was used to obtain the processor. In this case: $Pv_l = F(M_{IN}, \{p\})$ where Pv_l is the service provider (possibly including processors), $F(...)$ is the mapping function performed by the manager and $\{p\}$ are additional parameters.

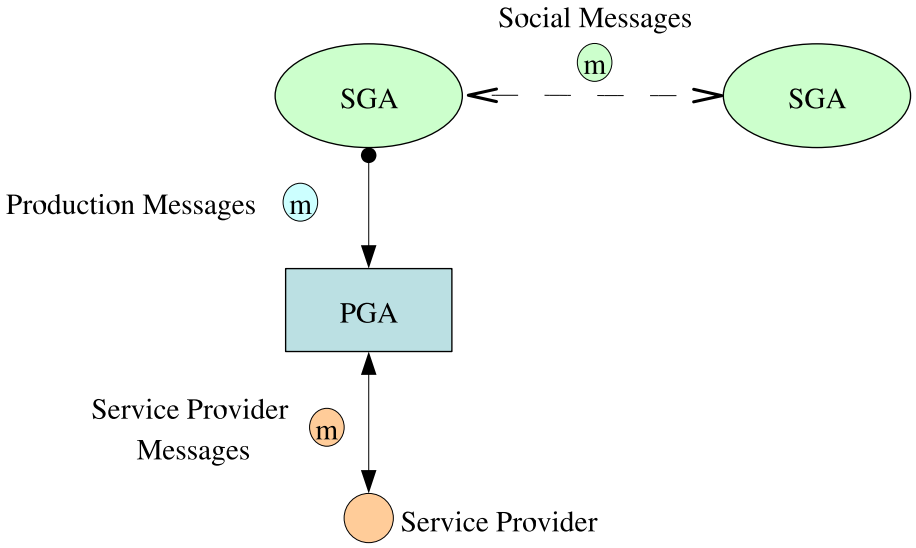


Figure 3-41: Different messages exchanged by agents.

The processor executes all the necessary steps and returns a message M_{OUT}. This we describe through processing functions:

$M_{OUT} = G_{Pr_k}(M_{IN}, F(M_{IN}), \{p\})$

where G_{Pr_k} is the processing function implemented by P_k (the k^{th} processor), F is the mapping function of the manager and $\{p\}$ are additional parameters.

The description of the overall behaviour then becomes:

$$M_{OUT} = G_{Pr_k}(M_{IN}, F(M_{IN}), \{p\}) \quad (3.23)$$

$$Pv_l = F(M_{IN}, \{p\}) \quad (3.24)$$

$$Pr_k = F(M_{IN}, \{p\}) \quad (3.25)$$

Where equation 3.23 describes the processing function returning the output message M_{OUT} and equations 3.25 and 3.24 are the mapping functions.

These maps are used to describe relations that link message types and message values including the identity of the requester (and sometimes the beneficiary) to policies and processors capable of handling them.

A possible implementation of this mapping technique is described in Figure 3-44. Firstly a message m is received by the agent from a sender s. The first step in the mapping process is to link the sender and the beneficiary to a relationship $(s, b) \rightarrow r$.

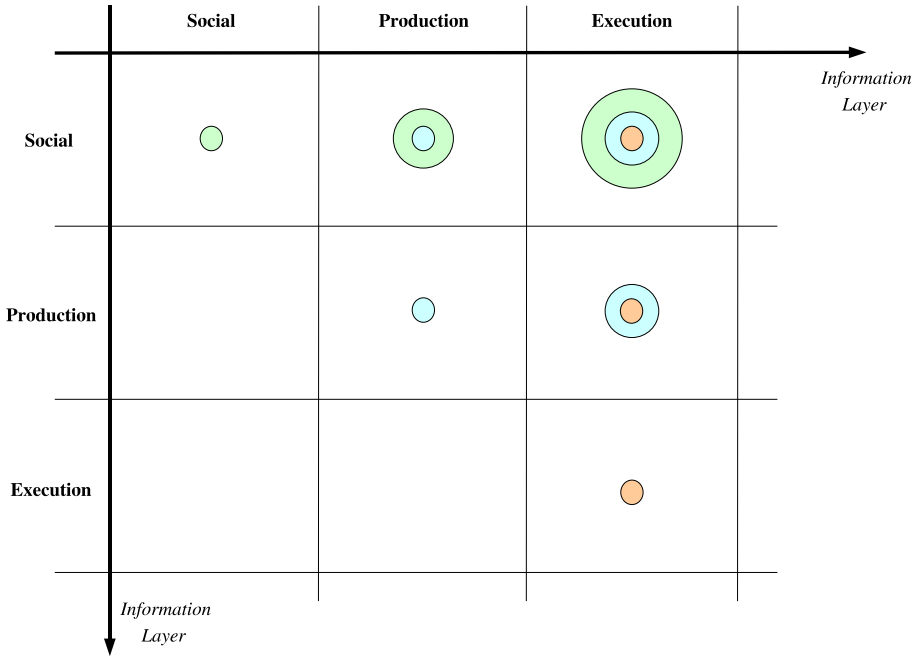


Figure 3-42: Different messages exchanged by agents.

Then the action requested in the message and the relationship are mapped to the set of granted policies: $(a, r) \rightarrow \{gp_i\}$. Then the modalities requested for the action and the set of granted policies are mapped to an enforced policy: $(m, \{gp_i\}) \rightarrow ep$. Finally the requested action and the enforced policy are mapped to a processor: $(a, ep) \rightarrow processor$.

Agents must take different actions depending on the sender's identity (and possibly the beneficiary's identity) and the message; for example an agent may accept service execution requests for free from an agent it has a cooperative relationship with, but may ask for a standard price for a different set of agents and also have a particular set of prices for a third set of agents with whom it has a particular economic relationship.

Processors

Processors are a key component of the Agent Behaviour Engines, they must be able to perform sequences of actions in response to received messages and they must also be able to react to messages whilst performing actions. In order to do so processors

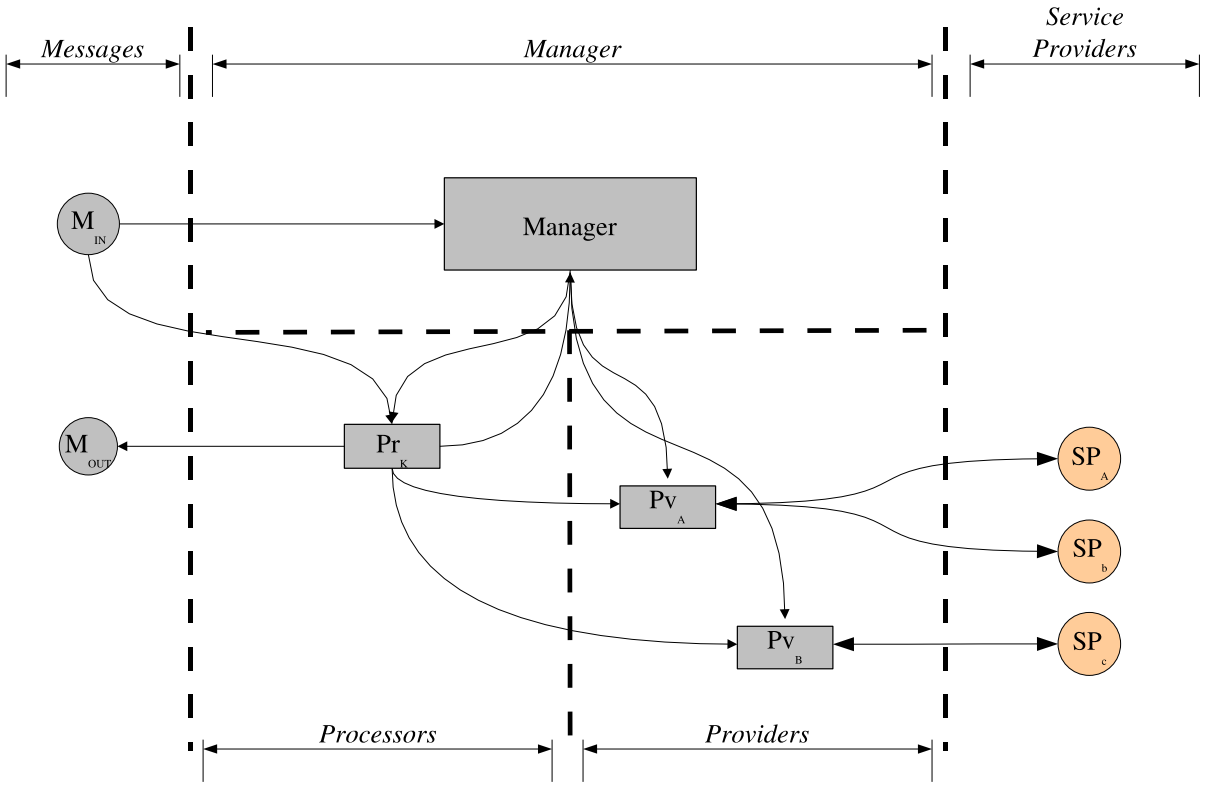


Figure 3-43: Agent Behaviour Engine Architecture.

contain a map based on the native language, which itself must be capable of mapping the current action, parts of the status of the processor and messages eventually received into the next step action, for example as in Figure 3-45. This map can be expressed as: $(A_i, m, s) \rightarrow A_{i+1}$ where A_i is the current action, m is the message received (if any) and s is a subset of the status of the processor $s \subseteq S$. In order to be able to react to messages whilst performing actions, processors are divided into two categories, synchronous and asynchronous.

Synchronous Processors

Synchronous processors react to messages with a pre-defined sequence of actions and return results at the end of this sequence. This behaviour is shown in Fig. 3-46. Once the processor receives the message M_1 it triggers one of the two chains of events $A_1 \rightarrow A_2 \rightarrow A_4$ or $A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_3 \rightarrow \ldots \rightarrow A_3 \rightarrow A_4$. Only at the end of these events is a message returned.

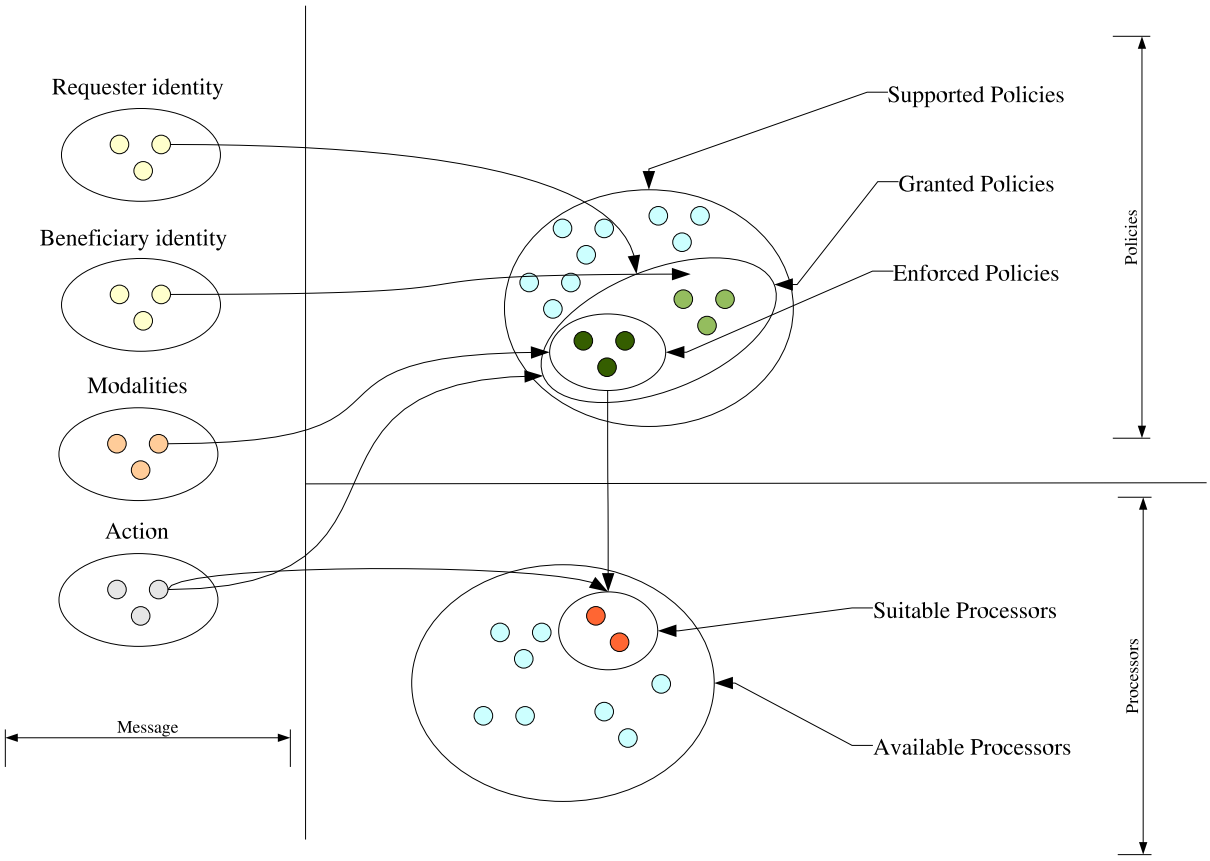


Figure 3-44: Mapping a message to a processor.

Obviously this behaviour is acceptable only if the sequence of actions can be executed in a short and predictable amount of time. Actions such as lengthy job submissions should not be implemented with a synchronous processor as the caller processes will have to wait (possibly for a significant time) until the processor has completed its sequence. This is already a good reason not to rely solely on synchronous processor but there are also two other reasons to develop asynchronous processors: timeouts and topologies. The native skeleton can exhibit timeouts that can easily expire if an invocation is not returned in due time; also, if agents wait long times for responses, they lock in chains of invocations that prevent the creation of many topologies. For these three reasons another type of processor, the *asynchronous processor*, is supported by the agents.

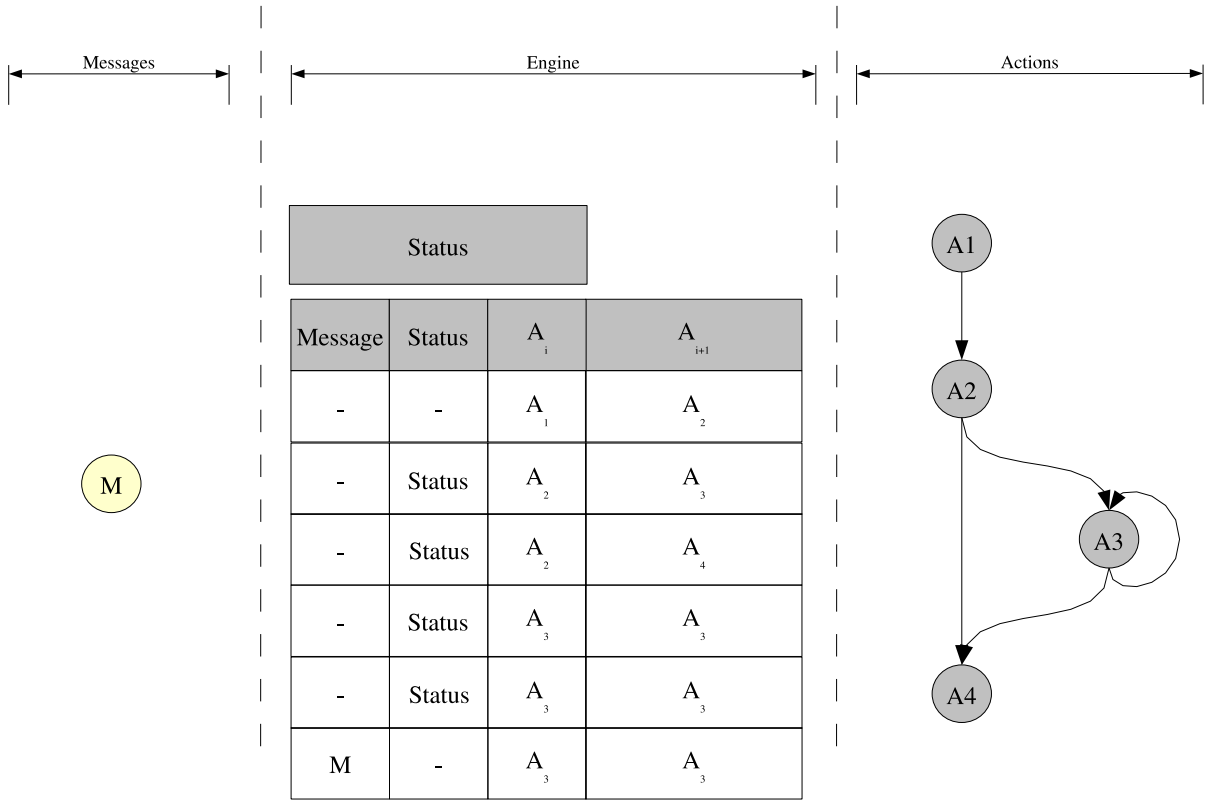


Figure 3-45: Example of steps performed by a synchronous processor.

Asynchronous Processors

Asynchronous processors inherit their behaviour from *a threading model* but they extend it with a synchronous sequence of actions at startup. This design feature is to allow rapid, predictable sequences of actions to be performed in a synchronous fashion before starting the thread. In the example of a job submission this allows checking whether the job can be performed before starting the submission. The behaviour of an asynchronous processor is described in Fig. 3-47.

This processor has two sequences of actions. The synchronous sequence is $A_1 \rightarrow A_2 \rightarrow A_3$ at the end of which message M_2 is returned. After this synchronous sequence the asynchronous part is triggered. This sequence could be $A_4 \rightarrow A_4 \rightarrow \ldots \rightarrow A_6$, or $A_4 \rightarrow A_4 \rightarrow \ldots \rightarrow A_5 \rightarrow A_6$ depending if message M_3 is received. Finally message M_5 will enquire about the current status of the processor. The asynchronous part of a processor has three characteristics that make it preferable for lengthy or unpredictable events:

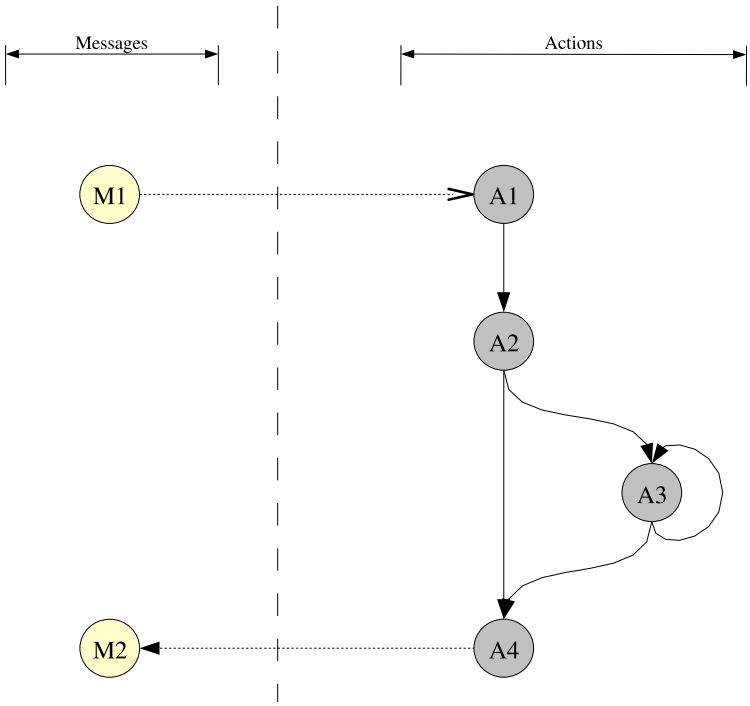


Figure 3-46: Behaviour of a synchronous processor.

- It has an *asynchronous behaviour* that allows the safe implementation of lengthy or unpredictable processes,
- its status can be *queried* by other processes, and
- it has a *reactive behaviour* that allows other processes to change the sequence of its actions.

Managers

Processors, either synchronous and asynchronous, are managed by *Managers*. The purpose of these components is to provide the right processors upon the reception of a message. They do this by mapping processors and messages in maps. While synchronous processors are easily managed, asynchronous processors need to be managed more carefully in order to allow them to be retrieved when necessary. To allow this a Processor Manager stores both types and instances of an asynchronous processor. This is illustrated in Figure 3-48.

Message M_1 is compatible with key K_1. When the processor manager receives it,

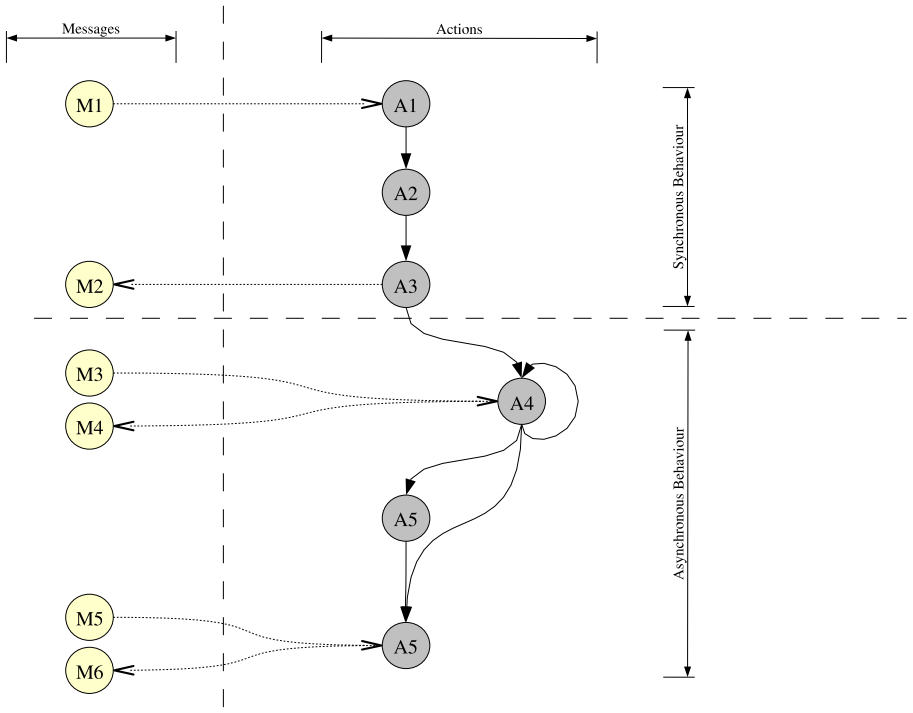


Figure 3-47: Behaviour of an asynchronous processor.

it creates an instance of Processor P_1 (by instantiating an object of class C_1). The processor performs its synchronous part and then returns a message. The Manager then *adds the instance* P_1 of the processor to its map with a different key K_2. Now the instance of the processor can be contacted through messages compatible with the key K_2, such as M_3, but the manager will also be ready to instantiate as many processors as necessary. For job submission, as an example, the manager will create an instance of the processor each time a job can be executed successfully but will also map each request regarding already running jobs to the appropriate instance of a processor.

3.16 Topologies

The previous sections were devoted to the description of the internal architecture of Social Grid Agents and their components. This section describes how the agents are connected in some of the topologies listed in section 3.3.

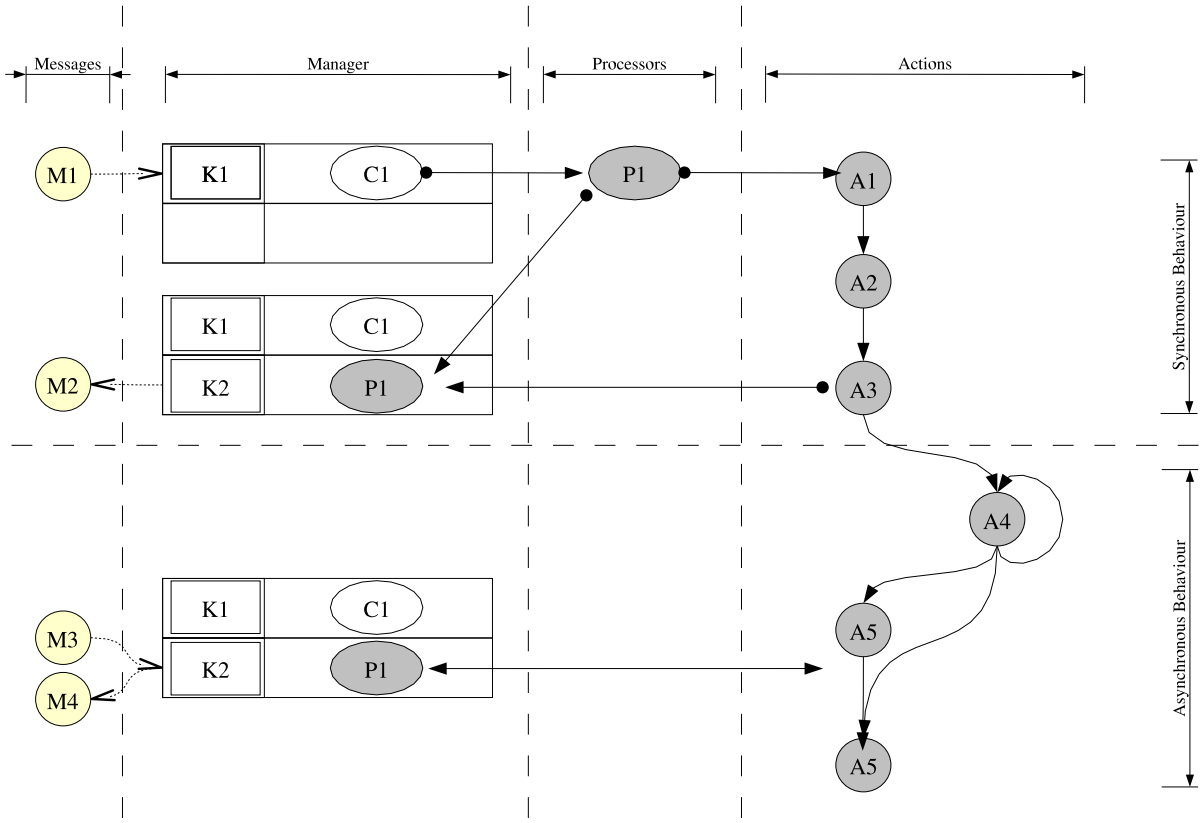


Figure 3-48: The manager.

3.16.1 Simple Purchase

A *Simple Purchase* topology (introduced in section 3.4.1) is used to cover three possible scenarios: a *Selfless Charitable* relationship, a *Commodity Market* and a *Posted Price Model* (the latter two economic models are described in section 2.2.2). The three different behaviours are achieved by simply setting the *Pricing Policy* (see section 3.2.6) as follows:

- *Charity.* To model a charitable behaviour, the agent providing the service sets to zero the Pricing Policy that determines the prices applied to the agents it intends to benefit. The agent can also describe more complex conditions to constrain the charitable donation by setting conditions on the *Allocation, Execution* and *Value Policies* (see section 3.2.6)

- *Commodity Market.* In a Commodity Market the Pricing Policy is set to be the same for all the clients.

- *Posted Price Model.* In a Posted Price Model, different *Pricing Policies* can be set as special offers. The model can be further extended by the possibility of defining specific *Pricing Policies* for particular users to model common commercial practices such as a Loyalty Program.

3.16.2 Pub Topology

In a *Pub Topology* (see section 3.5.3) two or more agents are involved in a sharing mechanism whereby a social agent may accept requests from another social agent provided that the requester is granted enough tokens.

Figure 3-49 shows a simple Pub Topology involving four agents: two social agents (SGA A and SGA B) that control two production agents (PGA A and PGA B). Let us suppose that at a given time SGA A decides to avail itself of the services of PGA B through the Pub Topology, then SGA A will send a request for a service to PGA B, if the request is granted (meaning that SGA B previously granted enough tokens to SGA A), then SGA A will instruct the production agent it controls (PGA A) to increase accordingly the number of tokens that are to be granted to SGA B.

This offers a very simple, self-adjusting load balancing system that can encompass any number of agents. This simple scenario can be expanded along many dimensions through the definition of appropriate policies: the value (and/or service metrics) of the required service can be limited by setting appropriate *Allocation* and *Value Policies*; tokens can also be granted to other pub parties on different conditions (at service request, upon completion of the service or upon successful completion of the service).

Two interesting aspects of the Pub Topology are its simplicity and its self-adjusting characteristics: in fact, if one party consistently fails to meet the requirements asked by the other it will simply not be granted any more tokens.

3.16.3 Tribe Topology

A *Tribe Topology* (shown in section 3.5.2) is similar to the *Pub Topology* with a meaningful difference: the policies (or part of them) that control if, how and when

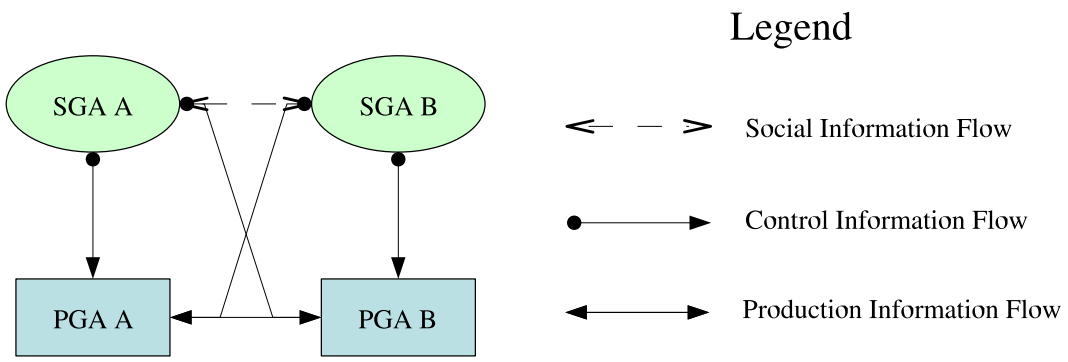


Figure 3-49: A Pub Topology.

the agents grant tokens to each other are defined by a *Chieftain*. The Chieftain is granted the authorization to set policies in the various agents by setting a proper *Authorization Policy* (see section 3.2.6) that controls the privileges granted to the messages received from the Chieftain.

3.16.4 Keynesian Scenario

A *Keynesian Topology* (shown in section 3.5.4) is a social structure where a *Keynesian Authority* grants special privileges to one or more agents. This topology aims at modelling scenarios where one or more authorities invest in public infrastructures to sustain projects of public interest. In a certain sense it is similar to the Tribe as in both cases Social Agents grant to an Authority the possibility of setting and modifying parts of their internal policies.

In its simplest form (shown in Figure 3-50) a Keynesian topology encompasses four agents: a *Client* (Client), a *Keynesian Authority* (Authority), a *Social Agent* (SGA A) and a *Production Agent* (PGA A).

If both Client and SGA A accept the authority of the Authority then they belong to a Keynesian Topology. The Client is granted by SGA A particularly favourable conditions that are described by policies defined by the Authority and accepted by SGA A; the most common policy affected in a Keynesian scenario is the *Pricing Policy* stating that Client is to be granted a lower price than the other costumers. These policy settings can model a scenario in which part or all the resources controlled

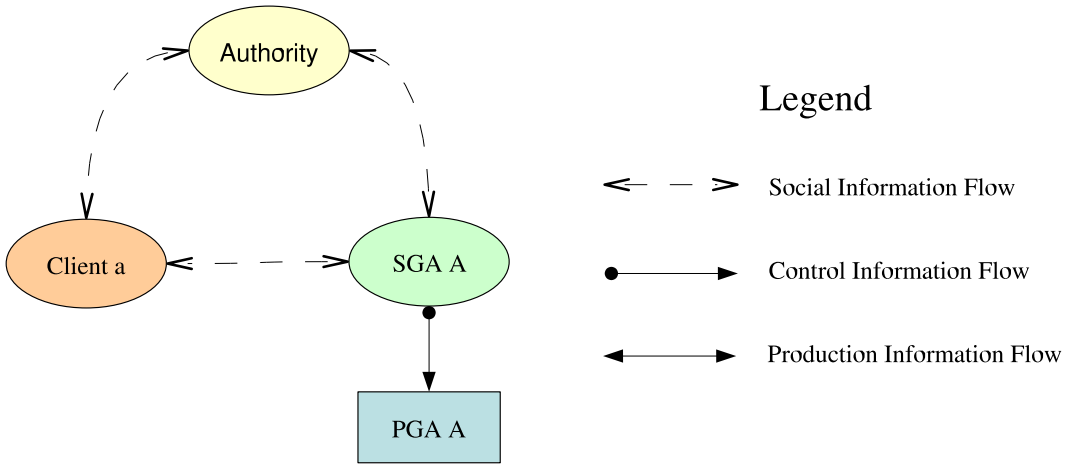


Figure 3-50: A Keynesian Topology.

by SGA A are partially or fully funded by the Authority.

But there is also the possibility that the Keynesian Authority decides to fully or partially fund the activities of Client by covering part or all the price requested by SGA A. In this case it will modify the *Billing Policies* of SGA A to allow a third party (in this case itself) to cover, partially or entirely, the costs.

3.17 Appropriate Technologies

This architecture has been realized and refined in various guises in a series of experimental prototypes. For the last of these, the software technology is Java™, the native service and skeleton technology is Globus GT4 Web Services (WSRF) [32], whilst the native language is Condor ClassAd. The latter is discussed in Chapter 3.18. It hardly needs to be said that the technological choices are just that, and that the architecture is an abstraction valid whatever the choices.

3.18 Behaviour Policies

In Section 3.12 we have introduced the concept of an *"Agent Behaviour Policy"* hosted in a *Agent Behaviour Engine*. Agents have to make decisions driven by external elements and their own behavioural characteristics. They must decide whether

to accept or reject requests from other agents depending on their social relations, they must assess the price of a service, they must execute actions in order to perform services. The architecture described in Section 3.12 implements a software infrastructure capable of mapping messages into policies and a sequence of actions. As yet there has been no explanation of how these decisions are described and memorized in the agents.

This Section is devoted to this topic.

3.18.1 The Native Language

The earliest experimental prototypes used the Java^{TM}Language to implement both the Behavour Engine infrastructure of the agents and their behaviour policies. As this proved to have too many shortcomings, the policies, maps and action sequences were re-expressed in a native language. The earlier work indicated than an efficient and expressible language was needed. Side-effect freedom was considered important for (future) provability, and the Haskell language[13] was considered. The current choice is the ClassAd language[80, 79, 81, 56].

This choice was based on four major considerations.

- The ClassAd language is widely used and understood in the Grid community.
- The ClassAd language is used by the gLite and Condor middlewares.
- The ClassAd language is specifically designed to associatively *matchmake* entities.
- The ClassAd language is a functional language in which the evaluation order and the priority of operators are specifically designed to allow the matchmaking of partly defined data structures.

It is not the focus of this thesis to describe in detail the ClassAd language and it should suffice here that the ClassAd Language is a *strongly-typed, side-effect-free functional language* whose main characteristic is its ability to support matchmaking of different entities.

The two ClassAd expressions *ClassAd A* and *ClassAd B* can be matchmaked against each other using the combined namespaces of both expressions. This means that we can define *if* and *how much ClassAd A* and *ClassAd B* are compatible.

This is achieved by two particular functions of the ClassAd Language: the *Requirements* function and the *Rank* Function.

If we define V as the set of all the possible ClassAd Values and $V \times V$ as the *Cartesian Product* of V, we can define the Requirements and Rank functions as :

$$Requirements(V \times V) \rightarrow V \tag{3.26}$$

$$Rank(V \times V) \rightarrow V \tag{3.27}$$

The Requirements function is a special function the range of which is either a Boolean or one of the two specific ClassAd values *UNDEFINED* and *ERROR*. We can describe in more detail the Requirement functions as:

$$Requirements(V \times V) \rightarrow Q \tag{3.28}$$

$$Q = \{true, false, UNDEFINED, ERROR\} \tag{3.29}$$

$$Q \subset V \tag{3.30}$$

The Rank function, on the other hand, is a special function the range of which is either a natural number or one of the two specific ClassAd values UNDEFINED and ERROR. We can describe in more detail the Rank functions as:

$$Rank(V \times V) \rightarrow K \tag{3.31}$$

$$K = N \bigcup \{UNDEFINED, ERROR\} \tag{3.32}$$

$$K \subset V \tag{3.33}$$

For example, let there be two ClassAd expressions, where the first, *ClassAd A*, is:

```
[
 a = 1;
 b = 4;
 Requirements = (a == 1) && (other.b == 2);
 Rank = a + other.b;
]
```

The second, *ClassAd B*, is:

```
[
 a = 3;
 b = 2;
 Requirements = (a == 3) || (other.b <= 10);
 Rank = other.a;
]
```

In this case *ClassAd A* and *ClassAd B* match. The rank of the match is 3 for *ClassAd A* and 1 for *ClassAd B*.

Both functions can use the value UNDEFINED internally to cope with values that are not defined in the current scope or evaluate to UNDEFINED when such values are met. For example, take the two following ClassAd expressions:

```
[
 a = 1;
 b = 4;
 Requirements = (other.c is undefined) || (other.c == 2);
 Rank = a + other.b;
]
```

```
[
 a = 3;
 b = 2;
 Requirements = (other.c == 2);
 Rank = other.a;
]
```

The Requirement function of the first ClassAd will evaluate to *true* while the second will evaluate to UNDEFINED.

3.18.2 ClassAd and Agents

The prototype implementation of Social Grid Agents uses the ClassAd Language extensively. Agents often *map* entities to other entities: actions are mapped to other actions, messages are mapped to processors, messages, relationships and modalities are mapped to policies and actions.

These mapping actions are defined in ClassAd in a component called a *ClassAd Mapper*; a ClassAd Mapper is a *HashMap* that links *keys* expressed in the ClassAd Language to Java^{TM}objects (that can also be *String* representations of yet other ClassAd expressions). The ClassAd Mapper is used in Social Grid Agents to perform three basic mapping operations:

- mapping messages to service providers,
- mapping actions to actions, and,
- mapping policies and modalities to enforced policies.

Messages and providers

Chapter 3.12 described how agents map messages to providers, either simple service providers or more complex processors. This operation is performed inside *Managers*, which are based on *ClassAd mappers*. An *Asynchronous Processor* capable of managing a gLite job submission can be mapped as follows:

```
[
 ...
 ProcessorManager prManager = new ProcessorManager();
 ...
 String key = "
  [Requirements =
   (other.modality == \"asynchronous\") &&
   (other.action == \"job execution\") &&
   (other.action.object.type == \"lcg2\");
  ]";
 ...
 prManager.add(key, JSAsyncProcessor.class);
 ...
];
```

Now the manager will be able to map messages such as:

```
[
 ...
 modality = \"asynchronous\";
 ...
 action =
 [
  name = \"job execution\";
  object =
  [
  type == \"lcg2\");
  ...
  ];
 ...
 ];
 ...
];
```

Actions

Processors are engines capable of executing sequences of *actions*. They can perform this either in synchronous or asynchronous mode. The abstract architecture of processors is described in Chapter 3.12. The ClassAd language is used both to define the single *actions* and their *sequences*.

```
[
 ...
 name = \"submission\";
 type = \"prepare\";
 execution = \"remote\"];
 ...
];
```

This describes the submission step of a job execution:

- The field *name* is used as an identifier of the action.
- The field *type* defines *prepare actions* that are to be performed during a synchronous run of the processor or *execution actions* that will be performed during an asynchronous run.
- The field *execution* defines *remote* actions that will be executed by other Service Providers or *local* actions that will be performed locally by the *processor*. A *processor* in which all the actions are defined as *remote* acts as a pure execution engine that invokes other Service Providers in the correct order.

```
[
 ...
 name = \"status check\";
 type = \"run\";
 execution = \"local\"];
 ...
];
```

This describes the *status check* action in which the current status of the job is investigated and analyzed.

Every processor has a description in ClassAd which contains the current action and some data related to its status. The engine of the processor will use this information to determine the next action.

```
...
String key_submission_ok = "[Requirements =
 ((other.current.action.name == \"submission\")
 &&
 (other.current.status.result == \"success\"))
;];";

String action_status_check = "[
 ...
 name = \"status check\";
 type = \"run\";
 execution = \"local\"];
 ...
];"
...
actions.add(key_submission_ok, actions_status_check);
...
```

This case maps the sequence $submission \rightarrow status\_check$.

Information Structure and Flows

As we have seen, ClassAd expressions are used to describe all information that is exchanged among the agents and the behaviour and status of the agents. This information is structured in a uniform way so that sets of parsers present in every agent are able to treat it in a semantically consistent fashion. The information is structured as follows:

Messages

Agents communicate through messages which are *ClassAd RecordExpr* expressions containing:

- A *Content* that is the content of the message. The content can either be an *Object* or an *Action*.
- A *Sender* that is the *Agent Identity* of the agent who sent the message.

Agent Identity

Agent's identities are described through *ClassAd RecordExpr* expressions that contain different fields that may have different ways to be *Authenticated*. This is to allow agents to implement and accept different *Security Policies* depending on the relations they are currently engaged in.

- A *Distinguished Name* that is an X509 distinguished name
- An *SGA NAME* that is an agent-specific identifier.
- The *Date of Birth* of the agent.
- A *Domain* to which the agent belongs.

Object

Objects are *ClassAd RecordExpr* expressions that describe either local or remote objects. They consist of:

- A *Description* that describes the type of the object and its location if it is not local to the expression.
- A *Value* that is defined only if the object is local to the expression.

Actions

Actions are *ClassAd RecordExpr* expressions that contain the following information:

- A *Description* that contains the information listed in 3.18.2 and the *Status* of the action.
- An *Input* that can be either an *Action* or an *Object*.
- An *Output* that can be either an *Action* or an *Object*.
- A set of *Modalities*. If it is defined, this is a *ClassAd RecordExpr* expression that describes the *Set Of Constraints* that is requested by the agent requesting the action, the *Requester*.
- A set of *Policies*. If it is defined, this is a *ClassAd RecordExpr* expression that describes the *Set Of Constraints* that is applied by the agent performing the operation for the *Requester*.
- A set of *EnforcedPolicies* that is a *ClassAd RecordExpr* expression that describes the *Set Of Constraints* that is a subset common to both *Policies* and *Modalities*.
- A set of *Data* that can be used to describe data relevant to the action that can be not easily handled as *Input*.
- A *Requester*, that is the identity of the agent requesting the action.
- A *Beneficiary*, that is the identity of the agent that is ultimately the beneficiary of the action.

A Set of Constraints

Policies, modalities and EnforcedPolicies are all described as sets of *Constraints*.

A Constraint

A Constraint describes *"how and if"* an action is to be executed. It consists of:

- A *Data* expression that contains any data specific to the constraint.
- A *Clause*, a Boolean expression that evaluates to either *true* or *false*.

Information Models

The information structure illustrated allows the agents to exchange information with a certain degree of flexibility. Depending on how the information flows are structured it is possible to implement two main computation models: *side effect free* and *with side effects*. Currently agents use mainly the *side effect free model*; where actions are composed through their *input* and *output* fields. We illustrate this with an example of the steps that usually arise in the handling of a message in an agent.

- The *Acceptance Action* is always the first action performed when a message is received (m_0) by an Agent. It accepts as Input the message itself and returns one that has been modified (m_1) in the fields relating to the *Authentication* of the sender (the sender must have a valid X509 certificate in order to have its message accepted by the GT4 container that hosts the agent) and the status of the message.

$$m_1 = f_{acc}(m_0) \tag{3.34}$$

- The *Action* contained in the message is then extracted from the accepted message and a suitable processor (if any) is found in the *manager*.

$$a_1 = (m_1).content \tag{3.35}$$

- The Action is then mapped to a set of *Policies* by a *Policies Mapping Provider*.

$$a_2 = f_{map}(a_1) \tag{3.36}$$

- The Action with the constraints represented by the *Policies* is then usually processed by a *Policies Enforcement Provider* that defines the subset of constraints defined by the *Policies* and *Modalities*.

$$a_3 = f_{enforcer}(a_2) \tag{3.37}$$

- The Action with the constraints represented by the *Enforced Policies* is then usually passed again to the *manager* to find a suitable provider; this operation can yield yet another action or an object.

$$e = f_{exe}(a_3) \tag{3.38}$$

.

Policies and Modalities

Agents receive, assess and possibly accept messages requesting execution of services. These requests are detailed with modalities. On the other hand, agents providing the services define a set of policies. If these two sets overlap, then the service can be executed. This is shown in Fig. 3-51.

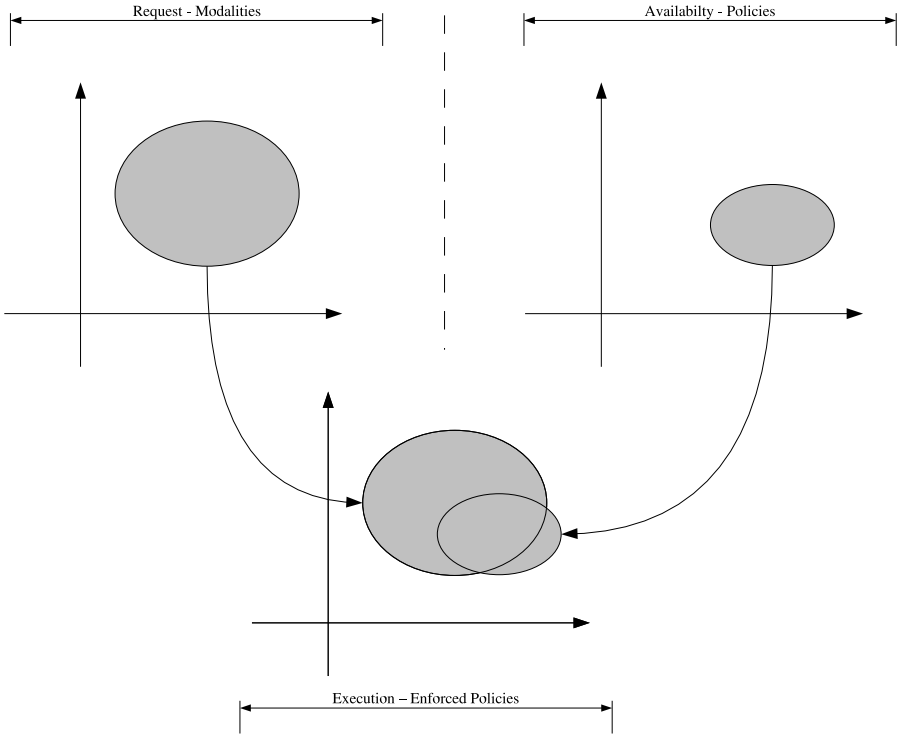


Figure 3-51: Policies and modalities.

The following snippets of code show this concept applied to a generic job submission. The request will be of the form:

```
[
 ...
 subject = agent A
 action = "job execution";
 ...
 Requirements = ... (other.cputime >= 10000) ...;
]
```

Here *agent A* requests the execution of a job with the maximum duration of 10000 seconds. On the service provider side there will be a set of policies that are applied to *agent A*:

```
[
  ...
  Requirements = ... (cputime <= 20000) ...;
  ...
];
```

In this case, *agent A* is entitled to a maximum of 20000 seconds of CPU time. The request is acceptable as both the requirements evaluate to true and the service request can be executed with this set of *enforced policies*:

```
[
 ...
 Requirements = ... (cputime >= 10000) && (cputime <= 20000)
     ...;
 ...
];
```

This example covers only a simple case in which the namespaces of both ClassAd expressions allow a complete evaluation. However, the architecture of the Social Grid Agents is multi-layered and it cannot be assumed that every layer has a perfectly overlapping namespace. In fact, the various layers of the architecture share only

subsets of their namespaces and this forces two evaluation strategies: *abstraction* and *delegation*

Policy abstraction

Agents are arranged in topologies that define production systems and social structures. To do this, they exchange information regarding the status of the resources they control, the production they perform and the societies they belong to. To manage this complex flow of information it is necessary for the agents to manage only the subset of information that really pertains to the decisions they have to make. Agents use two abstractions: Value and Price. This flow of information runs through the control chain of agents. Here policies are defined at different level of abstractions. This is described in the example of Fig. 3-52.

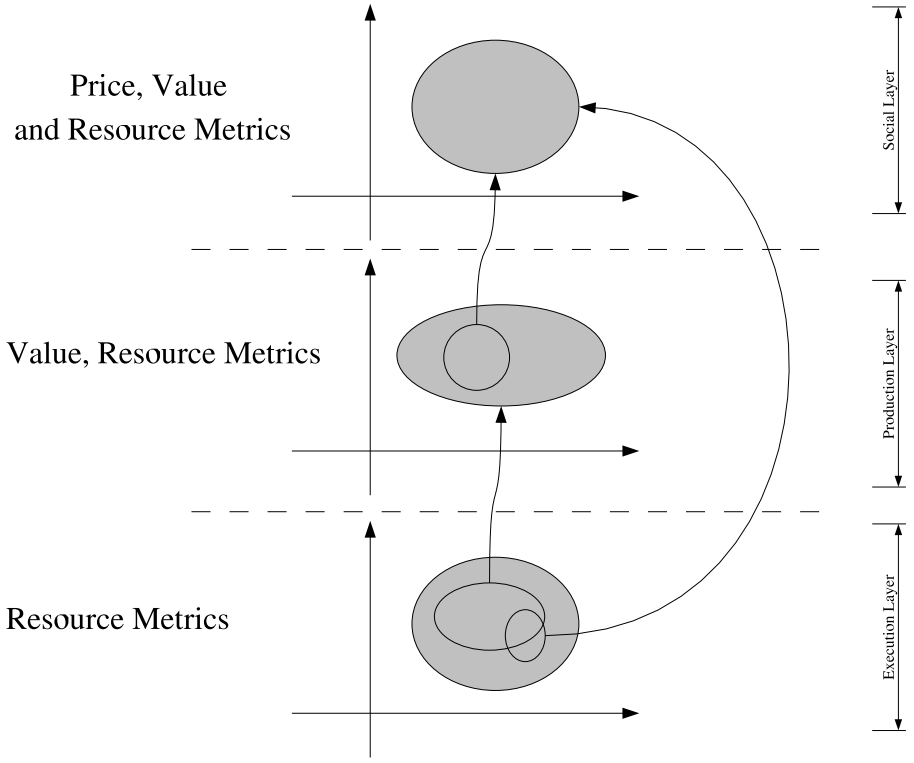


Figure 3-52: Policies and modalities abstraction.

Let p^i be the set of parameters that defines the i^{th} level. Then, at the $(i+1)^{th}$ level a set of parameters p^{i+1} can be defined which can be based entirely or partially

on the lower level i.

As a very simple example let us define value and price at the Production and Social level as functions of a metric that defines the number of free CPUs.

```
[
 ...
 free_cpus = 2;
 ...
]
```

Part of this information is used in the Production Layer to define the value of the resource. It is possible, for example, that the value of the resource is defined, among other parameters, as the inverse of the number of available CPUs.

```
[
 execution =
 [
  ...
  free_cpus = 2;
  ...
 ]
 ...
 value = ... (1/execution.free_cpus + 1) ...;
 ...
]
```

At the Social level, part of the information of the Production Layer and part of the information of the Execution Layer is used to define a policy that has to be applied to define the minimal price at which a job submission is to be sold to *agent A*:

```
[
 ...
 production =
 [
  execution =
  [
   ...
   free_cpus = 2;
   ...
  ]
  ...
  value = ... (1/(execution.free_cpus + 1) +1) ...;
  ...
 ]
 ...
 price = production.value +1;
 ...
 Requirements = (other.price >= price) && (other.free_cpus
 <= production.execution.free_cpus);
 ...
]
```

In this case the request of *agent A* will be accepted if and only if the price offered is greater than the (*production value* + 1) and the number of requested CPUs is available.

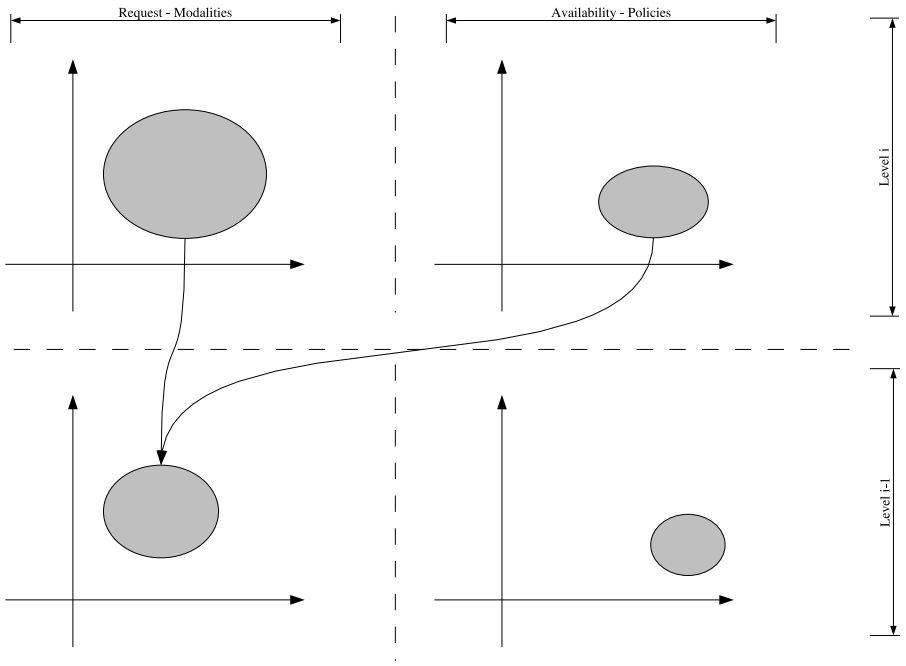


Figure 3-53: Policies and modalities enforcement.

3.18.3 Policy Enforcement

The complementary process is the enforcement of policies described in Fig.3-53. Let us describe an arbitrary level i. The *requester* will define its request and modalities with the ClassAd expression $expr^i_{req}$ that defines a set of values p^i_r, a Requirement and a Rank function:

$$Requirements = f^i_r(V \times V) \rightarrow Q \tag{3.39}$$

$$Rank = g^i_r(V \times V) \rightarrow K \tag{3.40}$$

Conversely, the *provider* will define its services and policies with the ClassAd expression $expr^i_p$ that defines a set of values p^i_p, a Requirement and a Rank function:

$$Requirements = f^i_p(V \times V) \rightarrow Q \tag{3.41}$$

$$Rank = g^i_r(V \times V) \rightarrow K. \tag{3.42}$$

At level i, the matchmaking process will have two possible outcomes: it can eval-

–	$TRUE$	$FALSE$	$ERROR$	$UNDEFINED$
$TRUE$	$TRUE$	$FALSE$	$ERROR$	$UNDEFINED$
$FALSE$	$FALSE$	$FALSE$	$ERROR$	$UNDEFINED$
$ERROR$	$ERROR$	$ERROR$	$ERROR$	$ERROR$
$UNDEFINED$	$UNDEFINED$	$UNDEFINED$	$ERROR$	$UNDEFINED$

Table 3.1: ClassAd extension of the Boolean AND function.

uate or not. Unless the Requirements and Rank functions use directly the function *is defined* to handle cases in which values are undefined, they will evaluate to UNDEFINED if any of the used expressions are not defined in the Cartesian product of the namespaces of $expr_p^i$ and $expr_r^i$: $p_p^i \times p_r^i$. In the latter case at least one of the functions will evaluate to UNDEFINED, so the agent will not be able to make a decision and will have to delegate it to a lower level.

The *provider* at level i will then act as a *requester* of level $i-1$, issuing a request that defines a set of values p_r^{i-1} (that contains all the needed values of the level i and $i-1$), a Requirements and a Rank function:

$$Requirements = f_r^{i-1}(V \times V) \rightarrow Q \tag{3.43}$$

$$Rank = g_r^{i-1}(V \times V) \rightarrow K. \tag{3.44}$$

where:

$$f_r^{i-1} = f_p^i \&\& f_r^i \tag{3.45}$$

$$g_r^{i-1} = F(g_r^i, g_p^i) \tag{3.46}$$

Where F is any function that combines the ranking functions and where && is the implementation offered by ClassAd to the Boolean AND function. This implementation also supports the *ERROR* and *UNDEFINED* values as described by Table 3.1

Partial Evaluation

At each step the requirements and rank functions are to be *partially evaluated*. A ClassAd function evaluates to *UNDEFINED* if any of its parts is *UNDEFINED* (and this possibility is not taken into account with the *is undefined* special function as described in section 3.18). This makes it impossible to perform a detailed partial evaluation in which only the values defined at each step are evaluated and all the others are propagated to the other steps. This problem has been solved by writing a *Partial Evaluation Parser* that parses a ClassAd expression and substitutes only the values that are defined.

Chapter 4

Implementation and Experiments

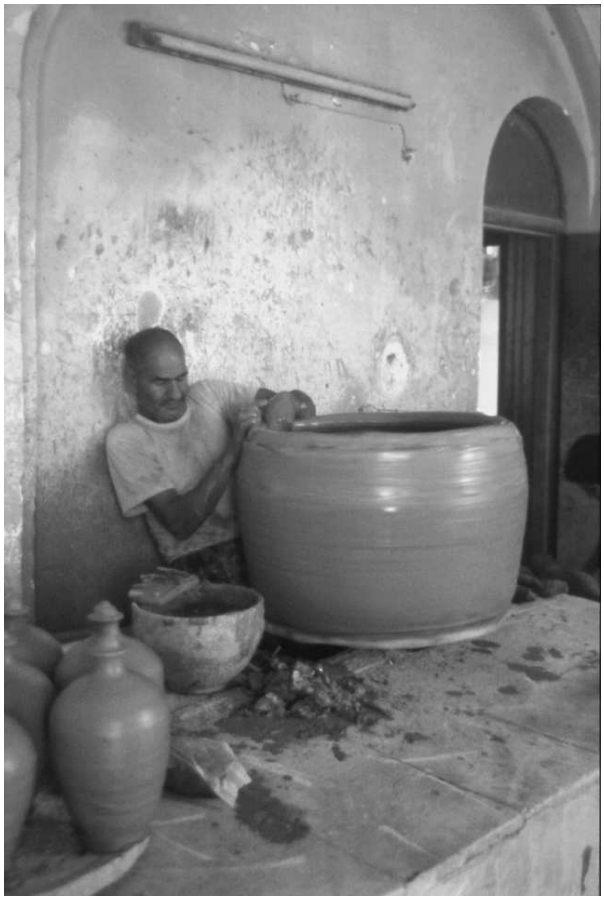


Figure 4-1: It's a dirty job but someone has to do it.

"A bad craftsman blames his tools, an even worse one blames the nail"

Anonymous

No amount of experimentation can ever prove me right; a single experiment can prove me wrong.

Albert Einstein

A theory is something nobody believes, except the person who made it. An experiment is something everybody believes, except the person who made it.

Albert Einstein

4.1 Introduction

While chapter 3 was devoted to the generic description of paradigm, abstract methods and abstract architecture of the agents, this chapter covers the details of the implemented prototypes. A brief description of the different prototypes implemented during this enquiry is followed by five main sections: the description of a metagrid prototype, the implementation details of different Social Grid Agents, the details of a prototype of a complex topology, the definition and enforcement of a set of policies and, at the end of the chapter, the experiments that were performed on Social Grid Agents.

4.2 Past Implementations

Four different prototypes have been developed during this research. This section will briefly describe them, their shortcomings and the design decisions that prompted the next implementation.

4.2.1 First Implementation

The first prototype of Social Grid Agents was based on the WebCom technology that provided both the communication platform among the agents and one of the middlewares encompassed by the prototype (the other two being LCG2 and GT4).

The purpose of this prototype was to to prove the feasibility of a platform able to submit jobs to multiple middlewares and to use *Condensed Graphs* to represent the dependencies among different jobs.

4.2.2 Second Implementation

The second prototype of Social Grid Agents [73] was shown and peer-reviewed at the WebCom-G project review held in Cork in October 2005. It had two major differences from the first prototype: communication among the various components was based on GT4 technology and a *File Staging* mechanism was implemented. This mechanism is based on the use of a central repository for files, which were moved to the agents only when needed in order to reduce the overhead caused by file staging.

The architecture of the second prototype is shown in Figure 4-2.

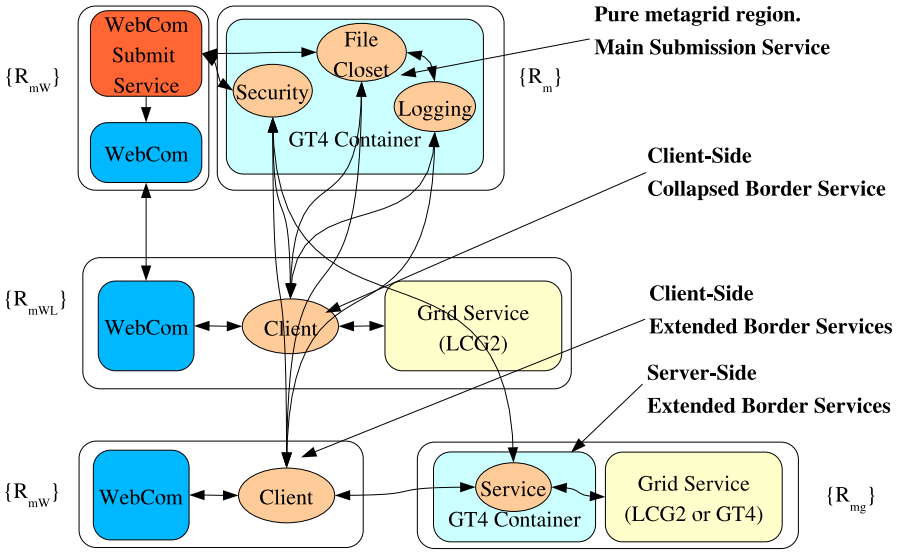


Figure 4-2: Architecture of the second prototype.

4.2.3 Third Implementation

The third prototype of Social Grid Agents extended the second implementation with the concepts of maps, messages, policies and modalities. Maps were based on *hash maps* while messages, policies and modalities were expressed as hierarchies of classes.

This design solution proved to be poor. It yielded an architecture that was both hard to understand and maintain. These difficulties were the main factors that led to the research of yet another language for the expression of policies.

4.2.4 Fourth Implementation

The decision to adopt the *ClassAd* language as the native language led to the fourth and last prototype.

4.3 The current prototype

The section on the last prototype describes its most important aspects:

- A prototype of the metagrid environment
- The implementation details of different Social Grid Agents
  - A production agent (*gLite-wms-pga*), a production agent that wraps the *gLite Workload Management System*
  - A production agent (*gt4-gram-pga*), a production agent that wraps the *GT4 GRAM System*
  - A production agent (*webcom-pga*), a production agent that wraps the *Webcom* workflow engine
  - A social agent (*gLite-wms-sga*), a social agent that controls *gLite-wms-pga*
  - A bank
  - An indexing agent
- A complex topology that supports different, coexisting allocation mechanisms
- An example of a description and enforcement of policies with the ClassAd language

4.3.1 A metagrid implementation

Figure 4-3 describes a prototype implementation of a metagrid that encompasses three different Grids: gLite, GT4 and WebCom, and the native Metagrid Job eXchange [71, 72]. In this implementation, the Metagrid Job eXchange and WebCom are directly controlled by the same social agent while gLite and GT4 service provision is controlled by two separate social agents with which social and/or economic exchange is to be established to gain the necessary services.

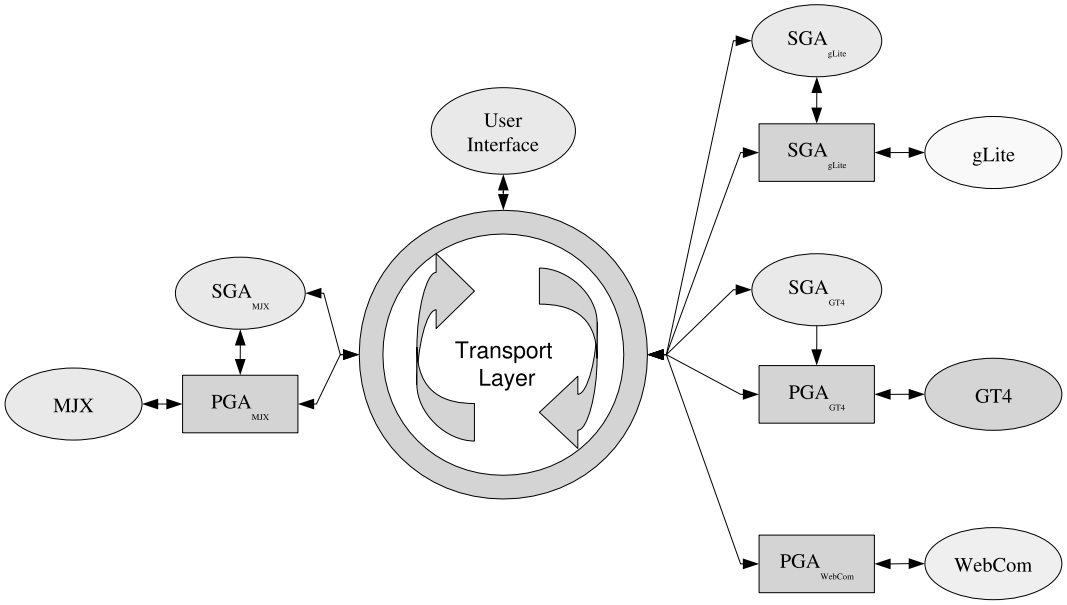


Figure 4-3: Architecture of the example.

The social topology is described by the upper part of Figure 4-4. The control/ownership relations are drawn with vertical lines. The production topology is represented in the lower part of Figure 4-4.

The behaviour of the system is as follows:

1. *Job submission*: The user submits a complex job (to a very simple metagrid user interface), the execution of which requires gLite, GT4 and WebCom resources.

2. *Execution planning*: The social agent (SGA_{MJX}) that controls the Metagrid Job eXchange Service decides whether to accept to try to submit the job or not

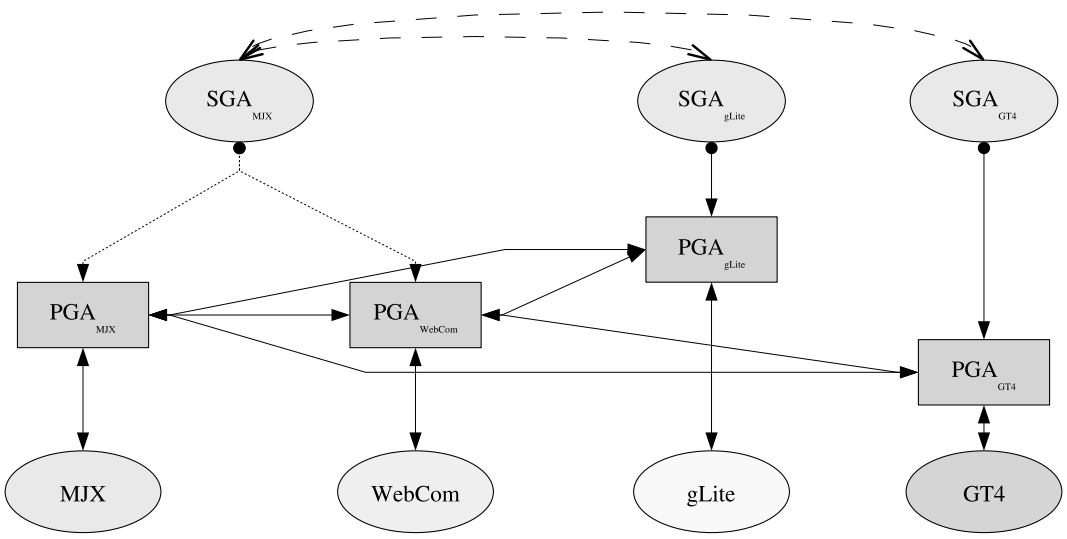


Figure 4-4: Topology of the example.

3. *Needs evaluation*: SGA_{MJX} assesses the needs for the successful submission of the job: gLite, GT4 and WebCom resources. It then checks this list against the resources it directly controls. This results in a list of the resources that are needed and that must be obtained through social exchange

4. *Social and Economic arrangements*: SGA_{MJX} checks if in the list of other social agents it knows there is any that offers the resources it needs. If so it contacts the agents and asks what is the cost of such resources, where this cost can be zero for social agents with which SGA_{MJX} has cooperative relationships. If the social and economic arrangements can provide enough resources for job execution then the social agents that control the gLite and GT4 resources (SGA_{gLite} and SGA_{GT4}) instruct their production agents (PGA_{gLite} and PGA_{GT4}) to accept the requests coming from SGA_{MJX}. The job is then passed to the production agent (PGA_{MJX}) with the list of the other production agents that will be needed (PGA_{WebCom}, PGA_{gLite} and PGA_{GT4}).

5. *Job execution*: PGA_{MJX} submits the job to the production agent that controls the WebCom resource. WebCom unfolds the workflow graph that represents the job and starts executing it. When a node representing an operation that needs a gLite or a GT4 resource is encountered, the job description (JDL or

RSL) is sent to the production agents that control these resources (PGA_{gLite} and PGA_{GT4}).

6. *Job termination*: When the complete job has been executed the results are passed to the Metagrid Job eXchange and then passed back to the user.

4.3.2 Implemented Prototypes of Social Grid Agents

We have seen in Section 3.10 how agents "live" in a metagrid environment that encompasses different middlewares. Usually border agents are production agents that interface to services that are specific to the middleware. Section 4.3.2 details the production agent that resides on the border with the gLite middleware and offers its services of job submission. Section 4.3.2 details the production agent that resides on the border with the GT4 middleware and offers its services of job submission. Section 4.3.2 details the production agent that resides on the border with the WebCom middleware and offers its services of workflow engine. Section 4.3.2 details a social agent that controls a production agent. Section 4.3.2 details a bank and section 4.3.2 details an agent with indexing capabilities.

A production agent: GLITE-WMS-PGA

The *gLite Wms Production Agent*, or *gLite-wms-pga* for the sake of brevity, is a production agent that wraps the *gLite Job Submission System* accessing it through an instance of the *gLite UI*. It is an *border agent* that connects the metagrid regions with the Grid region hosting gLite.

The agent reacts to two main sets of messages: *action-related messages* and *control-related messages*: the first concern job-submission related actions and the second concern control and management actions. The architecture of *gLite-wms-pga* is shown in Fig. 4-5; it comprises a *manager*, four *processors* and several *providers*, two of which are connected to external services: the *gLite UI* and the *Grid4C agents*.

The GLITE-WMS-PGA processors

The processors are :

- *The JSSyncProcessor*, a synchronous interface to the job submission system.
- *The JSAsyncProcessor*, an asynchronous interface to the job submission system
- *The Policy Processor*, a synchronous processor that checks and enforces all the policies related to the job submission
- *The Control Processor*, a synchronous processor that checks the authorization of messages that request modifications in the overall policies of the agent

The GLITE-WMS-PGA providers

The providers are :

- *The Acceptance provider* performs some standard operations at the reception of the message, such as logging the event and filling the implicit authentication data.
- *The Job Submission provider* constitutes the main interface to the Job submission system.
- *The Job Availability provider* checks if a job can be submitted without performing the submission operation.
- *The Policies Mapper* maps messages, sender id and modalities into a set of policies.
- *The Allocation Expert* enforces the policies that define the allocation of resources.
- *The Value Expert* enforces the allocation policies that are described as functions of value.
- *The Logger* manages the log
- *The Billing Expert* updates and queries the consumption records of the agent. It may be interfaced with Grid4C agents as described in section 4.3.2

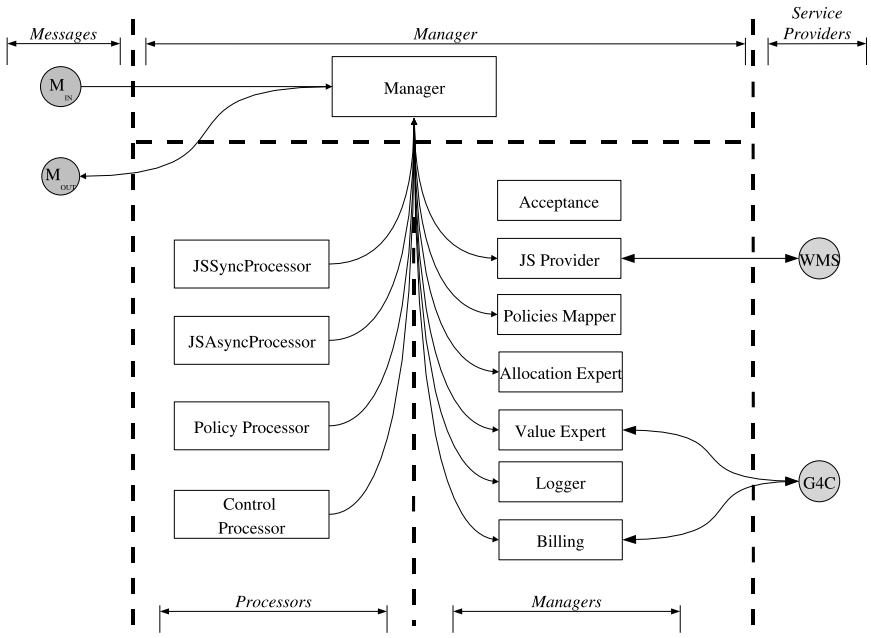


Figure 4-5: Architecture of the gLite-wms-pga agent.

The GLITE-WMS-PGA behaviour

Although PGAs are deliberately kept simple in concept, their behaviour can be complex. Let us examine the most common behaviour of this particular agent. A message similar to the one below will map to an *Asynchronous Job Submission Processor*.

```
[
 ...
 modality = \"asynchronous\";
 ...
 action =
 [
  name = \"job execution\";
  object =
  [
  type == \"lcg2\");
  ...
  ];
 ...
 ];
 ...
];
```

The initial sequence of actions defined by the *JSAsyncProcessor* can be performed synchronously as they are of short and predictable duration while the submission itself is performed asynchronously as it is not possible to predict the duration of the job.

The sequence of actions of the asynchronous job submission processor is illustrated in Fig. 4-6. It comprises:

- Acceptance: a preliminary step that writes in ClassAd format the data that is implicit by the fact that the message has been accepted by the GT4 Container. These are: date and time of the reception of the message and the X509-based authentication of the sender of the message.

- Map Policies: this step defines the sets of policies that must be enforced for the job submission.

- Policies enforcement: this step enforces the set of policies defined in the previous step.

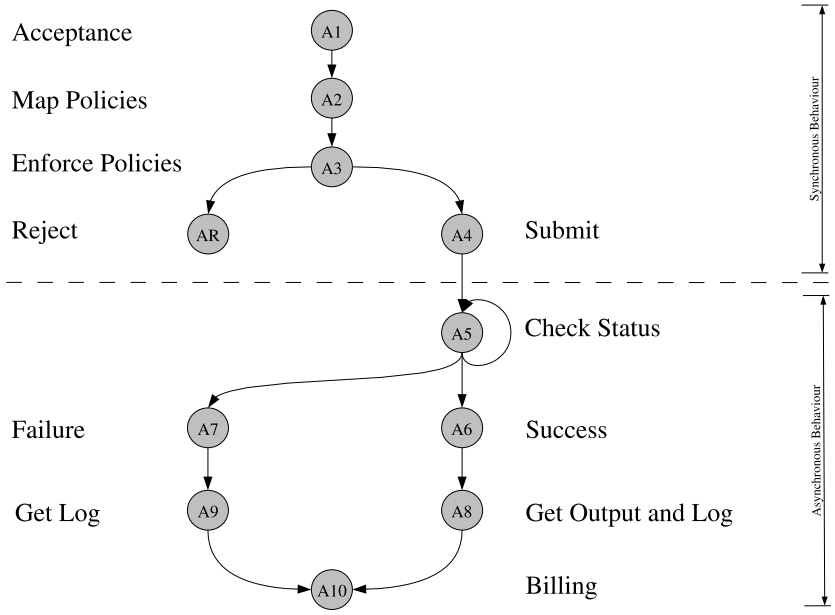


Figure 4-6: Actions performed by the Asynchronous Job Submission Processor.

Action	*Processor*	*Provider*
Acceptance	-	Acceptance
Map Policies	-	Policies Mapper
Enforce Policies	Policy Processor	-
Job Submission and execution	-	JS Provider
Billing	-	Billing

Table 4.1: Action to Provider map in the Job Asynchronous Processor.

- Job Submission and execution: these steps perform the usual actions needed for an execution of a job such as submission, status check, logging and output retrieval. The submission of the job is the last action performed in the synchronous part.

- Billing: this step updates the consumption records.

All these actions are marked as *external*; when they are to be performed then the manager attempts to find a proper provider. Table 4.1 shows the mapping performed.

All but the *Policy Processor* are simple providers that perform atomic actions.

Action	*Processor*	*Provider*
Authentication	-	Authentication Provider
Allocation	-	Resource Expert
Value	-	Value Expert
Authorization	-	Authorization Expert

Table 4.2: Actions to Providers map for the Policies Processor.

The policy processor, on the other hand, can enforce policies in a flexible order that can describe different priorities and inter-dependencies. The current behaviour of this policy processor is shown in Figure 4-7.

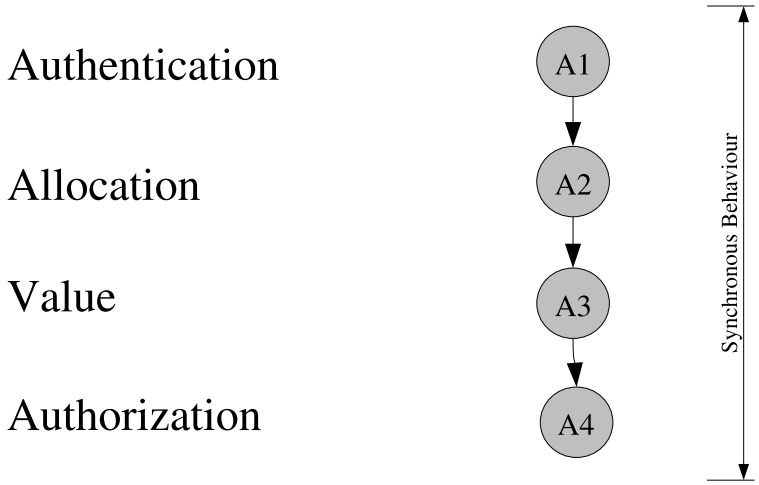


Figure 4-7: Actions performed by the Policies Processor.

Again, as all the actions of this processor are defined as external, the manager is responsible for finding a suitable provider. Table 4.2 shows the mapping performed.

Some of these providers such as the *Value Expert* and the *Authorization Expert* avail themselves of other providers. The Value Expert uses remote Grid4C agents for the assessment of the value of resources while the Authorization Expert uses the Job Submission provider to test the current availability of resources. Finally, many if not all providers use the *Logger* service.

A production agent: GT4-GRAM-PGA

The *GT4 Gram Production Agent*, or *gt4-gram-pga* for the sake of brevity, is a production agent that wraps the *GT4 Grid Resource Allocation and Management* service accessing it through an instance of the *WS-GRAM* grid service hosted in the same GT4 container of the agent. It is an *border agent* that connects the metagrid regions with the Grid region hosting Globus.

As for the *gLite-wms-pga*, the agent reacts to two main sets of messages: *action-related messages* and *control-related messages*.

The architecture of *gt4-gram-pga* is shown in Fig. 4-8; as for its gLite counterpart it comprises a *manager*, four *processors* and several *providers*. Differently from *gt4-gram-pga* only one provider is connected to an external service: the *WS-GRAM*, while there is not a G4C agent for GT4-Gram available yet.

The GT4-GRAM-PGA processors

The processors are the same of the *gLite-wms-pga agent*, they are:

- *The JSSyncProcessor*, a synchronous interface to the job submission system.
- *The JSAsyncProcessor*, an asynchronous interface to the job submission system
- *The Policy Processor*, a synchronous processor that checks and enforces all the policies related to the job submission
- *The Control Processor*, a synchronous processor that checks the authorization of messages that request modifications in the overall policies of the agent

The GLITE-WMS-PGA providers

The providers are :

- *The Acceptance provider* performs some standard operations at the reception of the message, such as logging the event and filling the implicit authentication data.
- *The Job Submission provider* constitutes the main interface to the Job submission system.

- *The Job Availability provider* checks if a job can be submitted without performing the submission operation.
- *The Policies Mapper* maps messages, sender id and modalities into a set of policies.
- *The Allocation Expert* enforces the policies that define the allocation of resources.
- *The Value Expert* enforces the allocation policies that are described as functions of value.
- *The Logger* manages the log
- *The Billing Expert* updates and queries the consumption records of the agent.

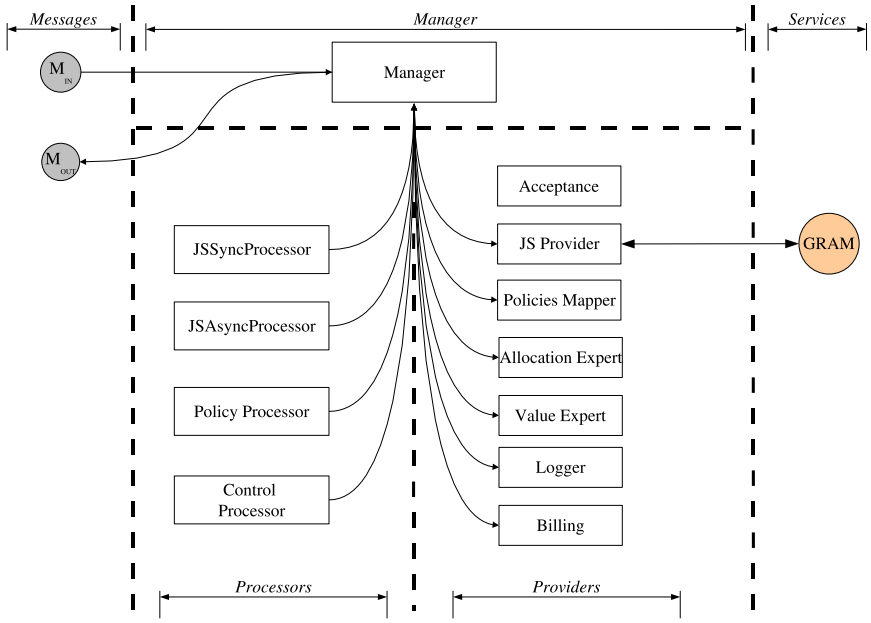


Figure 4-8: Architecture of the gt4-gram-pga agent.

The GT4-GRAM-PGA behaviour

The behaviour of agent *gt4-gram-pga* are the same of that of agent *glite-wms-pga* described in section 4.3.2.

WebCom-PGA agent

Another middleware that is encompassed by the Social Grid Agents architecture is WebCom [60].

It is connected to the metagrid space through the *WebCom Production Agent*, or *wcg-pga* for the sake of brevity, a production agent that wraps the *WebCom* workflow engine. It is an *border agent* that connects the metagrid regions with the workflow region hosting WebCom.

Although it is much more, WebCom can be considered a fledgling grid-enabled workflow engine. It offers a non-Von-Neumann programming model that automatically handles task synchronization (load balancing, fault tolerance, and task allocation at the system level) without burdening the application writer[59].

In WebCom, applications are specified as Condensed Graphs (CGs), in a manner which is independent of the execution architecture, thus separating the application and execution environments. Condensed Graphs are a mathematical abstraction capable of representing different computational models in a unified way. They consist of nodes with arcs connecting them. The nodes can be atomic nodes consisting in a single operation, or condensed graphs themselves. Thus a condensed graph consists of a series of atomic nodes that describe atomic operations and a topology that describes the relations among the different nodes.

The WebCom characteristics allow for two possible interactions with Social Grid Agents. In the first, WebCom is just yet another middleware controlled by a Production Agent that accepts jobs expressed as Condensed Graphs. A second and more interesting way to integrate Social Grid Agents and WebCom is to describe *Metagrid Jobs* as Condensed Graphs and then use WebCom as a workflow engine.

While not the only possible solution, the Unified Computational Model [60] known as Condensed Graphs [58] is ideal for expressing such workflows, where its implementation (WebCom [60, 48, 67]) allows its execution with a range of evaluation policies, e.g. eagerly or lazily.

To represent a metagrid job we can use the mathematical model of condensed graphs and the grid job description languages. In this case the representation of a

metagrid job can encompass two different layers: a topology layer that describes the dependencies among the various grid jobs (expressed in Condensed Graphs) and a grid layer that describes the grid jobs. The WebCom production agent can then be used as a workflow engine that executes a metagrid job in the metagrid environment.

Assessment of value and price

To implement a flexible framework for price and value determination, a WSDM [66] based grid management framework, Grid4C [82], has been employed. Grid4C is the subject of a thesis by Keith Rochford within my host research group in Trinity College Dublin, and this section is joint work.

Grid4C [82] is intended for command and control of grids. At this time, Grid4C agents have been developed solely for the gLite WMS service. This section therefore applies only to this type of middleware.

The interoperability between Social Grid Agents and Grid4C management endpoints allows a two-way exchange of commands and information. Grid4C endpoints can be used by Social Grid Agents as both sensors and actuators on the Grid fabric. They provide Social Grid Agents with information describing the production parameters of the various resources such as average waiting time, success rate and the like. The value, and ultimately the price, of the resources can then be extrapolated from these measured parameters.

The flexibility of Social Grid Agents in implementing complex social topologies also allows for trusted third parties to perform these measurements so that the price of the resources can be accepted by all those who trust the measurements actor.

The architecture of the interaction between SGA agents and Grid4C agents is shown in Figure 4-9. The different agents access the underlying Grid Services through different interfaces. SGAs access the gLite WMS functionalities through the JAVA™ API interfaces while Grid4C Management Endpoints employ agents residing on the managed resources for data collection and resource administration.

SGA agents employ Grid4C endpoints in order to take advantage of their management functionalities and the information they are able to obtain from the Grid

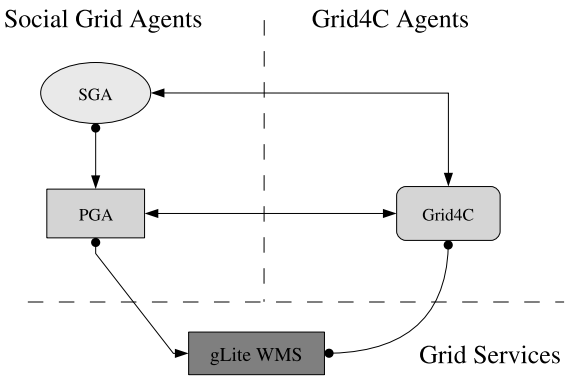


Figure 4-9: Interaction architecture among SGA and Grid4C agents.

Services. SGA agents have limited interfaces to the Grid Resources (in the gLite-WMS case the interface is currently limited to the UI API) and can make extensive use of the information provided by the Grid4C agents that, on the contrary, have much richer interfaces to the Grid services. The information obtained via the Grid4C agents is used to assess the quality of the services and define a value both of the service provider and the specific service being performed.

As discussed in Section 3.7 the value of a service can be defined as a function of two different sets of metrics: service metrics that provide information on the service and resource metrics that provide information on the resources that have been used by a specific execution of a service. The value of a service is described as $V = f(m_s, m_r)$ where m_s represent the metrics of the service and m_r are the metrics representing the actual resource consumption. For the specific case of the gLite WMS (that is the testbed for the SGA-Grid4C interaction) the metrics are described in Table 4.3:

A social agent: GLITE-WMS-SGA

The *gLite Wms Social Agent*, or *gLite-wms-sga* for the sake of brevity, is a social agent that uses and/or controls one or more *gLite Wms Production Agents*, its architecture

ServiceMetrics	*ResourceMetrics*
Operational Status and TCP response time	The Exit Status for a given job ID
Average Waiting Time for last n Jobs	The Wall-clock Time for a given job ID
Average Waiting Time since a given date	The CPU count for a given job ID
Average Match Time for last n Jobs	
Average Match Time since a given date	
Average Clear Time for last n Jobs	
Average Clear Time since a given date	
Percentage of the last n jobs aborted	

Table 4.3: Metrics for gLite Workload Management System.

and functionalities are identical to Social Grid Agents that control other production agents such as the *Gt4 Gram Production Agents*. It is a *native metagrid agent* as it resides in the metagrid region.

As for a production agent, it reacts to *action-related messages* and *control-related messages*.

The architecture of *gLite-wms-sga* is shown in Fig. 4-10; it comprises a *manager*, three *processors* and several *providers*, some of which are connected to external *Production Agents*.

The GLITE-WMS-SGA processors

The processors are :

- *The JPSyncProcessor*, a synchronous service that performs the purchase of synchronous execution of jobs.
- *The JPAsyncProcessor*, an asynchronous service that performs the purchase of asynchronous execution of jobs.
- *The Policy Processor*, a synchronous processor that checks and enforces all the policies related to the job submission
- *The Control Processor*, a synchronous processor that checks the authorization of messages that request modifications in the overall policies of the agent

The GLITE-WMS-SGA providers

The providers are :

- *The Acceptance provider* performs some standard operations at the reception of the message, such as logging the event and filling the implicit authentication data. As in all agents, in the case of a standard GT4 security context, the sender of the message is authenticated by default using its X509 certificate.
- *The DC provider* constitutes the interface to all *Production Agents* which are directly controlled.
- *The PUB provider* constitutes the interface to all *Production Agents* which are reachable through a *Pub topology*.
- *The SP provider* constitutes the interface to all *Production Agents* the services of which can be purchased through a *Simple Purchase* mechanism.
- *The Policies Mapper* maps messages, sender id and modalities into a set of policies.
- *The Price Expert* defines the price as a function of value and social relations.
- *The Logger* manages the log
- *The Endowment Manager* manages the endowment of the Agent. This provider may be connected to an external service as a bank.
- *The Discovery Service* searches in Markets and Indexes.

The GLITE-WMS-SGA behaviour

As for PGAs, SGAs are deliberately simple in concept but their behaviour can be complex. Let us examine the most common behaviour of this particular agent.

A message similar to the one below will map to an *Asynchronous Job Purchase Processor*.

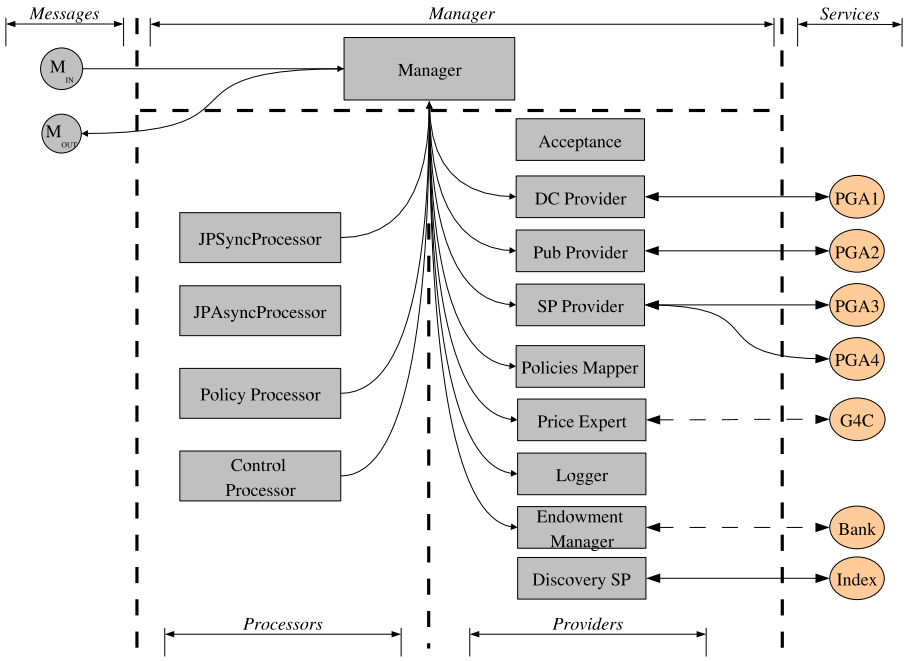


Figure 4-10: Architecture of the gLite-wms-sga agent.

```
[
 ...
 action =
 [
    ...
    name = \"simple purchase\";
    action =
    [
     modality = \"asynchronous\";
     name = \"job execution\";
     object =
     [
      type == \"lcg2\");
      ...
 ...
];
```

The initial sequence of actions defined by the *JPAsyncProcessor* can be performed synchronously as they are of short and predictable duration.

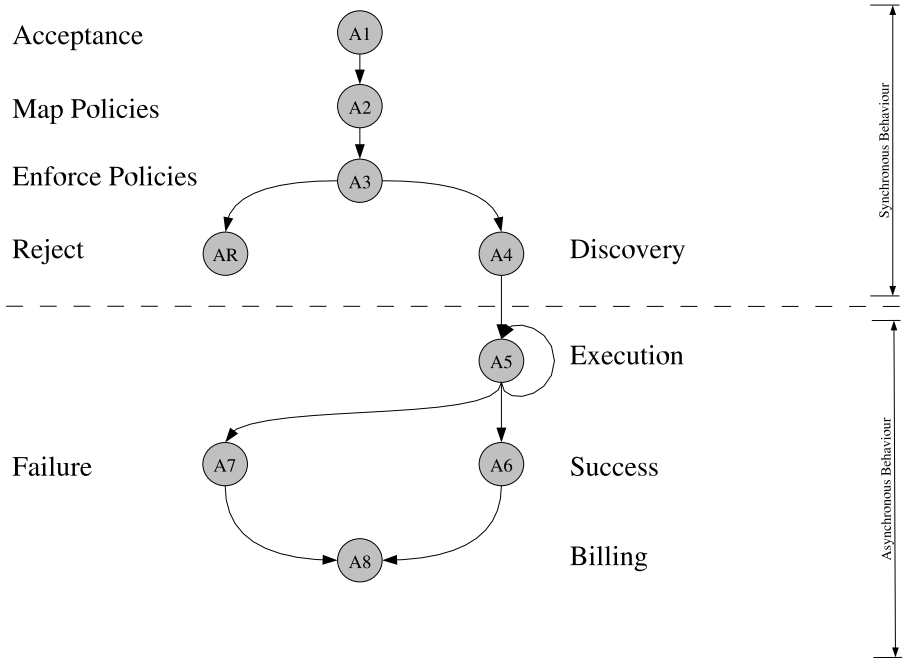


Figure 4-11: Actions performed by the Asynchronous Job Purchase Processor.

The sequence of actions of the asynchronous job submission processor is illustrated in Fig. 4-11. It comprises:

- Acceptance: a preliminary step that writes in ClassAd format the data that is implicit by the fact that the message has been accepted by the GT4 Container. These are: date and time of the reception of the message and the X509-based authentication of the sender of the message.

- Map Policies: this step defines the sets of policies that must be enforced for the job submission.

- Policies enforcement: this step enforces the set of policies defined in the previous step.

- Discovery: this step finds a suitable resource for the execution of the service. The candidate Service Providers are determined and ranked by *"Selection Policies"* that are specific to the Social Layer. The discovery step can be simple

Action	*Processor*	*Provider*
Acceptance	-	Acceptance
Map Policies	-	Policies Mapper
Enforce Policies	Policy Processor	-
Job Execution	-	DC Provider or Pub Provider or SP Provider
Billing	-	Billing

Table 4.4: Action to Provider map in the Job Purchase Asynchronous Processor.

when it involves only the Production Agents that are already known to the manager or more complex and involve external indexing agents and a market.

- Execution: controls the execution of the service on the provider that was chosen in the discovery step.

All these actions are marked as *external*; when they are to be performed then the manager attempts to find a proper provider. Table 4.4 shows the mapping performed.

All but the *Policy Processor* are simple providers that perform atomic actions. The policy processor, on the other hand, can enforce policies in a flexible order that can describe different priorities and inter-dependencies. The current behaviour of this policy processor is shown in Figure 4-12.

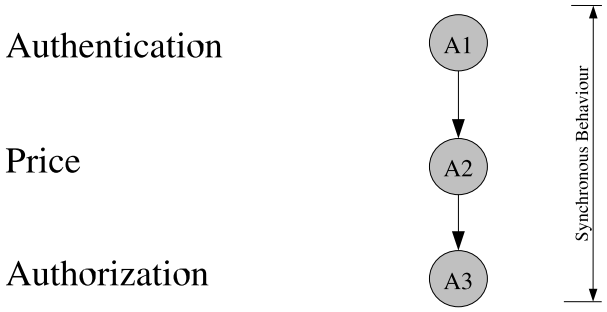


Figure 4-12: Actions performed by the Policies Processor in the gLite Wms Social Agent.

Again, as all the actions of this processor are defined as external, the manager is responsible for finding a suitable provider. Table 4.5 shows the mapping performed.

Action	*Processor*	*Provider*
Authentication	-	Authentication Provider
Allocation	-	Resource Expert
Price	-	Price Expert
Authorization	-	Authorization Expert and Endowment Manager

Table 4.5: Actions to Providers map for the Policies Processor in the gLite Wms Social Agent.

The Endowment manager shown if Figure 4-10 may use a remote bank agent. The discovery service shown in the same figure may use external indexing services.

The Bank

The *Bank* is a social agent that maintains accounts for different Social Agents. This may be required when there is no direct trust link between agents or when there is need of a centralized authority that manages the amount of credits available.

The *Bank* is a *native metagrid agent* as it resides in the metagrid region.

The architecture of the *bank* is shown in Fig. 4-13; it comprises a *manager*, three *processors* and several *providers*.

While a bank can have many different ways in which transfers are executed and accounts are managed, the current SGA bank supports only one simple policy. Any user can create an account that is defined by an amount of credits and by a list of other agents that can lodge or retrieve credits through transactions.

The Bank processors

The processors are :

- *The Transaction Processor*, a synchronous service that manages the transaction being requested.
- *The Policy Processor*, a synchronous processor that checks and enforces all the policies related to banking

- *The Control Processor*, a synchronous processor that checks the authorization of messages that request modifications in the overall policies of the agent

The Bank providers

At present, the only provider that is specific to a Bank agent is the Account manager that keeps track of the different accounts and performs the requested, authorized transactions.

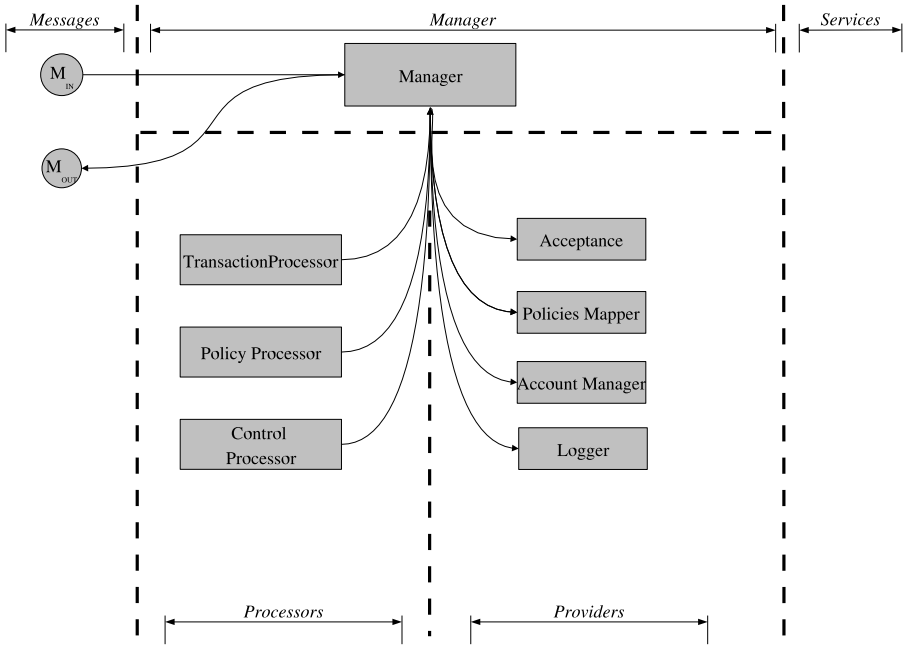


Figure 4-13: Architecture of a bank agent.

Indexing agent

Indexing agents are flexible entities that offer, as the name suggests, an *indexing service*. An indexing service may vary depending on the policies it is defined by. A *yellow page* indexing service will list all service providers whose description matches that of the query. By adding price policies and ranges to the description of the service and to the queries a *yellow page* indexing service can be transformed into a market where services can be ranked by their price.

The *Indexing Agent* is a *native metagrid agent* as it resides in the metagrid region.

The architecture of an *indexing agent* is shown in Figure 4-14; it comprises a *manager*, three *processors* and several *providers*, some of which are connected to external *Social Agents*.

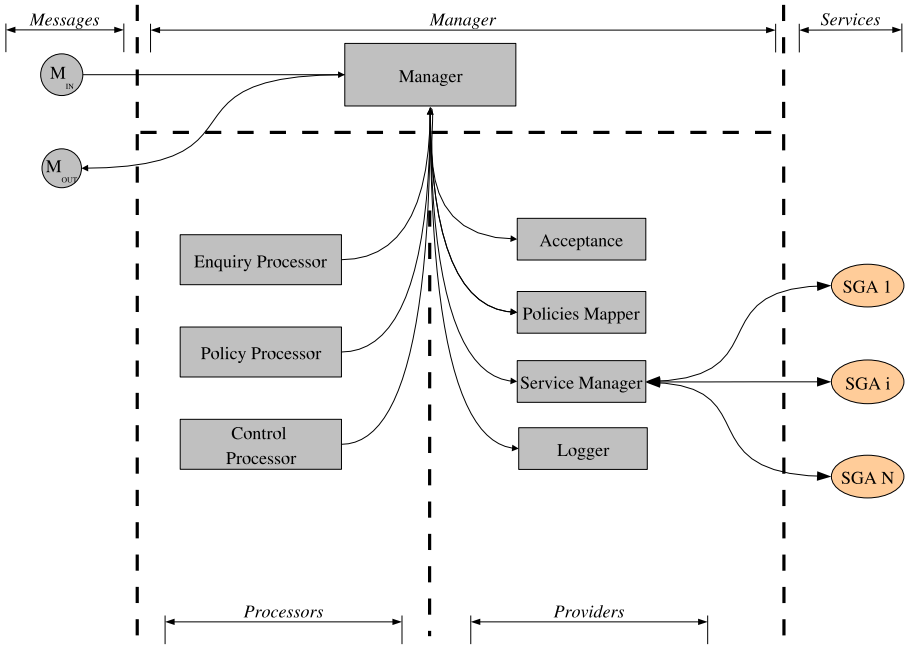


Figure 4-14: Architecture of an indexing agent.

The Indexing Agent processors

The processors are :

- *The Enquiry Processor*, a synchronous processor that receives, performs and returns the result of the queries submitted to the agent.
- *The Policy Processor*, a synchronous processor that checks and enforces all the policies related to indexing.
- *The Control Processor*, a synchronous processor that checks the authorization of messages that request modifications in the overall policies of the agent.

The Indexing Agent providers

At present, indexing agents have one specific provider: the *Service Manager* that maintain the map of the services and their descriptions.

4.3.3 An example topology

Previous sections (4.3.2, 4.3.2, 4.3.2, 4.3.2) described the implementation of some agents; this section is devoted to the description of an example topology that will be later used in Section 4.4.

This topology is described by Figure 4-15; it encompassed two sets of social and production agents and supports three different allocation philosophies: *direct control*, *Pub* and *Simple Purchase*. Social Agent SGA_1 is in control of Production Agent PGA_1, and similarly for SGA_2, PGA_2, SGA_3 and PGA_3. SGA_1 and SGA_2 have a pub relationship in which they both allow each other to access part of their resources. Each time SGA_1 is granted access to a resource controlled by SGA_2 (which, in this case, is PGA_2), it increases the amount of resources that SGA_2 can access in the production agents that SGA_1 is in control of (in this case, PGA_1). Finally, *Client*, SGA_1 and SGA_3 can engage in *Simple Purchase* transactions where a given amount of credits is exchanged in return for the services executed.

The following three sections describe in detail the messages that orchestrate the behaviour of these agents under three different conditions: *light workload*, *pub availability* and *outsourcing*.

In the first step (*step a* in 4-16, 4-17 and 4-18), that is common to all the scenarios, a *Simple Purchase* request from agent *Client* is received by SGA_1. SGA_1 maps this message to a set of enforced policies that specify:

- The price $P_{enforced}$ that is to be charged to *Client*.
- The price $P_{outsource}$ that is to be paid to SGA_3.
- How and on which data authentication is to be performed: specifically how and which agent's identities are to be authenticated. In this example it is considered that only the identity of the sender of the message has to be authenticated and that the level of authentication offered by the GT4 container is sufficient.
- How the payment is to be performed: specifically if it must be performed with the involvement of a bank and under what conditions the payment is to be

executed. In this example the payment does not involve a bank and it has to be executed upon the successful submission of a job.

- How to choose the resources to perform the request: specifically which requirements must be satisfied by the resources and in which order they must be queried. In this example, requests from *Client* will be satisfied with resources chosen in the following order:
  - Directly Controlled resources.
  - Resources available through a Pub topology.
  - Resources purchased though a Simple Purchase topology.

Given these policies, the behaviour of the system can be categorized in the following scenarios.

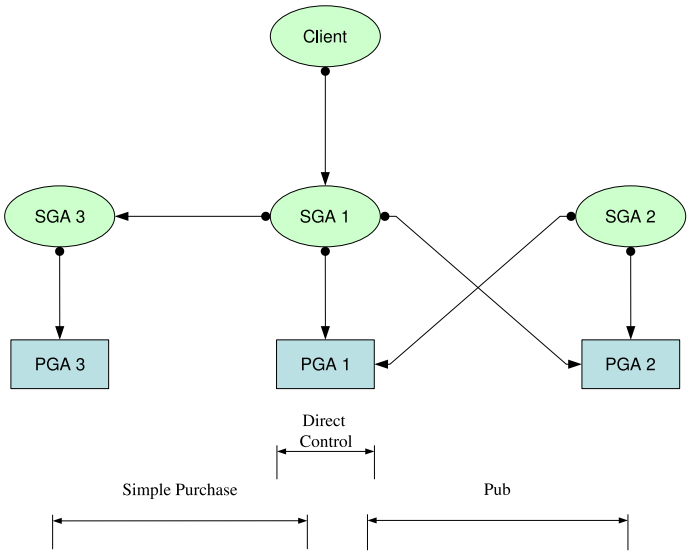


Figure 4-15: Architecture of the example topology.

Light workload

Figure 4-16 shows the interactions of the agents if the resource that is directly controlled is currently under a light workload and complies with the required *execution*

policies. In this case SGA_1 queries the availability of PGA_1 (*step b*), performs the execution (*step c*) and returns the result (*step d*).

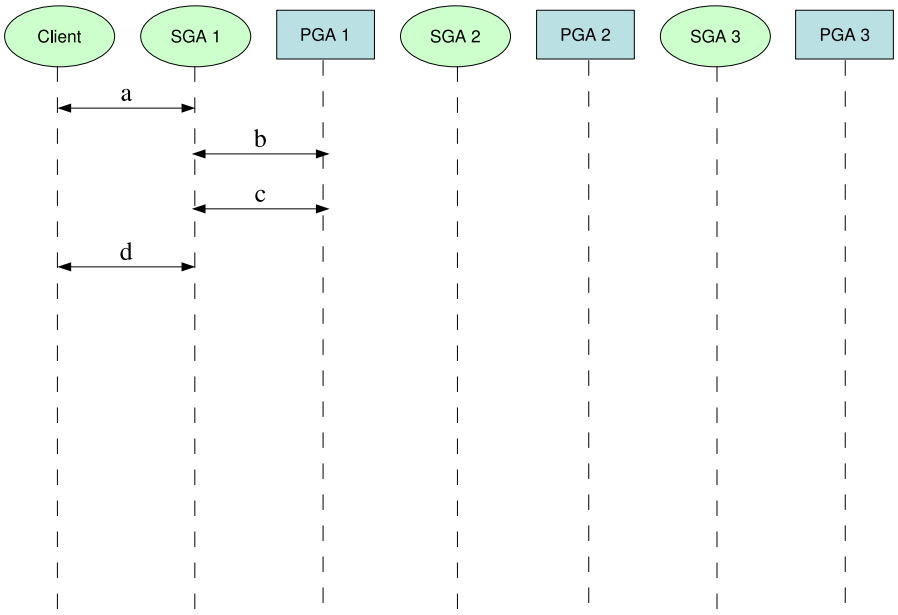


Figure 4-16: Agent's interaction under conditions of light workload.

Pub availability

Figure 4-17 shows the interactions of the agents if the resource that is directly controlled is does not comply with the required *execution policies*. After querying the availability of PGA_1 (*step b*), SGA_1 queries PGA_2 and, if the submission is successful instructs the resource it controls PGA_1 that in exchange there are additional resources that can be granted to SGA_2 in future exchanges.

Simple Purchase

Figure 4-18 shows the interactions of the agents if both the directly controlled resources (*step b*) and the pub resources (*step c*) fail. Then the execution of the service is outsourced with a simple purchase transaction to SGA_3. This is described in *step d* to *step i*.

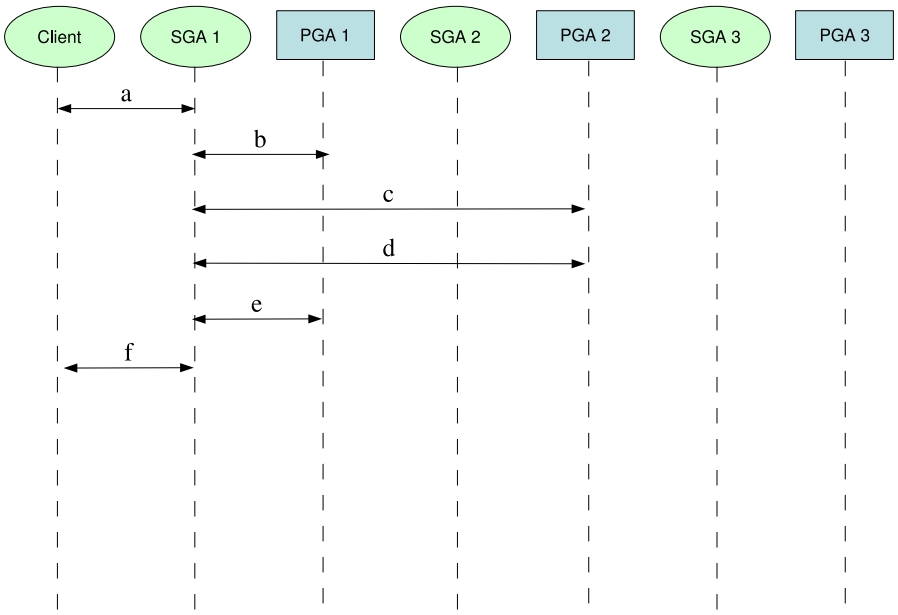


Figure 4-17: Agent's interactions in a pub topology.

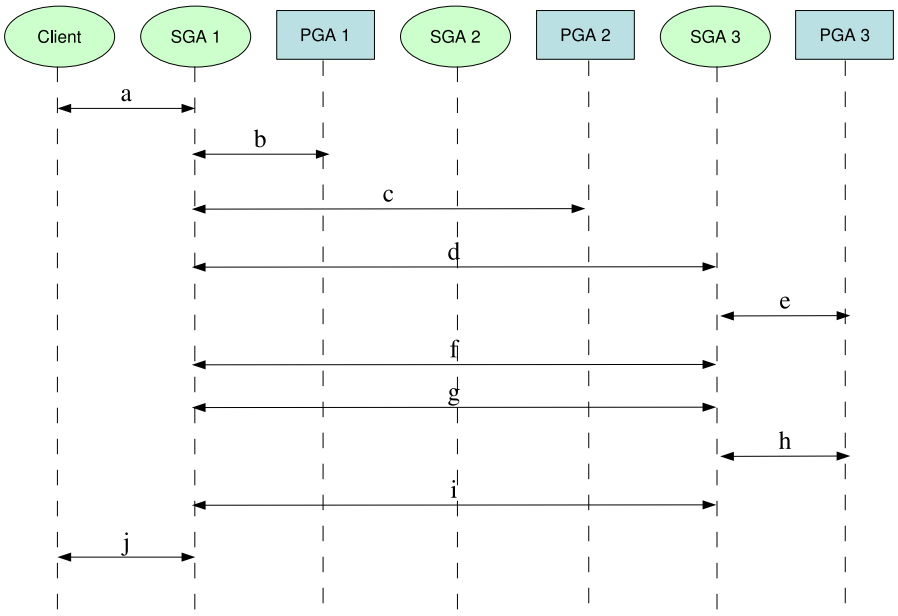


Figure 4-18: Agent's interactions during the outsourcing process.

4.3.4 An example of Policy Enforcement

This section illustrates an example of policies enforcement. It encompasses three agents (*Social Grid Agents Client* and *SGA* and the *Production Grid Agent PGA*) that control the access to a Grid resource (the Service Provider *SP*).

This example encompasses the five different steps illustrated in Figure 4-19.

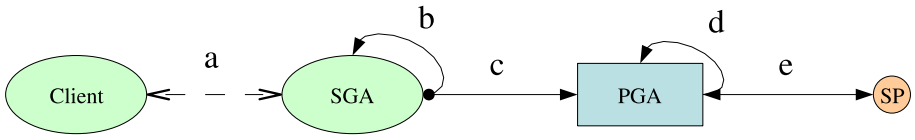


Figure 4-19: Policies and modalities enforcement.

Step a: Social Request

In *step a*, agent *Client* requests that *SGA1* executes a service with two modalities:

- A *Social Modality* (4.2) stating that the maximum price that the Client is willing to pay is 100 units
- An *Execution Modality* (4.1) stating that *metric 2* (in Figures it is referred to as m_2 for the sake of brevity) must be greater than 50 units

$$Metric_2 \geq 50 \tag{4.1}$$

$$Price \leq 100 \tag{4.2}$$

These modalities (4.1 and 4.2) define the subspaces illustrated in Figure 4-20 and are described by the ClassAd expression of Figure 4-21.

Step b: Mapping of Social Policies

In *step b*, agent *SGA1* maps this request to the following set of policies:

- A *Social Policy* (4.3) stating that the pricing policy that is applied to agent Client consists of a *Gain* of 5% of the price (a profit margin).

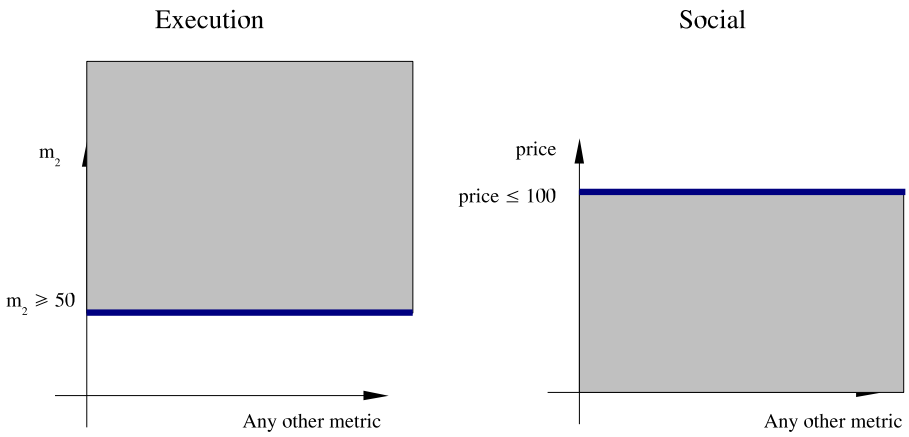


Figure 4-20: Subspace defined by the modalities requested by the Client.

- A *Social Policy* (4.4) stating that the price is to be calculated as the value plus the gain.

$$Gain = 0.05 * Price \tag{4.3}$$

$$Price = Value + Gain \tag{4.4}$$

The ClassAd expression resulting from the mapping of the Social Policies in Agent SGA1 is described in Figure 4-22.

Step c: Enforcement of Social Policies

In *step c*, agent *SGA1* enforces a set of policies that are a combination of the modalities requested by the Client and the policies that SGA1 applies to Client.

- A *Social Policy* (4.6) stating that the enforced price is the one that was requested by the Client
- A *Social Policy* (4.7) stating that the price is to be computed according to the policies defined by SGA1
- A *Social Policy* (4.7 and 4.8) stating that the price must comply to the policies defined by SGA1

These policies (4.5, 4.6, 4.7 and 4.8) define the subspace illustrated in Figure 4-23 and constitute the modalities with which SGA1 requests the services of agent PGA1. The ClassAd expression describing this is illustrated in Figure 4-24.

$$Metric_2 \geq 50 \tag{4.5}$$

$$Price \leq 100 \tag{4.6}$$

$$Gain = 0.05 * Price \tag{4.7}$$

$$Price = Value + Gain \tag{4.8}$$

Step d: Mapping of Production Policies

In *step d*, agent *PGA1* receives a request from agent *SGA1* and maps this request to the following set of policies:

- A set of *Production Policy* (4.9 and 4.10) stating bounding metrics a and b to be less than 10 and 100 units.
- A *Production Policies* (4.11) that define how the value must be computed from the metrics.
- A *Production Policy* (4.12) that define how the value of the service must be computed from the metrics.
- A *Production Policies* (4.13) that define a clause (independent from that inferred from price) on the maximum value that can be allocated.

$$metric_1 \geq 10 \tag{4.9}$$

$$metric_2 \geq 100 \tag{4.10}$$

$$value = 10 * metric_1 + 2 * metric_2 + value_{service} \tag{4.11}$$

$$value_{service} = 10 + metric_3 \tag{4.12}$$

$$value \leq 200 \tag{4.13}$$

These policies are described in the ClassAd expression of Figure 4-25.

Step e: Enforcement of Production Policies

In *step e*, agent *PGA1* enforces a set of policies that are a combination of the modalities requested by agent SGA1 and the policies that PGA1 applies to SGA1.

- An *Allocation Policy* that enforces the allocation policies requested by PGA1.
- A *Value Policy* stating that both value modalities and value policies must be respected.

These policies define the subspaces illustrated in Figure 4-27. The ClassAd expression describing this is shown in Figure 4-26

Finally all the policies are combined together to define the *Execution Modalities* that describe the execution request that agent PGA1 forwards to the Service Provider *SP* it is in control of.

$$metric_2 \geq 50 \tag{4.14}$$

$$Price \leq 100 \tag{4.15}$$

$$Gain = 0.05 * Price \tag{4.16}$$

$$Price = Value + Gain \tag{4.17}$$

$$metric_1 \leq 10 \tag{4.18}$$

$$metric_2 \leq 100 \tag{4.19}$$

$$value = 10 * metric_1 + 2 * metric_2 + value_{service} \tag{4.20}$$

$$value_{service} = 10 + metric_3 \tag{4.21}$$

$$metric_1 \leq 200 \tag{4.22}$$

These modalities define the subspace of Figure 4-28.

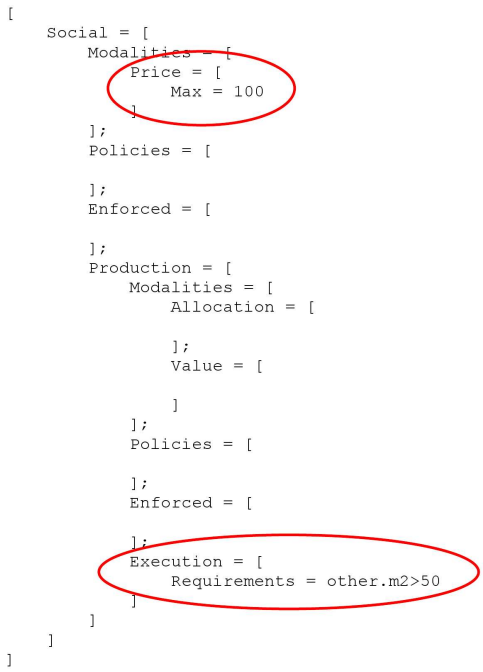
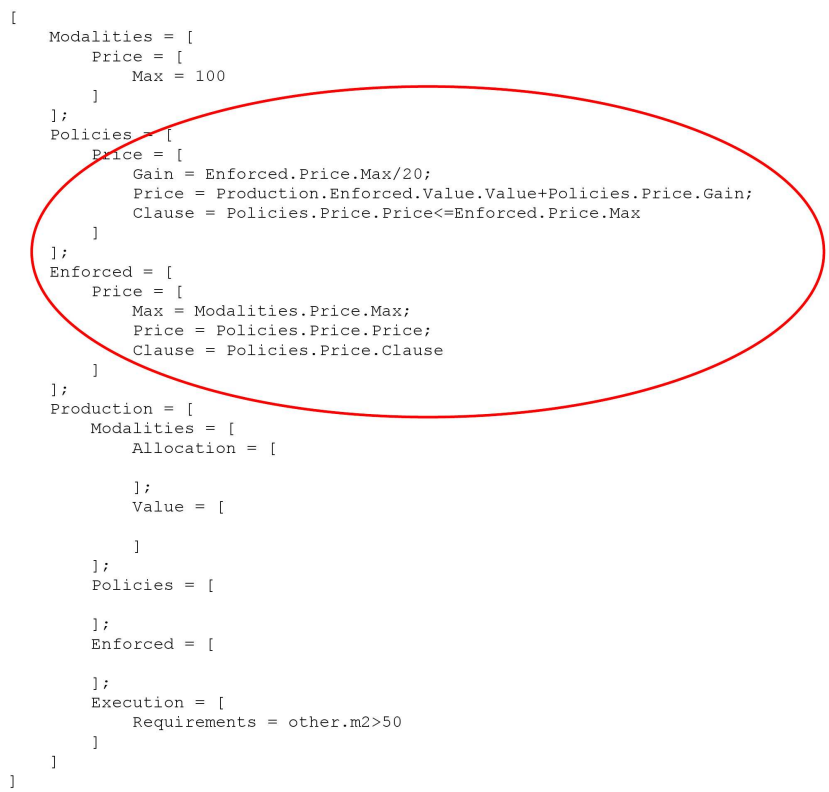


Figure 4-21: ClassAd Expression defining the modalities requested by the Client.



```
[
    Modalities = [
        Price = [
            Max = 100
        ]
    ];
    Policies = [
        Price = [
            Gain = Enforced.Price.Max/20;
            Price = Production.Enforced.Value.Value+Policies.Price.Gain;
            Clause = Policies.Price.Price<=Enforced.Price.Max
        ]
    ];
    Enforced = [
        Price = [
            Max = Modalities.Price.Max;
            Price = Policies.Price.Price;
            Clause = Policies.Price.Clause
        ]
    ];
    Production = [
        Modalities = [
            Allocation = [

            ];
            Value = [

            ]
        ];
        Policies = [

        ];
        Enforced = [

        ];
        Execution = [
            Requirements = other.m2>50
        ]
    ]
]
```

Figure 4-22: ClassAd expression after the social policies mapping.

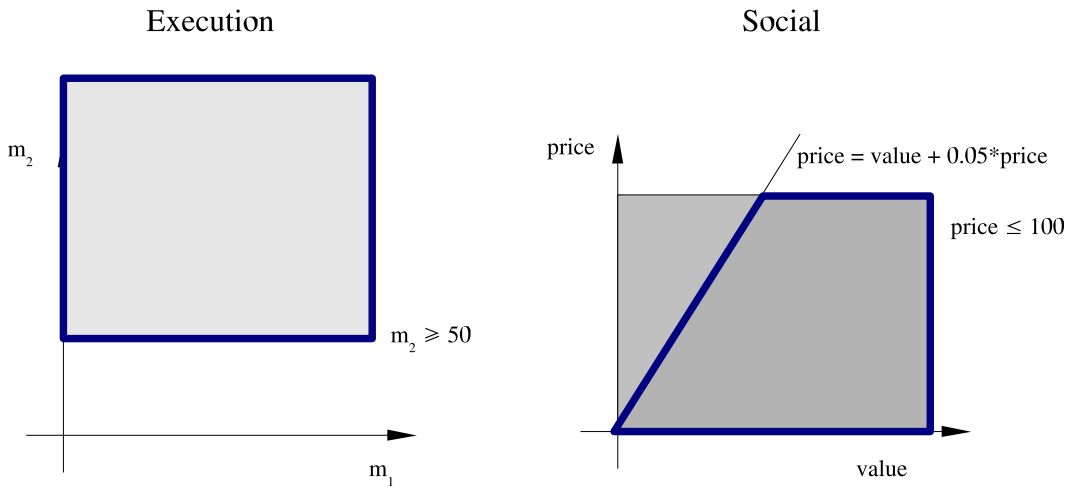


Figure 4-23: Subspace defined by the social policies.

```
[
    Modalities = [
        Price = [
            Max = 100
        ]
    ];
    Policies = [
        Price = [
            Gain = Enforced.Price.Max/20;
            Price = Production.Enforced.Value.Value+Policies.Price.Gain;
            Clause = Policies.Price.Price<=Enforced.Price.Max
        ]
    ];
    Enforced = [
        Price = [
            Max = Modalities.Price.Max;
            Price = Policies.Price.Price;
            Clause = Policies.Price.Clause
        ]
    ];
    Production = [
        Modalities = [
            Allocation = [

            ];
            Value = [
                Clause = true&&Enforced.Value.Value+100/20<=100
            ]
        ];
        Policies = [

        ];
        Enforced = [

        ];
        Execution = [
            Requirements = other.m2>50
        ]
    ]
]
```

Figure 4-24: ClassAd expression after the social policies enforcement.

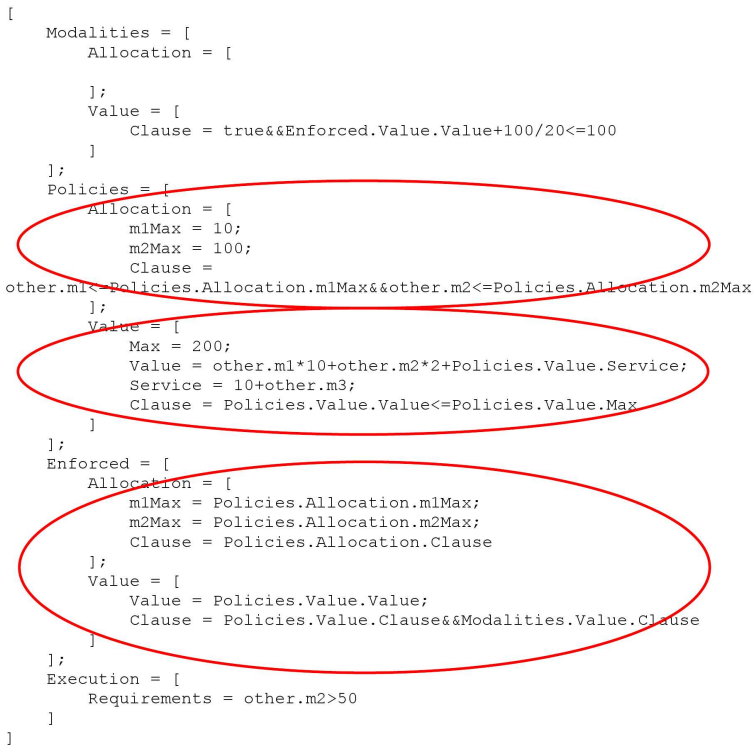


Figure 4-25: ClassAd expression after the production policies mapping.

```
[
    Modalities = [
        Allocation = [

        ];
        Value = [
            Clause = true&&Enforced.Value.Value+100/20<=100
        ]
    ];
    Policies = [
        Allocation = [
            m1Max = 10;
            m2Max = 100;
            Clause = other.m1<=Policies.Allocation.m1Max&&other.m2<=Policies.Allocation.m2Max
        ];
        Value = [
            Max = 200;
            Value = other.m1*10+other.m2*2+Policies.Value.Service;
            Service = 10+other.m3;
            Clause = Policies.Value.Value<=Policies.Value.Max
        ]
    ];
    Enforced = [
        Allocation = [
            m1Max = Policies.Allocation.m1Max;
            m2Max = Policies.Allocation.m2Max;
            Clause = Policies.Allocation.Clause
        ];
        Value = [
            Value = Policies.Value.Value;
            Clause = Policies.Value.Clause&&Modalities.Value.Clause
        ]
    ];
    Execution = [
        Requirements =
other.m2>50&&(other.m1<=10&&other.m2<=100)&&(other.m1*10+other.m2*2+(10+other.m3)<=200&&(true&&other.m1*10+
other.m2*2+(10+other.m3)+100/20<=100))
    ]
]
```

Figure 4-26: ClassAd expression after the production policies enforcement.

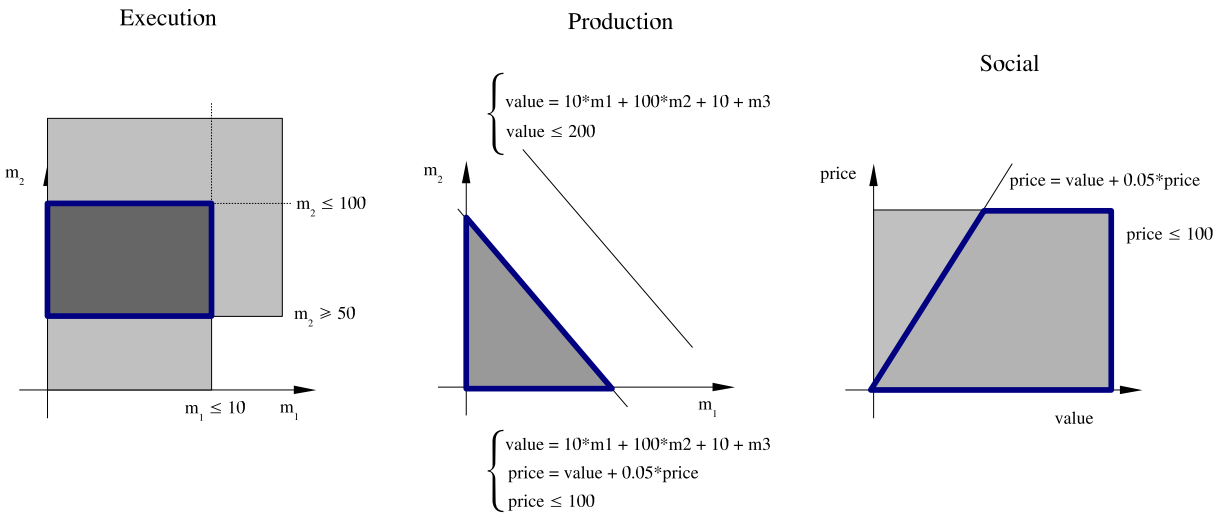


Figure 4-27: Subspace defined by the production policies.

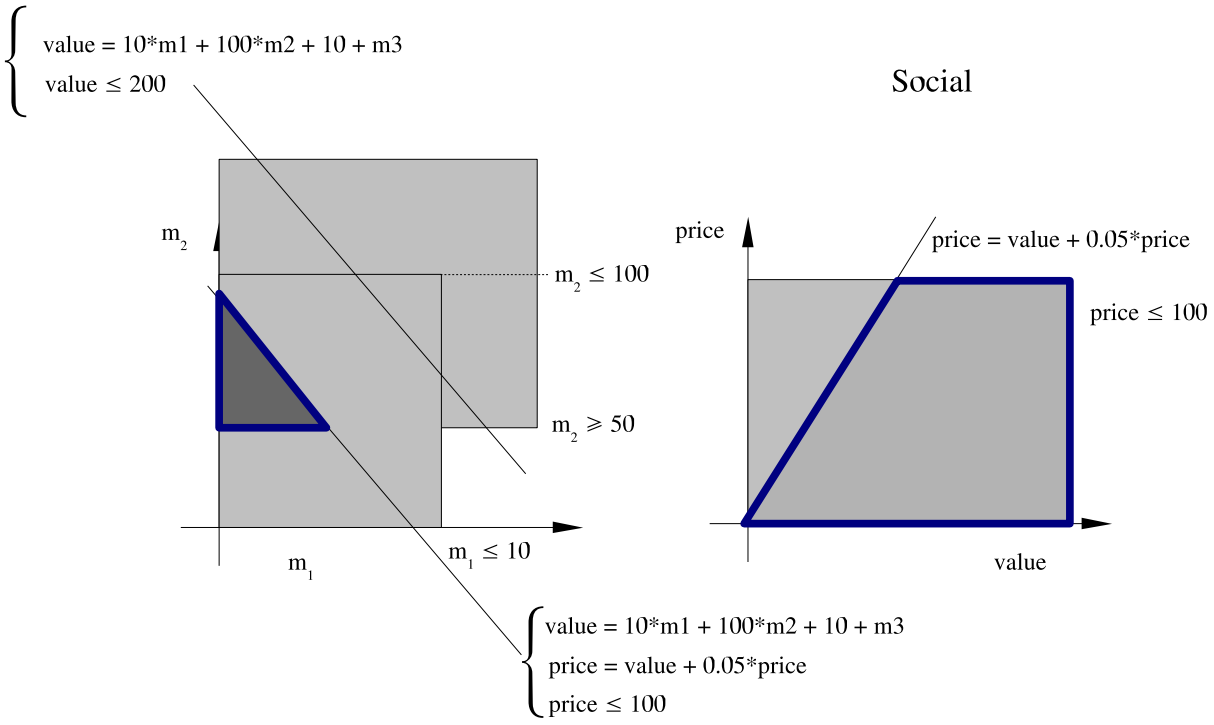


Figure 4-28: Subspace defined by the execution policies.

4.4 Experiments

The most immediate research questions that should be answered can be informally stated as: *Do Social Grid Agents degrade the performance of resources they control ? How scalable are Social Grid Agents ?* and, finally *How do they behave ?* In order to answer these questions, three experiments will now be described.

4.4.1 Reliability and Efficiency

The *reliability* and *efficiency* experiments aim at answering the first set of research questions. To assess the level of degradation of the service induced by the agents I set up an experiment with the abstract testbed described in Figure 4-29. Here a *Tester* submits workload to a resource through three different paths:

- The *Direct Submission* where the workload is submitted directly to the resource.
- The *Production Access* where the workload is submitted through a Production Agent.
- The *Complete Topology* where the workload is submitted through a Social and a Production Agent.

For each of the submission paths, four different metrics are gathered:

- The *Submission Reliability* that is the percentage of jobs successfully submitted.
- The *Submission Time* that is the average time it takes to submit a job.
- The *Submission Efficiency* that is the percentage submission time relative to the submission time for direct submission, calculated as $\frac{Direct\ Submission\ Time}{Actual\ Submission\ Time}$
- The *Execution Reliability* that is the percentage of jobs successfully executed.

These set of experiments were run on the gLite border.

Figure 4-30 shows the real testbed that was used to evaluate the gLite border. The Workload was submitted from a machine (*tgui.testgrid*) that hosts the User Interface of the *TestGrid* [31] infrastructure available in my host research group.

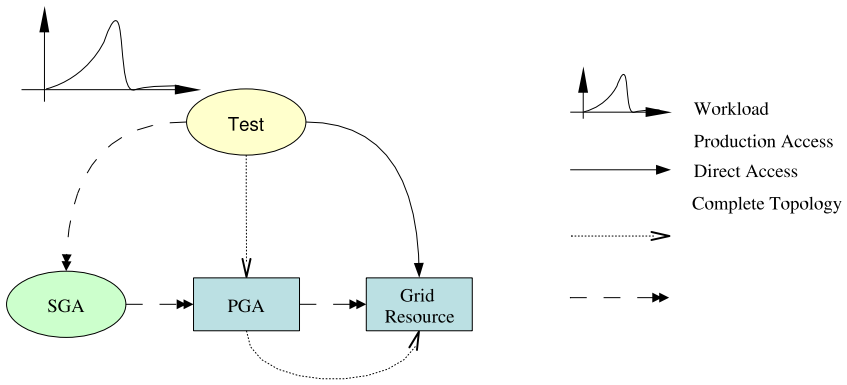


Figure 4-29: Abstract view of the reliability and efficiency experiments.

The agents were hosted in the GT4 containers of two different XEN [24] virtual machines (*lcg2-b1* and *gt4-b1*) hosted on a machine called *tg18* (a Tyan Tomcat motherboard with a 2.4 GHz Intel P4 CPU, 2GByte of memory, and 1Gbps network interface).

Tables 4.6, 4.7 and 4.8 recapitulate the data showing that the degradation introduced by the Social Grid Agents is acceptable for a prototype.

Appendix A contain detailed graphs of the experiments in Figures A-1, A-2, A-3 and A-4.

Submission and Execution Reliability are both unaffected and only the Submission Efficiency was degraded by up to 10%.

The reason the Submission Efficiency linearly degrades with the number of consecutive jobs is that new threads for managers and processors are issued each time a job is received and at least one processor thread runs until the job is completed. Consecutive submissions thus result in a linearly increasing workload responsible for the degradation of the service. This characteristic is also responsible for the degradation of the Efficiency in the experiment in section 4.4.2.

It is worth noticing that the result of $98\% \pm 4\%$ referred to in Table 4.8 obtained in the 100 jobs experiment with the direct submission was due to a temporary problem in the TestGrid middleware.

Metrics	*Direct Submission*	*Production*	*Complete Topology*
Submission Reliability	100% ± 0%	100% ± 0%	100% ± 0%
Submission Time	7.79 ± 0.23 sec	8.21 ± 0.19 sec	8.35 ± 0.23 sec
Submission Efficiency	100.00%	94.89%	93.36%
Execution Reliability	100% ± 0%	100% ± 0%	100% ± 0%

Table 4.6: Degradation of service on the gLite border (10 consecutive jobs repeated 5 times).

Metrics	*Direct Submission*	*Production*	*Complete Topology*
Submission Reliability	100% ± 0%	100% ± 0%	100% ± 0%
Submission Time	7.92 ± 0.30 sec	8.40 ± 0.32 sec	8.78 ± 0.08 sec
Submission Efficiency	100.00%	94.25%	90.84%
Execution Reliability	100% ± 0%	100% ± 0%	100% ± 0%

Table 4.7: Degradation of service on the gLite border (50 consecutive jobs repeated 5 times).

Metrics	*Direct Submission*	*Production*	*Complete Topology*
Submission Reliability	100% ± 0%	100% ± 0%	100% ± 0%
Submission Time	7.85 ± 0.15 sec	8.62 ± 0.58 sec	8.99 ± 0.67 sec
Submission Efficiency	100.00%	91.05%	87.35%
Execution Reliability	98% ± 4%	100% ± 0%	100% ± 0%

Table 4.8: Degradation of service on the gLite border (100 consecutive jobs repeated 5 times).

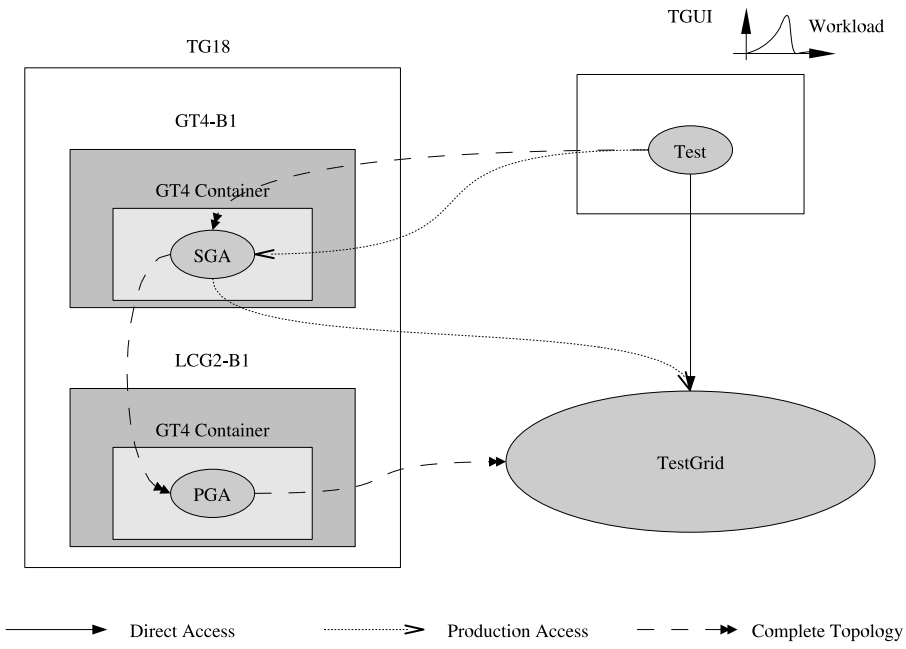


Figure 4-30: Abstract view of the reliability and efficiency experiments.

4.4.2 Scalability

These scalability experiments aim at demonstrating how well the current of Social Grid Agents can be scaled with regard with: the number of concurrent clients, number of agents and number of resource.

The current implementation consists in a one-to-one mapping of a production agent to a resource, thus the main question to be addressed is the scalability of one agent with regard to the numbers of concurrent requests it receives from other agents (modelled as clients).

The experiments consists of the same experiment run in two different testbeds: a *Real Testbed* consisting of a complete agent with its GT4 skeleton and a *Synthetic Testbed* that tests only the architecture of the agent's brain.

Real Testbed

The abstract experiment testbed is shown in Figure 4-31.

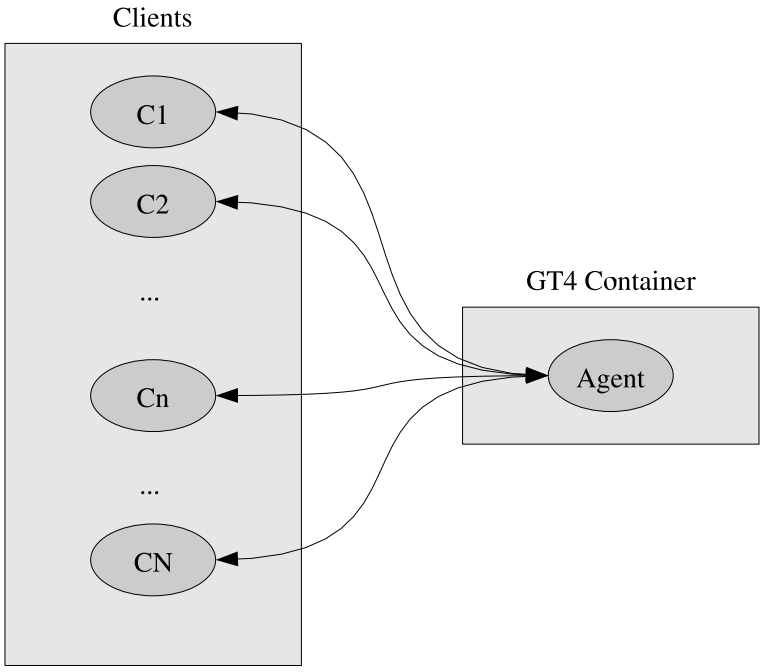


Figure 4-31: Testbed for the scalability experiment.

The agent capability to service requests is tested through the analysis of reliability (percentage of failed actions) and efficiency (time needed to perform actions) metrics.

Parameter	*Value*
Experiment duration	300 seconds
Average of client delay	2 seconds
Standard Deviation of client delay	1 second
Average of test agent execution time	2.5 seconds
Standard Deviation of test agent execution time	2.5 seconds
Workload	From 0 to 100 concurrent clients

Table 4.9: Parameters of the scalability experiment.

As the only technology common to all agents is their GT4 skeleton, this experiment utilizes a *test agent* that does not rely on any additional technology. The test agent performs an action that has a random duration. The testbed consists of a *client load* that simulates a population of other independent agents trying to connect to the test agent. Each of these *clients* is an independent thread that will connect to the test agent; it will delay for a time that is a parameter of the experiment and then it will connect again. The delay time for each client is computed randomly to ensure a uniform workload; the average and standard deviation of this delay are two of the parameters of the example.

One of these clients is an *instrumented client* that keeps track of reliability and efficiency of each call and then saves the related data to files.

The experiment was executed with the parameters in Table 4.9:

The execution time ranging from 0 to 10 seconds cover the execution time of most actions taken by SGAs, the duration of the experiment (300 seconds) is long enough to gather data with statistical relevance. The Client delay ranging from 1 to 3 seconds allows for a workload with reasonable statistical characteristics.

The experiments results are summarized in Table 4.10. Appendix B details the experiment metrics in graphs B-1, B-2, B-3 B-4.

These results highlight that the efficiency with which the agents react to the increasing number of clients degrades linearly and it is mainly due to the *dispatch time* (the time that it takes for the message to reach the service). The reliability was

Number of Clients	*Execution Time*	*Total Time*
1	2.26 ± 1.42 seconds	1.51 ± 1.43 seconds
10	2.20 ± 1.21 seconds	3.04 ± 3.23 seconds
50	2.31 ± 1.26 seconds	13.03 ± 2.85 seconds
100	2.62 ± 1.42 seconds	26.32 ± 9.54 seconds

Table 4.10: Results of the scalability experiment.

not affected by the increasing number of parallel clients as all messages were replied to.

The degradation of execution time is relatively small (15% for 100 concurrent accesses); this shows that the native agent architecture copes reasonably well with an increasing parallel workload. On the other hand, the dispatch time is severely degraded. This is mainly due to three reasons: the startup time of the concurrent threads of the managers and processors, the operations that are performed by the GT4 container each time a message is received, and the increased workload given by the threads that are already running. To investigate which one of these three reasons is the most relevant, another experiment run without the GT4 container is detailed in Section 4.4.2. This effect is related to that of experiment 4.4.1 but the overload caused by the selection and startup of the processors and by the operations performed by the GT4 container is more severe than the overload caused by those threads that are already running, thereby resulting in a greater degradation of the response time.

As the current implementation of Social Grid Agents is composed of a small number of agents this is not a severe problem but it will certainly be much more problematic in larger topologies and has thus to be further investigated and solved.

Synthetic Testbed

In order to confirm that the degradation is mainly caused by the GT4 container, a synthetic testbed was created that excluded GT4. As for the experiment on the *Real Testbed* (see section 4.4.2), the agent capability to service requests is tested through

Parameter	*Value*
Experiment duration	600 seconds
Average of client delay	2 seconds
Standard Deviation of client delay	1 second
Average of test agent execution time	2.5 seconds
Standard Deviation of test agent execution time	2.5 seconds
Workload	From 100 to 1000 concurrent clients

Table 4.11: Parameters of the large scalability experiment on the synthetic testbed.

the analysis of reliability (percentage of failed actions) and efficiency (time needed to perform actions) metrics.

Two sets of experiments were executed to assess the scalability of the agent's brain: a small scale experiment with the same parameters as for the real testbed experiment (see table 4.9) and a larger scale experiment with the parameters shown in Table 4.11.

The synthetic testbed was implemented in the *Eclipse* [7] development platform hosted on a 1400 Mhz Intel™Pentium™M processor with 1 GB of memory running Windows XP Professional.

Detailed results for the small scale experiment results are shown in the graphs C-1, C-2, C-3 C-4 of Appendix C.

These results show how at a small scale of accesses (1 to 100 concurrent clients) the *Execution Time* remains almost constant; the *Dispatch Time* increase with the number of concurrent accesses is present but much smaller than for the real testbed and the *Return Time* is negligible. In the synthetic testbed the increase is much smaller because it lacks all the additional operations performed by the GT4 container at the reception of messages.

The *Return Time* is negligible in the synthetic environment because it just consists of function returns and in the serialization/deserialization of the message.

Detailed results for the larger scale experiment results are shown in graphs C-5, C-6, C-7 C-8 of Appendix C.

They show how the *Dispatch Time* increases significantly in the scale from 100 to 1000 concurrent clients. This highlights that the current architecture is capable of supporting up to a few hundred concurrent messages without too much degradation of its response time.

Parameter	*Value*
Minimum Size	1000
Maximum Size	100000
Number of Iterations	10

Table 4.12: Parameters of the scalability experiment on the ClassAd Mapper.

Large Scale ClassAd Maps

A key component of Social Grid Agents is the *ClassAd Mapper* (see section 3.18.2). As this component is used extensively it is important to assess its scalability. The testbed of this experiment is the same as for section 4.4.2

This experiment uses a test ClassAd Mapper that maps simple *RecordExpr* expression that contain a single value to a simple object (an instance of a *String* class).

```
[
        key = 12345;
        Requirements = true;
        Rank = 1;
]
```

The map is filled with keys ranging from 0 to the maximum map size and is queried with a *RecordExpr* expression that matches one key.

```
[
        Requirements = other.key == 12345;
        Rank = 1;
]
```

The metrics of the experiment are the time that is needed to create the entire ClassAd Mapper and the time needed to retrieve a random entry. The parameters of the experiment are shown in Table 4.12

The results of the experiment are shown in Figure 4-32 and 4-33; both the creation and the query time remain reasonable in the explored range. The longest time needed to create a map with 100,000 entries was 9.033 seconds and the longest time needed

to retrieve an entry was 2.524 seconds. The linear increase of the creation time is due to the loop needed to insert all the entries into the map whilst the linear increase in the query time is due to the fact that a match-make operation is to be performed on each key of every entry until a match is found.

The ClassAd RecordExpr composing each key was, in this experiment, very simple; the behaviour of *ClassAd Mappers* of higher complexity will combine the linear behaviour highlighted with this experiment with a characteristic of the ClassAd language: the fact that *"... a (ClassAd) expression can be evaluated in time proportional to the size of the expression"* [56].

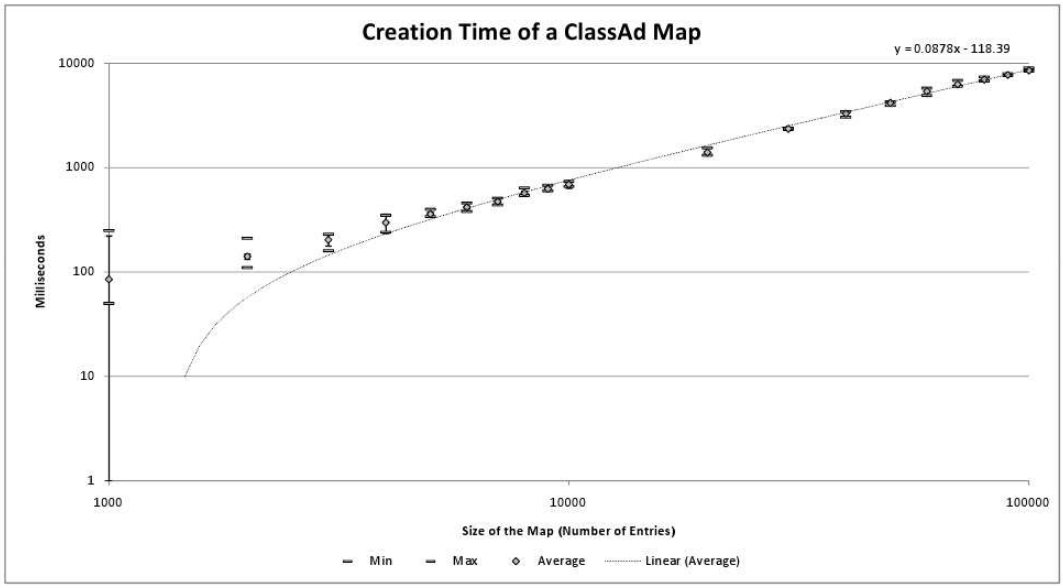


Figure 4-32: Creation Times for a Large Scale ClassAd Map.

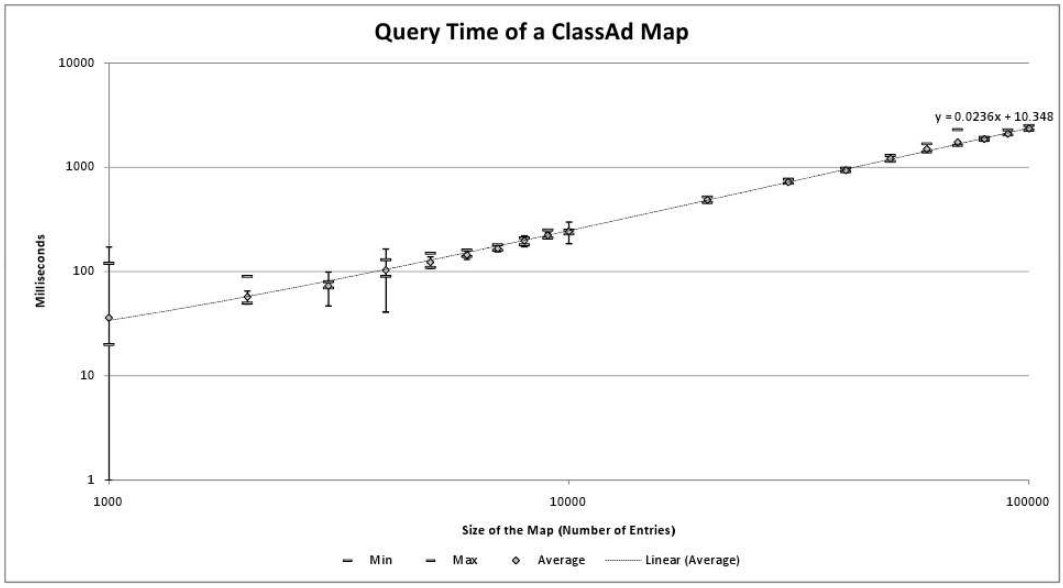


Figure 4-33: Query Times for a Large Scale ClassAd Map.

4.4.3 Agent behaviour

The behaviour experiments aim at demonstrating how different topologies of Social Grid Agents can enforce various resource allocations. This set of experiments is focused on three different topologies: the Simple Producer, the Pub and the Simple Purchase.

4.4.4 First behaviour experiment

A workload is submitted to an agent that belongs to all three topologies. The agent's behaviour is determined by a policy that states which resource is to be chosen first and, consequently, which topology is to be preferred under different conditions.

The abstract testbed for the experiment is illustrated in in Figure 4-34. The testbed is composed of six agents of which three are Production Agents (*PGA1*, *PGA2* and *PGA3*) and three are Social Agents (*SGA1*, *SGA2* and *SGA3*).

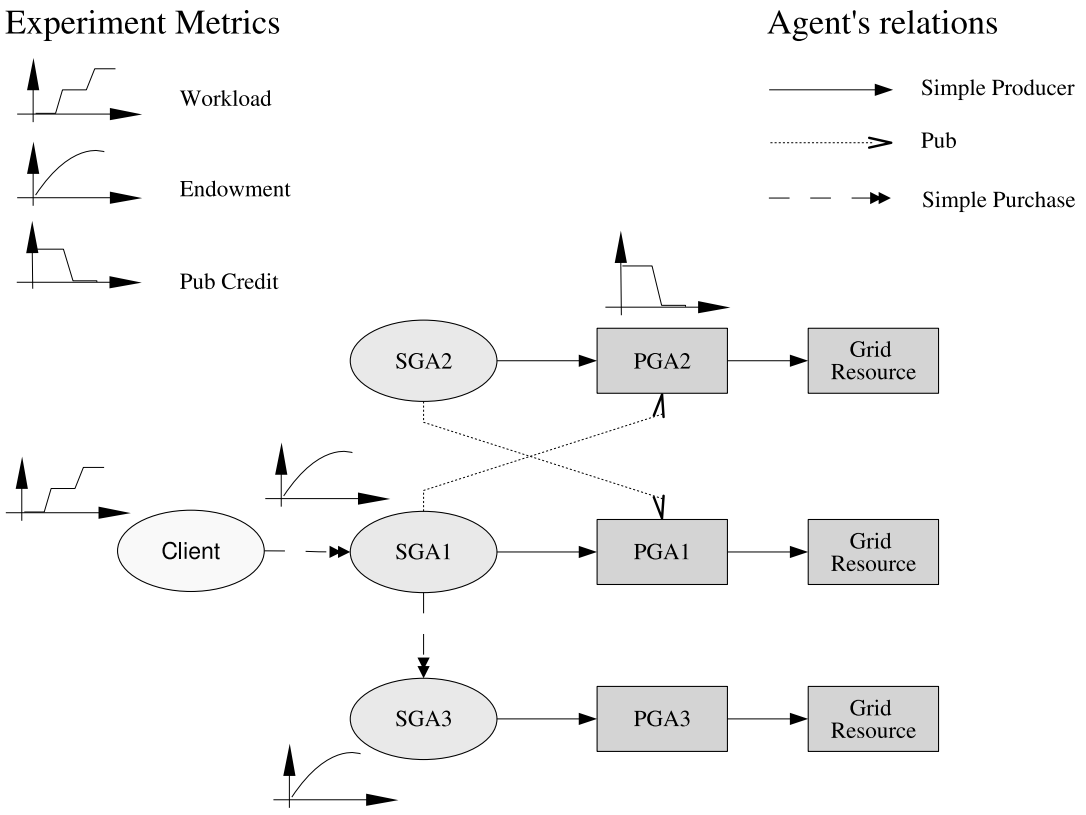


Figure 4-34: Testbed for the behaviour experiment.

Experiment Topologies

The agents are arranged in four different topologies:

- the *Simple Producer* relationship consists of a Social Agent in control of a Production Agent. If this is the only relationship an agent belongs to, then the agent can only face a workload less or equal to the resources it is in control of. In the testbed there are three of these *Simple Producer* relationships encompassing *SGA1* with *PGA1*, *SGA2* with *PGA2* and *SGA3* with *PGA3*.

- the *Pub* relationship consists of a set of social agents, *SGA1* and *SGA2*, that "lend" to each other part of their resources *PGA1* and *PGA2*. Each of the actors allow for a certain "credit" to each of the other "pub friends". This topology that allows load balancing when the different workloads do not synchronously encompass the four agents *SGA1*, *SGA2*, *PGA1* and *PGA2*.

- the *Simple Purchase* consists in the purchase of the needed services, an economic scenario that entails the transfer of credits. This topology allows leverage of heavy workloads only in the case that the agent that is purchasing the resources has a sufficient endowment. This relation encompasses two couples of agents: *Client* with *SGA1* and *SGA1* with *SGA3*.

Agent Policies

Social Grid Agent 1

Agent *SGA1* chooses between its resources (*PGA1* which it directly controls, *PGA2* which can be reached through a Pub agreement and *PGA3* which can be purchased) as dictated by the following policies.

- *Price Policy*: states that the resource can be used by anybody provided that the offered price is sufficient (10 credits per submission).

- *Execution Policy* enforced on Production Agent *PGA1*: this policy states that the resource can be used on the condition that at least one job slot is left available.

- *Selection Policy* enforced on Social Agent *SGA1*: this policy states that the first resource to be used is the one that is directly controlled (*PGA1*), then the one that is accessible through pub relations and finally, as a last resort, it can be purchased.

Production Grid Agent 2

Agent *PGA2* accepts or not jobs from *SGA1* based on the following policies:

- *Execution Policy*: states that the resource can be used on the condition that at least one job slot is left available.
- *Authorization Policy*: states that the resource can be used by *SGA1* only if the *'Pub Credits'* are greater than zero. Each time it performs a job on behalf of *SGA1* the amount of its tokens is decreased by a unit.

Social Grid Agent 3

Agent *PGA2* accepts or not jobs from *SGA1* based on the following policies:

- *Price Policy*: states that the resource can be used by anybody provided that the offered price is sufficient (10 credits per submission).

Experiment Initial Conditions

The experiment was performed using the *TestGrid* [31] infrastructure available in my host research group. The initial condition of the testbed were :

- *SGA1 Endowment* was set to 100 credits.
- *SGA3 Endowment* was set to 0 credits.
- *PGA2 Tokens*[1] was set to 5 units.

[1]Tokens are introduced in 3.2.2, they represent the number of job submissions that an agent is granted in a Pub topology

This set of initial condition was chosen to explore all the possible behaviours of the agents. The Direct Control execution policies was set to always leave a free slot so that on average it would not suffice, Pub tokens were set to a low number (and not zero) so that they could be used but not suffice for heavy workload leaving the Simple Purchase as the last viable choice.

Experiment Results

The results of the experiment are recapitulated in Figure 4-35 that shows the four sets of metrics that were collected during the experiment and in Figure 4-36, a graphical summary of the behaviour of the entire topology during the experiment. There are three main phases of the experiment: Appendix D contains Figures D-1, D-2, E-2 and D-4 that describe in greater detail the various metrics gathered during the experiment.

Figure 4-35 shows the four sets of metrics that were collected during the experiment.

- The *workload*[2] is represented by the graph on the left of the image.
- The *endowment of agent SGA1* is represented by the graph that is immediately right to the workload.
- The *endowment of agent SGA3* is represented by the graph on the bottom of the image.
- The *number of tokens granted to agent SGA1 by agent PGA2* is represented by the graph on the top of the image.

Figure 4-36 is a graphical summary of the behaviour of the entire topology during the experiment. There are three main phases of the experiment:

- In *phase A* the workload of agent SGA1 can be met entirely by the resource (Production Agent PGA1) it is in control of. As specified by its *Selection Policy* all jobs are then executed by this resource until one job slot remains free.

[2]The workload represents the overall amount of job submissions that the agent copes with

- During *phase B* the directly controlled resource PGA1 does not match any more the execution policy of having one free job slot. The Selection Policy dictates that, in this case, the Production Agent PGA2 that can be reached through a Pub topology has to be used. This behaviour is sustainable until the number of tokens granted is greater than zero.

- During *Phase C* the pub resource is not available any more and the Selection Policy dictates that a Simple Purchase topology is to be used. As the price paid to perform the execution to agent SGA3 is the same that the Client paid to SGA1 the endowment of SGA1 remains constant while the endowment of SGA3 increases.

This experiment showed how SGAs can be part of different topologies characterized by different social rules and that they behave accordingly to policies that define execution modalities, social and economic parameters.

Agent SGA1 is part of three different topologies: *Direct Control*, a socially inspired *Pub* topology and a market-driven *Simple Purchase* topology.

It is possible to analyze the advantage of belonging to different topologies (*societies*) by considering what would have happened had SGA1 belonged to only one topology

- *Direct Control*. Had SGA1 belonged to just a *Direct Control* topology and thus able to submit jobs only to the resources it directly controls it would have had to either relax its execution policies either allowing the complete use of its resources and/or the queueing of more jobs (accepting longer submission times) or it would have had to renounce the execution of some of its jobs.

- *Pub*: Had SGA1 been part of only a Pub topology it would have been able to submit jobs only if granted enough tokens by SGA2. This case happens when the workloads of the different agents encompassed in the pub topology are out of sync so that the pub credits accumulated during the peak of one of the workloads can be claimed during the other. Clearly, if the workloads happen at the same time the pub topology is not of help.

- *Simple Purchase*: Had SGA1 been part of only a Simple Purchase topology, then it would have been able to submit its jobs if and only if its endowment was sufficient.

By being able to encompass different topologies and thus expose different social behaviours SGA1 can take advantage of its "social relations" and its capability of coping with unexpected workloads grows with the richness of its social environment.

4.4.5 Second behaviour experiment

The testbed is similar to that of section 4.4.4 with the difference that there are two simple purchase relations. The latter, being more expensive, is used only as a last resort resource.

The abstract testbed for the experiment is illustrated in in Figure 4-37. The testbed is composed of eight agents of which four are Production Agents (*PGA1*, *PGA2*, *PGA3* and *PGA4*) and four are Social Agents (*SGA1*, *SGA2*, *SGA3* and *SGA4*).

Experiment Topologies

The agents are arranged in four different topologies:

- the *Simple Producer* relationship encompasses *SGA1* with *PGA1*, *SGA2* with *PGA2*, and *SGA3* with *PGA3*.
- the *Pub* relationship encompasses the four agents *SGA1*, *SGA2*, *PGA1* and *PGA2*.
- the *Simple Purchase* encompasses three couples of agents: *Client* with *SGA1*, *SGA1* with *SGA3*, and *SGA1* with *SGA4*

Agent Policies

Social Grid Agent 1

Agent *SGA1* chooses between its resources (*PGA1* which it directly controls, *PGA2* which can be reached through a Pub agreement and *PGA3* which can be purchased) as dictated by the following policies.

- *Price Policy*: states that the resource can be used by anybody provided that the offered price is sufficient (10 credits per submission).
- *Execution Policy* enforced on Production Agent *PGA1*: this policy states that the resource can be used on the condition that at least one job slot is left available.
- *Selection Policy* enforced on Social Agent *SGA1*: this policy states that the first resource to be used is the one that is directly controlled (*PGA1*), then the one that is accessible through pub relations, then it can be purchased at the same price for which it was bought and finally, as a last resort, it can be purchased from the more expensive resource but on the condition that the endowment be above a certain threshold.

Production Grid Agent 2

Agent *PGA2* accepts or not jobs from *SGA1* based on the following policies:

- *Execution Policy*: states that the resource can be used on the condition that at least one job slot is left available.
- *Authorization Policy*: states that the resource can be used by *SGA1* only if the *'Pub Credits'* are greater than zero. Each time it performs a job on behalf of *SGA1* the amount of its tokens is decreased by a unit.

Social Grid Agent 3

Agent *PGA2* accepts or not jobs from *SGA1* based on the following policies:

- *Price Policy*: states that the resource can be used by anybody provided that the offered price is sufficient (10 credits per submission).

Social Grid Agent 4

Agent *SGA4* accepts or not jobs from *SGA1* based on the following policies:

- *Price Policy*: states that the resource can be used by anybody provided that the offered price is sufficient (12 credits per submission).

Experiment Initial Conditions

The experiment was performed using the *TestGrid* infrastructure [31]. The initial condition of the testbed were:

- *SGA1 Endowment* was set to 100 credits.
- *SGA3 Endowment* was set to 0 credits.
- *SGA4 Endowment* was set to 0 credits.
- *PGA2 Tokens*[3] was set to 5 units.

This set of initial condition was chosen to explore all the possible behaviours of the agents. The Direct Control execution policy was set to always leave a free slot so that on average it would not suffice, Pub tokens were set to a low number (and not zero) so that they could be used but not suffice for heavy workload, leaving the two Simple Purchase relations as the last viable solutions. These settings are similar to those of 4.4.4 with two differences: firstly the workload is incremented so that the topology of 4.4.4 which is a subset of this one, is not sufficient. Secondly a fourth agent (SGA4) with resources that are kept ready is connected through a *Simple Purchase* relation that offers them at a higher price (12 credits per submission). This is to allow SGA1 to sort the purchase with regard of cost, thus using the services of SGA4 as a last resort.

[3]Tokens are introduced in 3.2.2, they represent the number of job submissions that an agent is granted in a Pub topology

Experiment Results

The results of the experiment are recapitulated in Figure 4-38 that shows the four sets of metrics that were collected during the experiment and in Figure 4-39, a graphical summary of the behaviour of the entire topology during the experiment. There are three main phases of the experiment: Appendix E contains Figures E-1, E-2, E-3, E-4 and E-5 that describe in greater detail the various metrics gathered during the experiment.

- The *workload*[4] is represented by the graph on the left of the image.
- The *endowment of agent SGA1* is represented by the graph that is immediately right to the workload.
- The *endowment of agent SGA3* is represented by the graph on the top of the image.
- The *endowment of agent SGA4* is represented by the graph on the bottom-right of the image.
- The *number of tokens granted to agent SGA1 by agent PGA2* is represented by the graph on the bottom-left of the image.

Appendix E contains Figures E-1, E-2, E-3 and E-5 that describe in greater detail the various metrics gathered during the experiment.

Figure 4-39 is a graphical summary of the behaviour of the entire topology during the experiment. There are three main phases of the experiment:

- In *phase A* the workload of agent SGA1 can be met entirely by the resource (Production Agent PGA1) it is in control of. As specified by its *Selection Policy* all jobs are then executed by this resource until one job slot remains free.
- During *phase B* the directly controlled resource PGA1 does not match any more the execution policy of having one free job slot. The Selection Policy dictates

[4]The workload represents the overall amount of job submissions that the agent copes with

that, in this case, the Production Agent PGA2 that can be reached through a Pub topology has to be used. This behaviour is sustainable until the number of tokens granted is greater than zero.

- During *Phase C* the pub resource is not available any more and the Selection Policy dictates that a Simple Purchase topology is to be used. As the price paid to perform the execution to agent SGA3 is the same that the Client paid to SGA1 the endowment of SGA1 remains constant while the endowment of SGA3 increases.

- During *Phase D* the cheap simple purchase resource is not available and the Selection Policy dictates that a last resort Simple Purchase topology is to be used provided that the endowment is above a certain threshold (that, in this case, is the initial endowment - 100 credits).

The results of this second behaviour experiment show how SGA1 is able to sort its economic-driven relations with regard to their cost and to make decisions accordingly. In this case the decision criteria was to use cheaper resources first and to use the more expensive only as a last resort and only if the endowment is above a certain threshold. The set of policies of the agent allowed for the definition of an opportunistic behaviour where the endowment was to be increased or, at least, kept constant and where the agent preferred to use it's own resources, those accessible through Pub relationships and only as a last resort those accessible through a market. Obviously different policies in the management (Execution Policies) of its own resources and in the ordering of the resource reachable through social topologies result in different behaviours. An agent may be instructed to use Simple Purchase resources (provided that its endowment allows it) first and leave it's own free as a last resort or rely firstly on Pub topologies if the timing of the workloads allows for load balancing with this solution. The main result of this experiment is that SGAs can provide this flexibility.

4.4.6 Conclusion

The reliability experiments (detailed in 4.4.1) showed that Social Grid Agents pose an acceptable overhead to access resources and do not show particular problems of reliance. This implies that border regions of the metagrid will be stable and reliable and that offering to users access of resources through an SGA topology will not result in a perceivable degradation of the quality of service in term of submission times and successful execution percentages.

The scalability experiments (detailed in 4.4.2, 4.4.2 and 4.4.2) show that the performance of Social Grid Agents remains acceptable with numbers in the range of hundreds but rapidly increases with greater numbers. This behaviour is mainly due to the technology that hosts the agents (the GT4 container). Two other sets of experiments show that the agents allow to execute in a synthetic environment (and thus without the overhead imposed by the GT4 container) can scale in their thousands with an acceptable (although significant) overhead.

The behaviour experiments showed how the agents can implement different, co-existing allocation philosophies and that these behaviours were those specified by their behaviour policies. The behaviour experiments were performed on a complex topology where three different allocation modalities were present: one where the resource where directly controlled, one where two parties shared part of their resources and an economic-based relationship where monetary credits where asked in exchange of the execution of jobs. The experiments highlighted that the main feature for which SGAs were developed (to implement an allocation mechanism capable of encompassing different allocation philosophies) was successfully achieved but they also highlighted limits in the number of concurrent transactions that agents, with the current technology, are capable of executing.

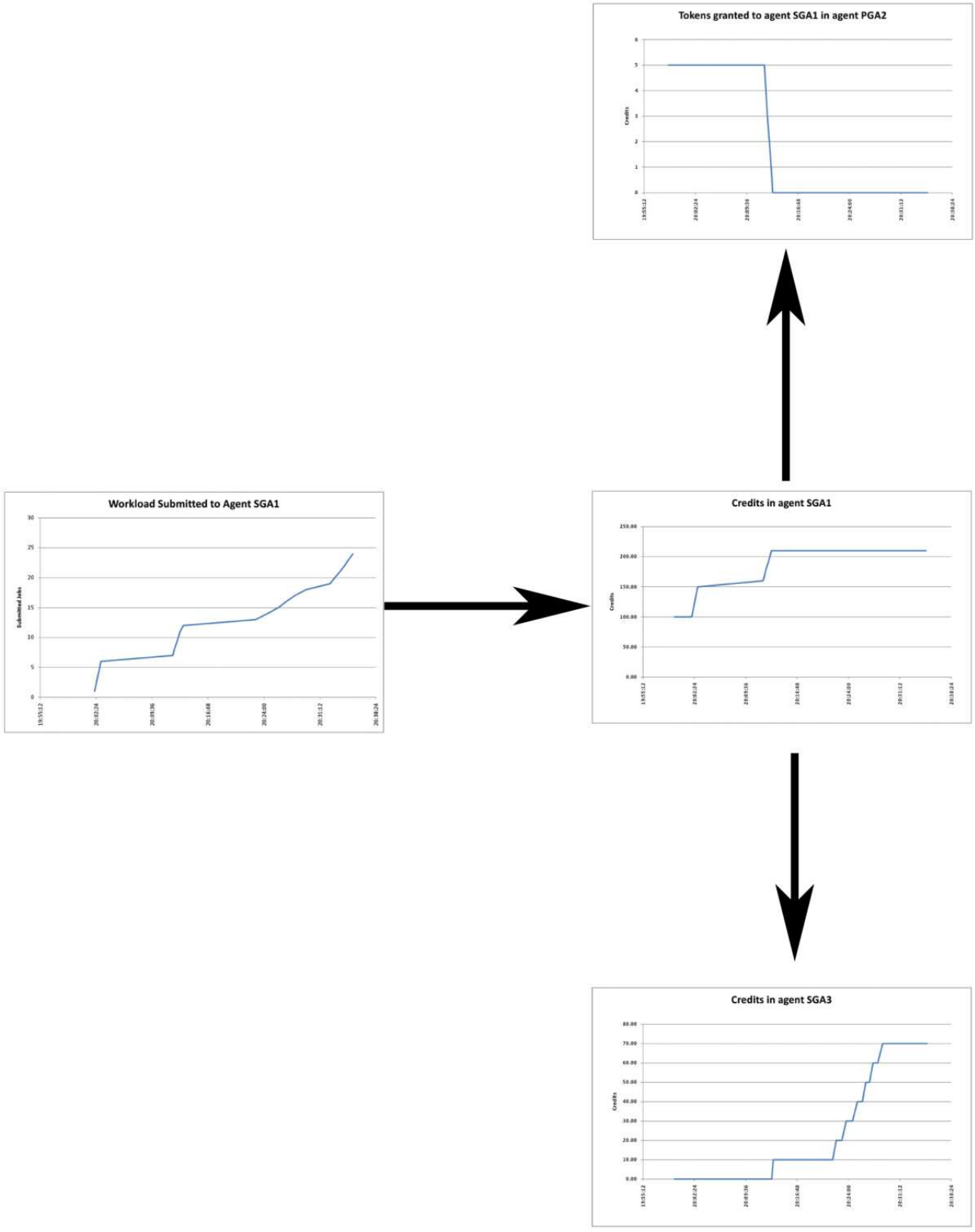


Figure 4-35: Relations among the different metrics of the first behaviour experiment.

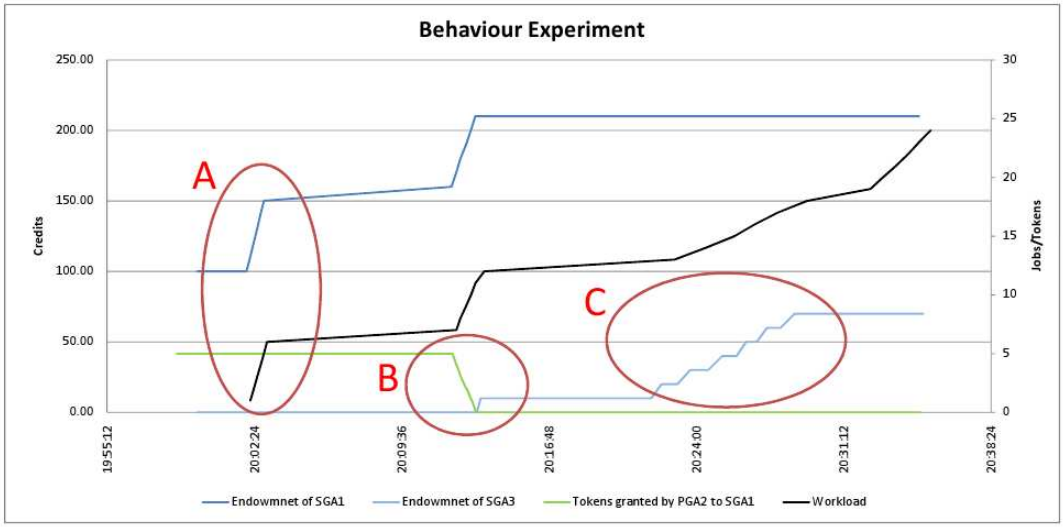


Figure 4-36: Main metrics of the first behaviour experiment.

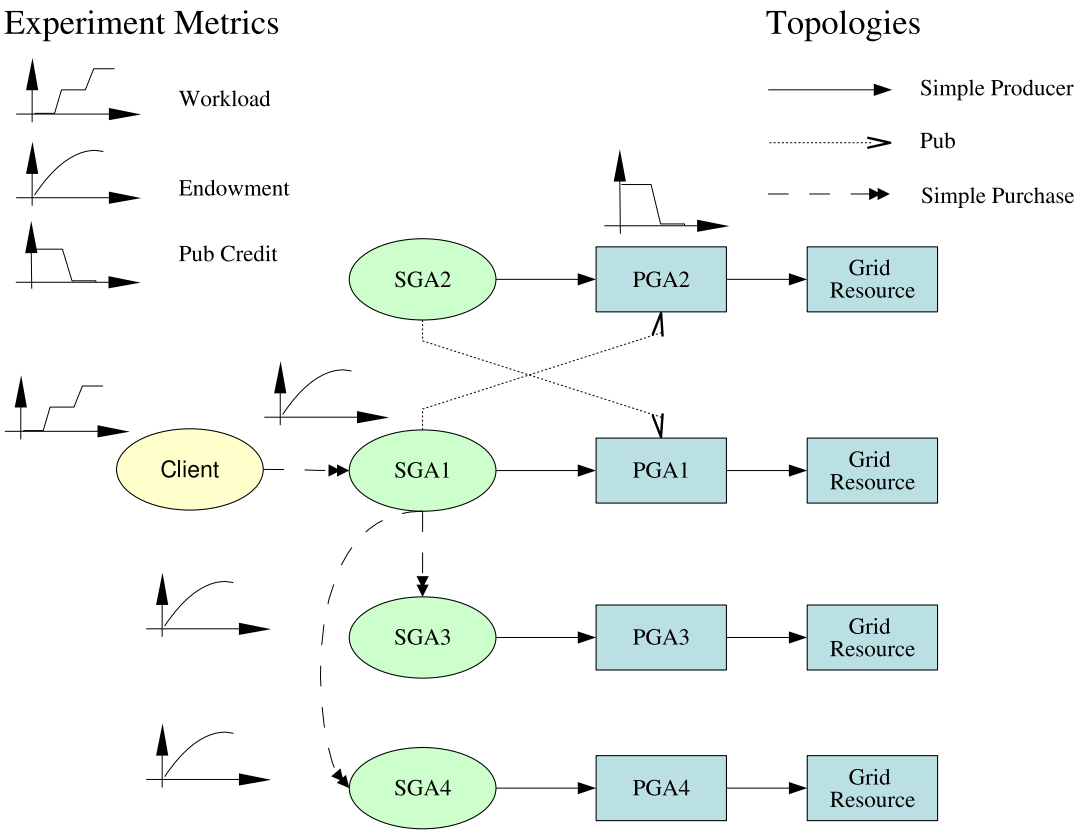


Figure 4-37: Testbed for the second behaviour experiment.

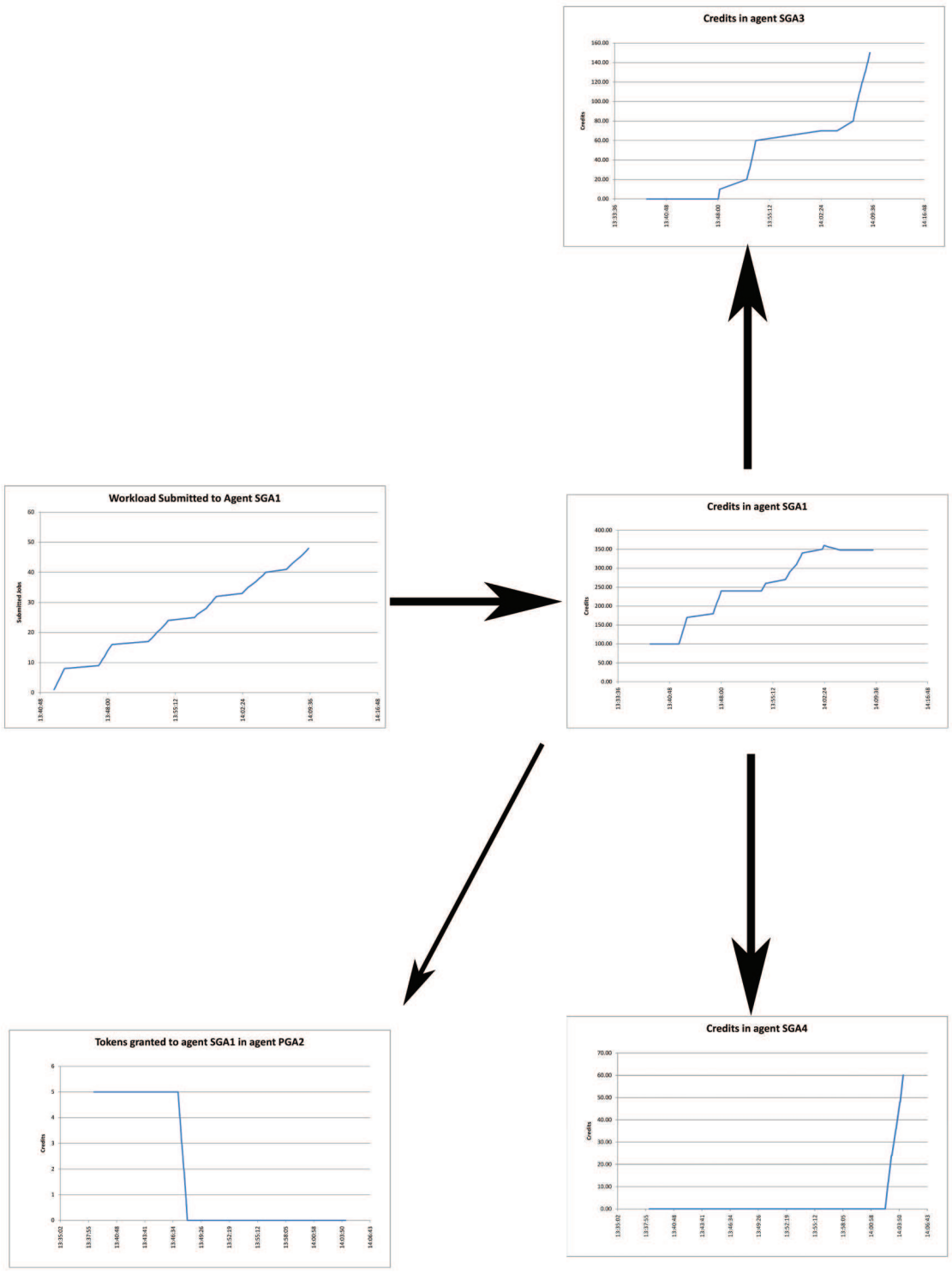


Figure 4-38: Relations among the different metrics of the second experiment.

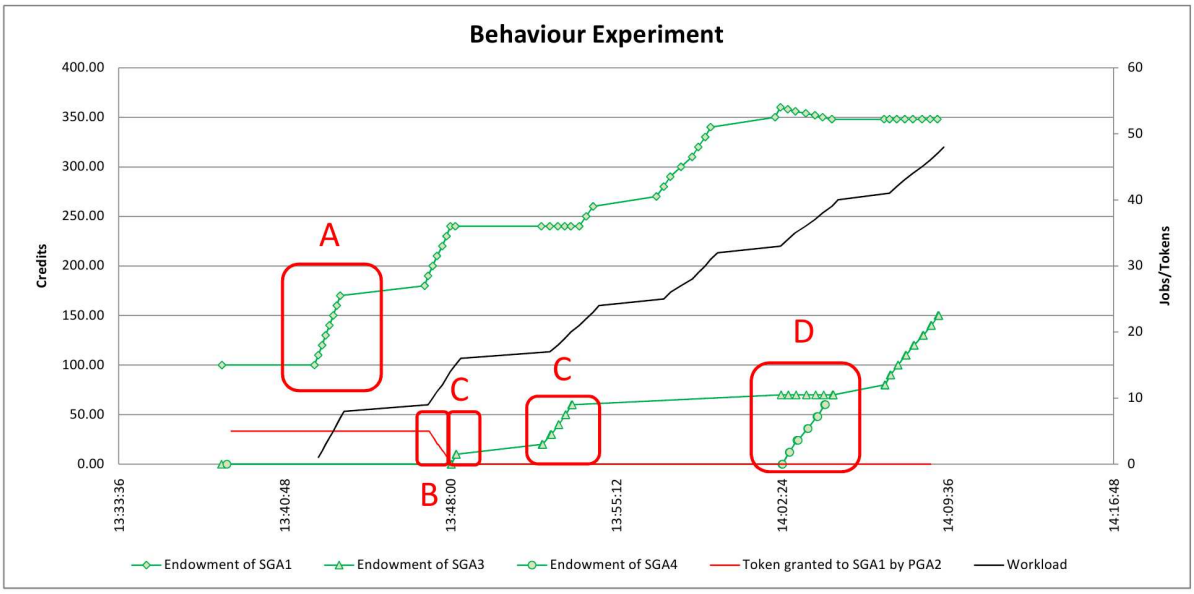


Figure 4-39: Main metrics of the behaviour experiment.

Chapter 5

Conclusions

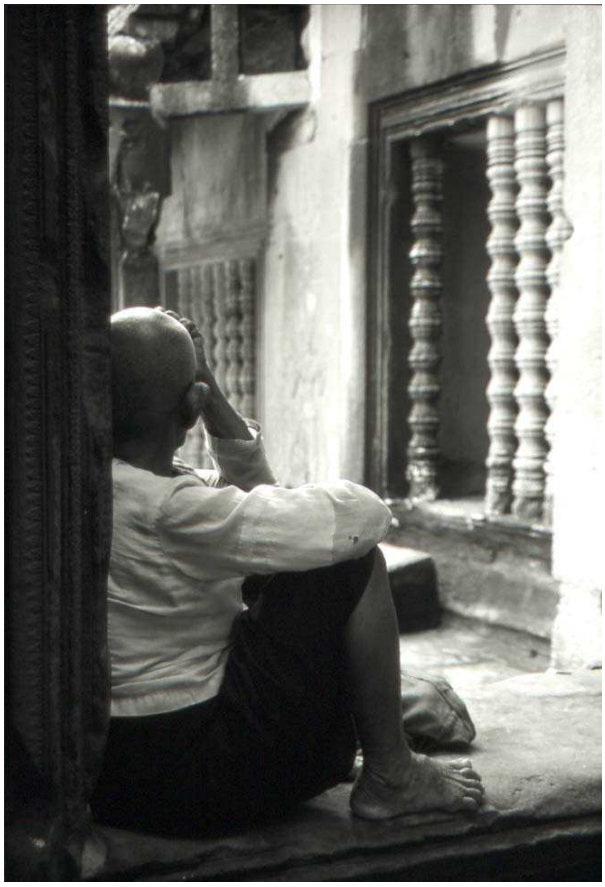


Figure 5-1: It's time to take a look back.

The first thing God created was the journey; then came doubt and nostalgia.

Theodoros Angelopoulos

All endings are also beginnings. We just don't know it at the time.

Mitch Albom

In the beginning of this journey we set the aim of this research at the creation of a set of software agents capable of enforcing different co-existing sharing and allocation methodologies in Grid computing. We also decided to use a social paradigm in the hope that mimicking behaviours that were useful in human history to tackle the problems of resource sharing and allocation would also be helpful in Grids. It is now time to try to re-examine the results obtained during this journey, the successes and the shortcomings.

5.1 Evaluation versus core criteria

The core criteria that the agents must be capable of encompassing different co-existing allocation philosophies has been proved by behavioural experiments that show how different policies of the agents result in different allocation philosophies. The fact that these policies can be expressed in terms of metrics, values and prices allows the definition of behaviours from economic to social. The latent assertion that this is better can be seen to be true when one recognizes that Grid is becoming a means to distribute computational power in an increasing number of scenarios and that some of the challenges posed by this expansion (coexistence of actors with different aims) can clearly be tackled with the SGA paradigm, although it is equally clear that this is a work in progress.

5.2 Evaluation of the technologies used

The last prototype of Social Grid Agents is implemented using the Java™language, the ClassAd language and the GT4 container as a hosting environment. This bundle of technologies had major advantages and one important shortcoming.

- The *GT4 container* allowed the implementation of the agents as *WRSF Compliant Grid Services*; in addition to this, the GT4 container provided a hosting environment that has been specifically designed for Grid Computing. Finally, the GT4 container directly hosts grid services, such as the GRAM submission system, that are used by the Border Agents.

- The *Java^{TM}language* allowed the implementation of the prototype with all the advantages and features of an *Object Oriented* language and offered the possibility of using APIs whenever they were available. Unfortunately, the different releases and versions of libraries gave rise to a frustrating problem nicknamed *Jar Wars* to which no definitive solution has been found yet. Section 5.2.1 is devoted to this topic.

- The *ClassAd language* allowed the definition of policies and their enforcement across different middlewares and across the different layers of the Social Grid Agents architecture. To express its full potential, the Java^{TM}API to the ClassAd language has been enriched by specific functions that allow for partial evaluation of a ClassAd expression as detailed in section 3.18.3. With the possibility to selectively evaluate only the part of each expression that is defined by the current scope the ClassAd language has proven to be a valuable and effective tool for the expression of Actions, Objects and Constraints. On the other hand, ClassAd expressions proved to be all but terse with consequent difficulties in reading and debugging the code. This shortcomings may well justify further research for an alternative functional-style language.

5.2.1 Jar Wars

Someone once said that the *"Advantage of sitting on a border is that you can be shot from two sides"*; this facetious statement describes well a *Jar War*: a conflict that arises whenever two application that have to interact are built with incompatible sets of libraries (that, in Java^{TM}, are implemented in archive files known as *jar files*).

As described in Sections 3.10 and 3.12 agents may depend on one, two or even

more technologies; whenever any of these technologies depend on incompatible sets of jar files there is a *Jar War*. There are different ways to cope with these conflicts but none of them has proven yet to be completely satisfying:

- *Class harmonization*: if two jar files have a conflict but they have a high degree of compatibility it is sometimes possible to build a third jar file from the subset of classes that are compatible to both. This was the first solution that was tried to solve a conflict between the GT4 container and WebCom libraries.

- *Jar switching*: consists in switching the *System Context* of the Java Virtual Machine (*JVM*) before and after each invocation of the API calls that cause the conflict.

- *System invocation decoupling*: consists in decoupling the two applications that are in conflict and letting them interact through *system calls* instead than API invocations. Jar files are then switched, controlling *environment variables*. This solution has the major disadvantage of disrupting the interoperability design based on API invocation and forcing all information to be expressed in *strings* as this data type is the only one directly supported by a system call.

The current prototype uses a combination of the above-mentioned solutions with a reasonable degree of efficiency but a unified and more formal solution based on the *jar switching* technique would certainly be greatly advantageous and justify the effort of some further investigation.

5.3 Evaluation of the architecture

The architecture of Social Grid Agents was designed to allow the implementation of different features: the expression and enforcement of policies in a flexible way, the possibility to interface with different middlewares, the possibility for the agents to expose a certain degree of "awareness" of themselves and their surroundings. We will now evaluate how well the implemented architecture served these purposes and complied to the design constraints and specifications.

5.3.1 Flexible policies

The description of policies through ClassAd expressions that are partially evaluated through the different steps proved to be suitably flexible. It allowed the description of policies that may depend on the identities of the sender and/or the beneficiary of an action, the current status of resources, and other policies. The description of these constraints as modalities defined by the requester and policies defined by the provider, that combine in enforced policies, lets all parties involved describe their needs and preferences.

The architecture also allowed the expansion of basic topologies along three additional dimensions (trust, banking and discovery).

5.3.2 Support for multiple middlewares

The example *metagrid* environment, based on common language and translation services, allowed Social Grid Agents to interoperate successfully with services offered by different middlewares. Furthermore the ClassAd based description of messages, actions, object and constraints allows for the definition of inter-dependent sets of actions that can model complex interdependent job flows that span multiple middlewares. An important challenge posed by the simultaneous harnessing of different middlewares resides in their different security requirements and policies. Being able to harness services from different middlewares can be useless if the requirements demanded by those cannot be met by the agents. This has been achieved with a reasonable degree of success, allowing the agents to implement different security modalities (specified by policies) and by describing the agent's identity with structures that host heterogeneous identity-related information.

5.3.3 Awareness of self and surroundings

One last feature that was required by the agents was the capability of being *"aware"* of themselves and their surrounding both in space and time. This has been achieved by a fundamentally associative design. Information regarding the agent itself and

its environment is memorized in maps that can be queried through keys. Keys are expressed in ClassAd and are searched with the *Requirements* function. Information covers time through logs. This system proved successful in memorizing the information needed by the agents. No thorough assessment on the scalability of this solution has being performed yet. At present information is not categorized with respect of relevance or age and it is all treated uniformly. This approach will not be able to scale both in time and space as logs and indexes will grow to the point of being un-manageable and too slow to be queried. A hierarchical, cache-like approach that handles information with respect of age and relevance might be worth investigation. Additional information could be easily added as additional fields in the ClassAd description of the keys and in the search criteria defined in the Requirements function.

5.4 Evaluation of the Prototype

To properly evaluate the prototype and its results it is necessary to analyze the results of the experiments from the perspective of the research questions that were stated at the beginning of this investigation, i.e. as stated in 4.4:

- Do Social Grid Agents degrade the performance of resources they control ?
- How scalable are Social Grid Agents ?
- How do they behave ?

5.4.1 Do SGAs degrade the performance of the resources they control ?

Two main considerations apply to the question of whether SGAs degrade or not the resources they control: one based on the architecture of SGAs and another based on experimental data.

From an architectural perspective SGAs are *users* of resources and they do not interfere in their internal working; in fact, production agents connect to resources

through their user interfaces (e.g. the UI for gLite and the GRAM service for GT4). SGAs thus *cannot degrade the performance of a resource* more than any other user.

On the other hand, the experiments in sections 4.4.1 and 4.4.2 show *degradation in the way resources are accessed*. This is mainly due to the overhead of multiple threads that are spawned, and the GT4 operations that are performed, for each received message.

5.4.2 How scalable are SGAs ?

This topic is closely related to that of section 5.4.1. Experiment of section 4.4.2 showed how the response time of SGAs degrades severely with the growing number of concurrent accesses. The influences on scalability of SGAs have been assessed by the experiments of sections 4.4.2 and 4.4.2. These results imply that the main scalability problem resides in the interactions between the GT4 container and the thread management of SGAs that severely slows down the access to the services of the agents, rather than the scalability of the agents themselves. In fact, the experiments on the scalability of the agent's brains (see section 4.4.2) show how the architecture of the agents (without the GT4 container) has better scalability characteristics that allows to manage up to a few hundred concurrent messages.

5.4.3 How do SGAs behave ?

One important aim of this investigation was the possibility of the definition of different allocation philosophies within the same framework. In this regard the Social Grid Agents model proved to be capable of defining diverse allocation philosophies ranging from the altruistic social behaviour of the pub model to the selfish market-driven purchase mechanism.

Chapter 4.4 describes in detail a behaviour experiment where three allocation topologies co-existed. The experiments detailed in 4.4.3 shows how an agent that belongs to different topologies can be instructed to rank and choose services and resources availing itself of relations that span from direct control to service purchase, from a social-based model to an economic one.

The behaviour experiments showed that SGAs behaved as would be expected from their policies: they managed the resources they where in direct control of and those that could be reached through social topologies with respect of their internal policies. The allocation of resources that were directly controlled by Production Agents was successfully tuned by execution policies that described how much of each resource could be used and if queueing was to be allowed. The economic endowment (e.g. the number of credits) and the *social endowment* of the agents (e.g. the number of tokens granted in pub topologies) were successfully described and enforced. This allowed the overall behaviour of the agents to be selfish (thus trying to maximize or keep constant their endowment) or not.

It is worth noticing that all this diversity of behaviour was obtained by orchestrating basic tiles (the service providers) and by defining policies, modalities and actions, thus proving how SGAs can be easily configured and assembled in complex topologies.

Although the agents behaved according to their policies there are weaknesses that were not shown in the experiments. The most important of all is due to the latency times with which the resources publish their status. In the experiments the average length of the jobs was enough to allow the resources to publish their correct status but shorter jobs may result in the agents taking decisions based on out-of date data. This latency problem is common to all distributed systems. SGAs cope with it by keeping and updating a local copy of the information regarding the status of jobs. This solution allows a more accurate behaviour but bears the disadvantage of the increasing workload of the job controller thread and its communication to perform polling.

5.4.4 Other questions about the prototype

- *Is it simple ?* Although SGAs are conceptually simple, the prototype implementation is far from that. Policies definition and description is hard to implement and the resulting ClassAd code is often long and very hard to understand and debug.

- *Did a single-stub approach simplify outsourcing ?* Yes it did. This design decision to have SGAs expose a single method allowed for the composition of services without the additional complexities brought by the need to discover the characteristics of an interface each time a new agent is contacted. On the other hand, this approach makes it much more difficult for SGAs to interface with systems such as OGSA that rely on richer interfaces.

- *What are the least satisfactory features of the prototype ?* There a number of issues:

  - *How agents get to exist*: at present this information is statically defined in the GT4 container.

  - *How initial relations are established*: at present they are statically defined and there is no mechanism for new relationships to begin.

  - *How topologies are composed*: at present they are statically defined and, although the policies of a single topology may change, there is no mechanism for creation of new topologies.

  - *How namespaces are established*: at present namespaces are managed with ClassAd scopes, but this proves to be often unreadable and hard to maintain.

5.4.5 How do SGAs fit into the surveyed Taxonomies ?

An interesting question is where SGAs fit into the Taxonomy of Resource Allocation Systems in [52]. On one hand, the architecture can be described as hierarchical as it defines two distinct layers (the social and production layers) with a clear hierarchy - the social layer controls the production layer; on the other hand, it is also true that the architecture is cell-based as the relationships and information between the agents can be configured to define domains across the hierarchical structure.

The schema is extensible provided that all agents have an updated version of the processors. Soft QoS support is implemented with the use of modalities, policies and enforced policies.

The information is stored by each agent locally in files and there is no database support; the information is updated with a pull policy each time an agents requests information to another, although push policies may govern the interactions with indexes. Finally, scheduling is provided by external services through existing middleware.

With reference to the economic models surveyed in [28] and [29], SGAs offer support for *Commodity Markets* and *Posted Price Models* in addition to *Charitable donations*, non-monetary based *Pub Models* and *Tribe Models*, and third-party funding with *Keynesian Models*.

It is worth noticing that by providing the agents with the proper processors it should (at least at a theoretical level) be possible to allow SGAs to behave according to other economic models.

5.4.6 Relation with related work

This section describes what are the relations, the commonalities and the differences with the related work surveyed in Chapter 2.

The gang-matching system proposed in [81] and described in Section 2.2.1 extends the match-making mechanism to more than two parties; although no explicit match-making mechanism is implemented yet in Social Grid Agents there are strong commonalities. Firstly, they both use the ClassAd language to convey information and base their decision mechanisms on the match-making mechanism, secondly the delayed evaluation of policies of Social Grid Agents allows for a form of implicit multi-step gang-matching as all the different actors that manipulate the ClassAd message participate in the final decision.

The solutions proposed by Buyya in his GridBus project described in Section 2.2.2 are related to Social Grid Agents in the sense that they support different economic mechanisms. The main difference with Social Grid Agents is that Buyya's approach is strongly economical and does not extend to social behaviours that cannot be modelled in economic terms while Social Grid Agents aim at encompassing different economic models within a larger view as just one of the possible allocation philosophies that

an actor can follow. GridBus, on the other, hand is in a much more advanced implementation state and is already a "ready to use" solution while Social Grid Agents are still in a prototypical stage.

Finally, Social Grid Agents relate to gLite (described in 2.2.1) and Globus (2.2.1) allocation mechanisms by using their allocation mechanisms for the resource allocation within their domains; border production agents, in fact, invoke the resource allocators of gLite and Globus in the border regions between these two middlewares and the metagrid region.

For what regards the description and enforcement of policies, the paper [95] offers a similar view of the relationships between collaboration models (very similar to the topologies of Social Grid Agents), behaviour and policies. The paper offers both a conceptual model and a formal specification for a policy language.

5.5 Contributions

Social Grid Agents where designed to be *economic-agnostic* and *technology-agnostic* so to encompass both different allocation philosophies (defined by their social context) and different Grid middlewares.

The capability of encompassing different, co-existing middlewares was tackled with the concept of a *metagrid* that allowed the creation of a flexible system harnessing different Grids. The contribution of the *metagrid* concept is beneficial as it allows to include new middlewares localizing the issues of interoperability to a set of border agents that act as translators both from the definition of functionalities and policies, thus allowing interoperability both at a technological and logical level.

Social Grid Agents prove to be beneficial on the following counts:

They allow to model in the Grid a variety of social relations that exist among its different stakeholders; topologies can be set to allow co-operation and competition depending on the social context of the Grid actors.

They allow for different allocation philosophies to co-exist so that an actor can fully avail himself of all the social relations it has in a Grid without being constrained in just one topology.

They will allow in the future for Social Grid Agents to encompass types of middlewares that are based on different allocation philosophies such as commercial Clouds and volunteer-based computing such as BOINC [3], this interesting possibility is offered by their agnosticism with regard to economy and technology.

A feature that was not foreseen in the beginning, the functional characteristics of its language, proved beneficial as it allows the definition of functionalities and policies that take into account the needs of all the actors involved and define abstract patterns where functionalities, policies and behaviours of the agents interact without the need of designing a priori the entire decision process.

5.6 Future Work

Although the current implementation of Social Grid Agents yielded some useful results, its full potential may yet to be fully discovered. Future directions of research will move along different dimensions.

5.6.1 Deployment on a Production Infrastructure

If SGAs were to be deployed in real production infrastructures with their stricter quality standards, then the following issues must be successfully tackled.

- *Scalability*: the scalability issues referred to in 4.4.2, 4.4.1 and especially in 4.4.2 are the most important issues that prevent SGAs being deployed in a realistically large and complex production infrastructure. The degradation of the quality of service for numbers of concurrent accesses that are normal in a production infrastructure is unacceptable and, as long as this issue is not solved, SGAs will not be usefully deployed in real production infrastructures.

- *Startup*: at startup SGAs have a pre-defined set of topologies they belong to and this information is coded in the ClassAd policies that describe the agent. Obviously this solution lacks scalability and must be replaced by a *startup procedure* that uses default indexes to allow the agent to insert themselves in existing societies when they *awake* or when they are *born*.

- *User Interface*: the prototype of SGAs has a minimal text-based interface to the user. This is unacceptable if the agents are to be deployed in a real production infrastructure; SGAs need usable graphical user interfaces that allows user to instruct their agents on execution and behavioural policies and resource owners to describe the behaviour of the agents controlling the resources. These GUIs must be easy to use and must not require their users to have any knowledge of the ClassAd language. HyperGraphs [51, 53, 62] and related work on e-Learning conducted by Kathryn Cassidy of my host research group [30] could be interesting avenues in this regard.

- *Packaging*: SGAs would of course require packaging for use by the fabric management tools used for deployment of Grid software components.

5.6.2 Additional Services

To enhance the usefulness of SGAs more middlewares of different nature should be encompassed. Effort should especially be focused on middlewares based on different philosophies such as commercial-based platforms like the IBM cloud and volunteer-based platforms like BOINC.

In addition to this it would be profitable to encompass more services of the middlewares that are already party of the system; specifically file storage and information services which will provide both a more realistic and useful platform and a much richer and diverse environment in which to study the outcome of different social and economic models.

Early prototypes included a *File Closet*, but future implementations may simply include a SRM [45] interface so that they can employ file catalogs like LFC [44] and/or file system interfaces like STORM [76].

5.6.3 Negotiation Protocols and Service Level Agreements

In SGAs actions are described with the use of modalities and a rather primitive, one-step form of negotiation is offered by the intersection of policies and modalities

that will define how the action is really executed. Social Grid Agents will certainly be much more interesting if they had a more advanced approach yielding to formal Service Level Agreements and Negotiation Protocols. For the latter it could be feasible and profitable to encompass some existing technology and to delegate the act of negotiation to it.

5.6.4 Advanced decision systems

For now, SGAs take most decisions based on thresholds and other simple algorithms. It would be an interesting field of enquiry to develop more complex and realistic decision system based on the metrics of both the agents and the resources they are in charge of.

5.6.5 Large scale indexes

The lack of scalability that characterize the management of internal information of the agents is also present in the topologies of the agents (see Section 4.4.2). Although the scalability of the main component of Banking and indexing agents, the *ClassAd Mapper*, was the topic of the experiment in section 4.4.2, their overall behaviour has not being tested for scalability. Banking should be scalable and it could be wise to abandon the current, prototypical banking agent and interface the system directly with an existing banking technology (e.g. that of SGAS).

Indexing agents should also take into account larger amounts of more sophisticated data about the services they index, accepting only that information that could prove profitable. It could also be interesting to investigate multi-layered hierarchical indexing services.

5.6.6 Advanced Social, Economic and Financial Models

As yet, only relatively simple social topologies and their policies have been developed for the agents. In the future, research should embrace more complex topologies. At a micro-economic level, the existing solutions may be be enriched by *insurance* agents

that compensate with credits when middlewares fails; for trust reasons, these agents could be controlled by the *arbitrator* agent described in section 3.7.

Currently agents define prices based only on the data from the supply side without taking into account demand. This doesn't allow the definition of realistic price models and thus some mechanism capable of gathering demand-related information would prove very profitable for the implementation of more sophisticated agents.

A much more advanced topic that is still to be tackled is the relation between the credit system used by the agents and the real economy. For the present there is no direct connection between the credits that are used among the agents and real currencies. This is a serious limitation that prevents agent's societies from being connected to the real economy. Whether a connection between these two economies would yield positive or negative outcomes for one or both is a very open questions that, in my opinion, could constitute a proficuous field for future enquiries.

5.6.7 Trust

As of now, Social Grid Agents, support only one concept of trust (Discussed in Section 3.2.7 and Section 3.8.3) based on the authority of one agent that the other parties can accept or not. This is a very limited view of the concept of trust and a more articulated approach to this topic would be beneficial to Social Grid Agents. The two other views of trust, the one based on the recognition of behavioural patterns through time and the one based on the opinion of other other trusted parties, should be encompassed in Social Grid Agents in a relatively easy way. Social Grid Agents, in fact, keep records of events written in ClassAd records by their logging service and queries of patterns of events can be performed through matchmaking queries of these records; level of trust can than be inferred from this information. The exchange of "opinions" that the different agents have of each other can be achieved in two possible ways: an agent may share part of its logs to its parties and allow them to infer trust level from this information or they can directly their "opinion" on the trustworthiness of a party. This latter solutions implies that all the involved party use the same algorithm and the same metrics to infer the trustworthiness from the

logs.

On the other end, once trust becomes a reliable information metric on Social Grid Agents it could be taken into account in a large variety of policies and decisions. As an example, trustworthy parties can be alloted greater resources, they can be granted lower prices or they can be granted larger shared resources in a pub model or, finally, only parties whose trust level is above a certain threshold may be given access to resource controlled by a Keynesian authority.

5.6.8 Service Level Agreements

As of now, Social Grid Agents, allow for very limited implementation of Service Level Agreements. They consist only in the data that specify both the functionalities and policies for the execution of jobs. On the other end, the functional characteristics of the native agents language should allow more sophisticated implementation of Service Level Agreements as modalities requested by the agents a should allow a flexible way to define the behaviour of the agents accordingly.

As an example, a requested deadline requested for the completion for a job may not only influence the resource allocated but also the modality with which the status of the job is queried. Theoretically, the homogeneous nature of the agent's internal and external language allow for the creation of arbitrarily complex mappings between action, policies, modalities and behaviours; the difficulty of reading and debugging the ClassAd code may represent, on the other hand, a strong constrain to the level of complexity of these patterns.

5.7 Acknowledgements

Funding for this work was provided in the main by the SFI WebCom-G project, and latterly by the HEA PRTLI4 e-INIS project. My thanks to both organizations, to Trinity College Dublin and the Irish Government for this support.

5.8 Conclusions

The journey that was introduced in the foreword comes here to a stop. As all other journeys it yielded more questions than answers and it ended in a place that is not precisely the one decided in the beginning. Some parts proved much harsher and more difficult than foreseen while unexpected and welcome surprises compensated for some of the most frustrating parts of the path. Is this not, after all, the very nature of every journey ?

Appendix A

Detailed Metrics of Reliability Experiment

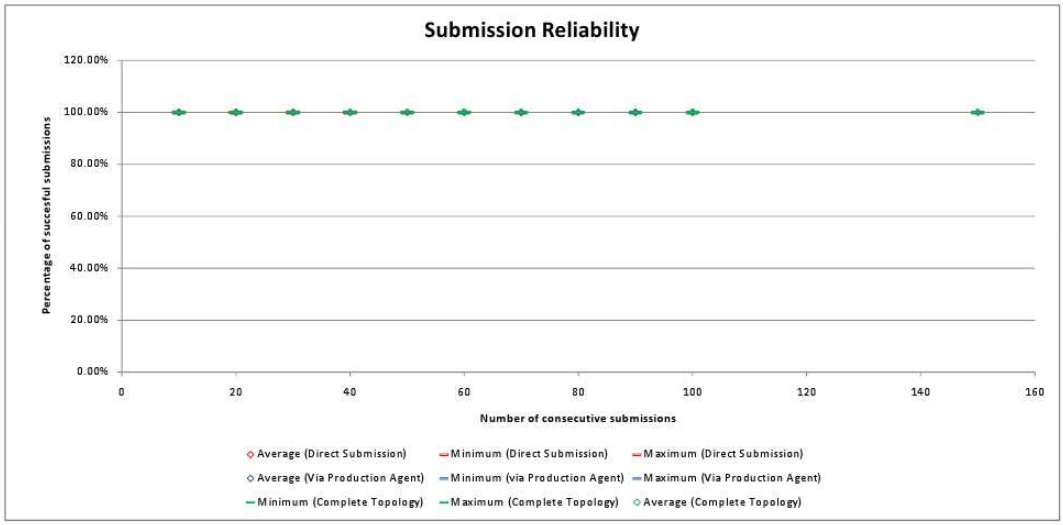


Figure A-1: Submission reliability of the gLite border.

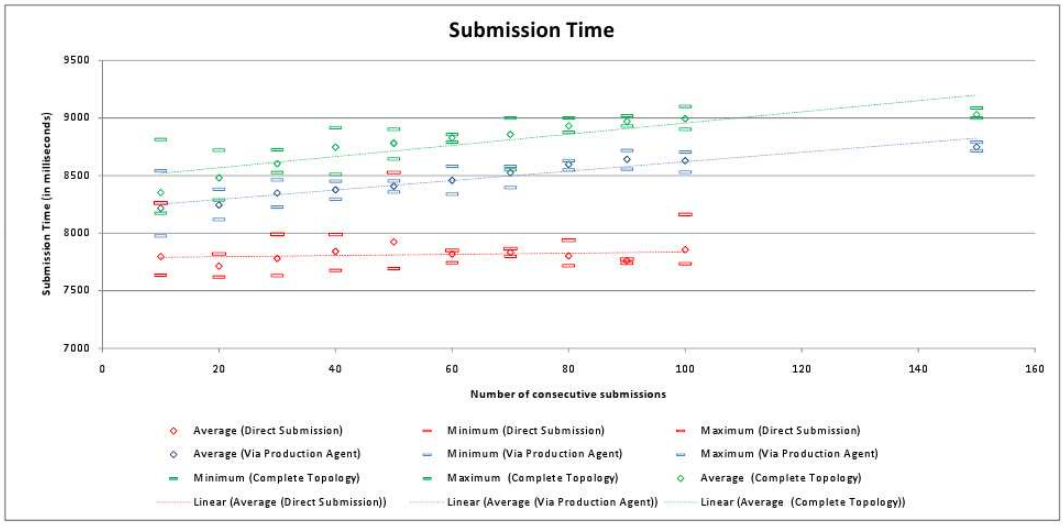


Figure A-2: Submission time of the gLite border.

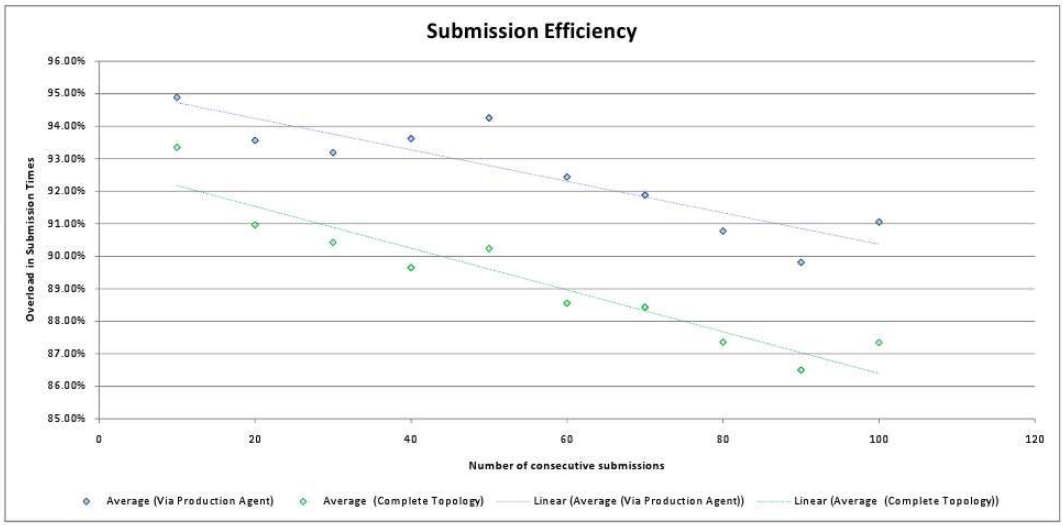


Figure A-3: Submission efficiency of the gLite border.

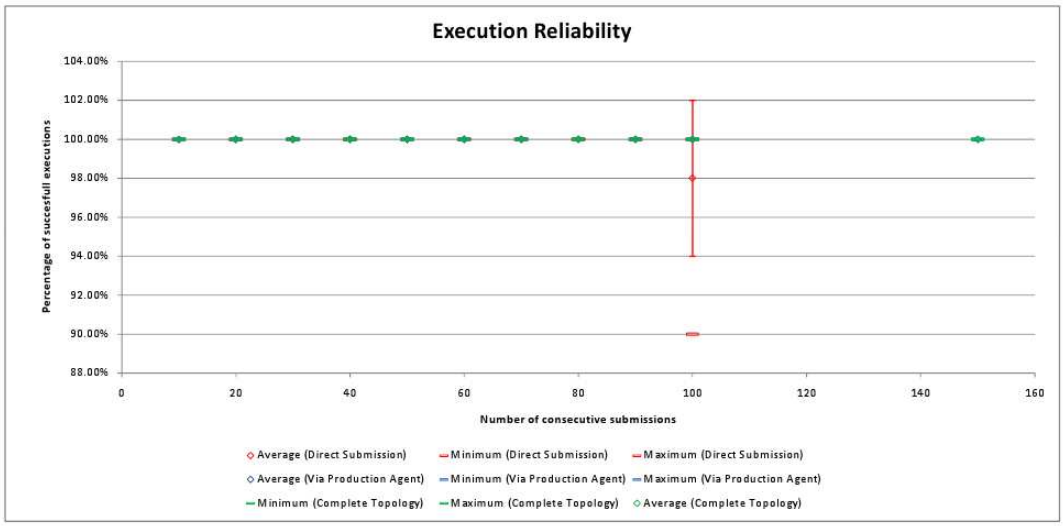


Figure A-4: Execution reliability of the gLite border.

Appendix B

Detailed Metrics of the Scalability Experiment on the Concrete Testbed

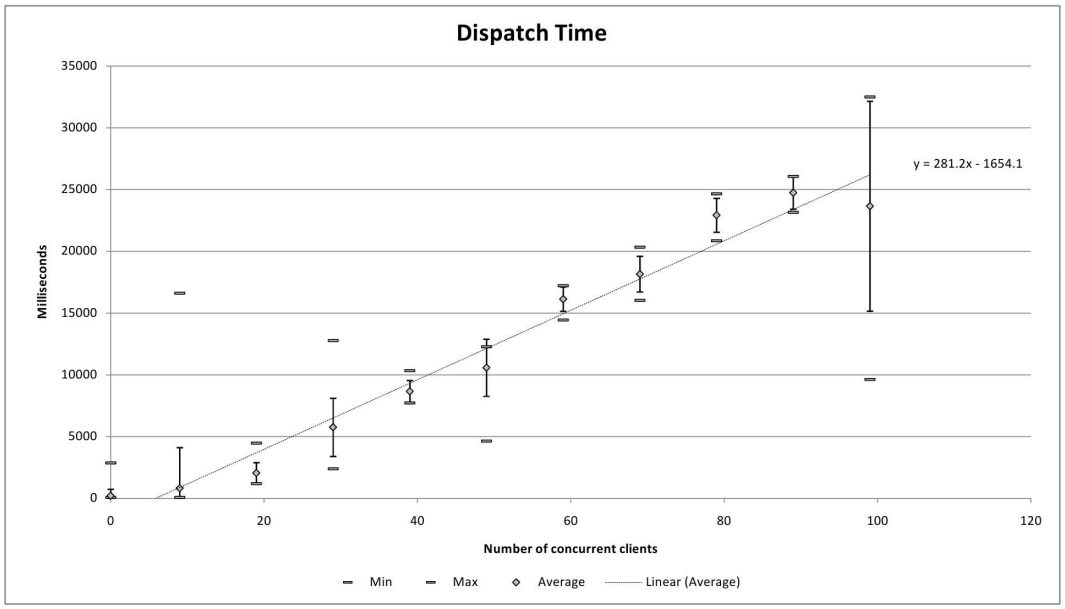


Figure B-1: Dispatch time of the scalability experiment.

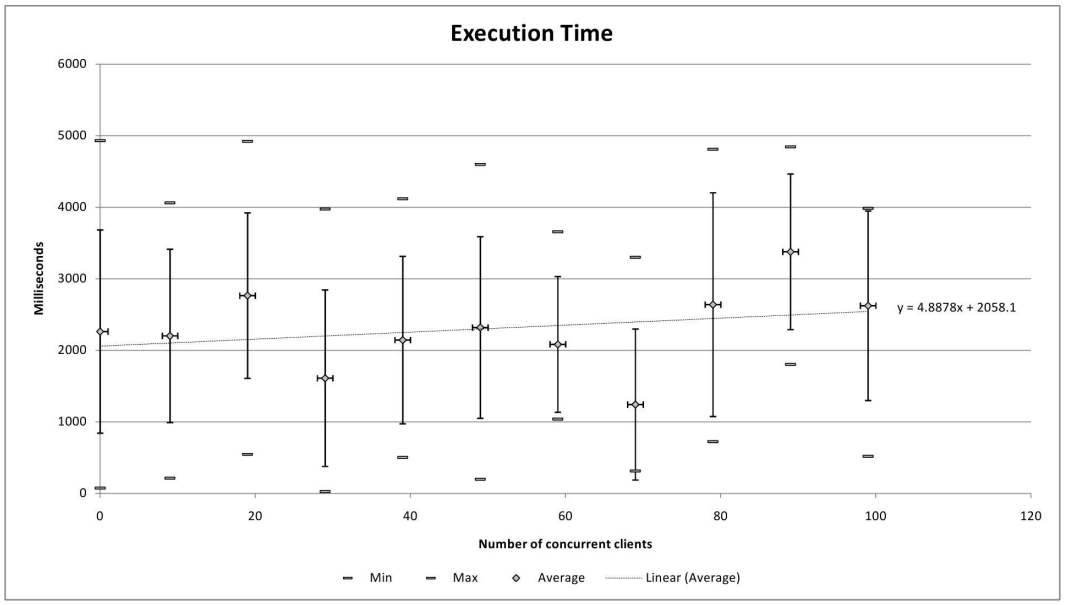


Figure B-2: Execution time of the scalability experiment.

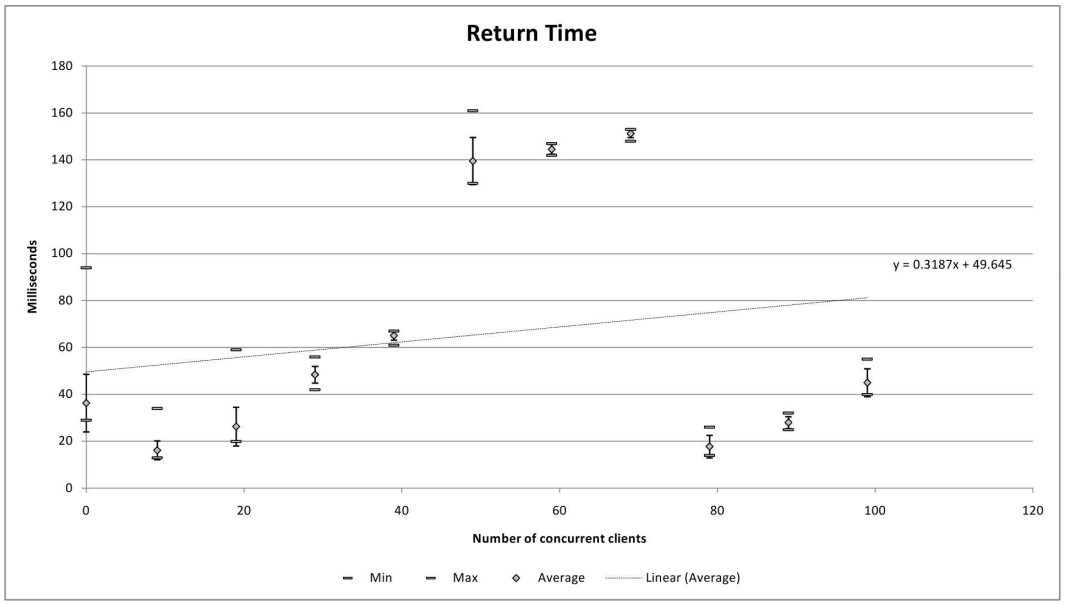


Figure B-3: Return time of the scalability experiment.

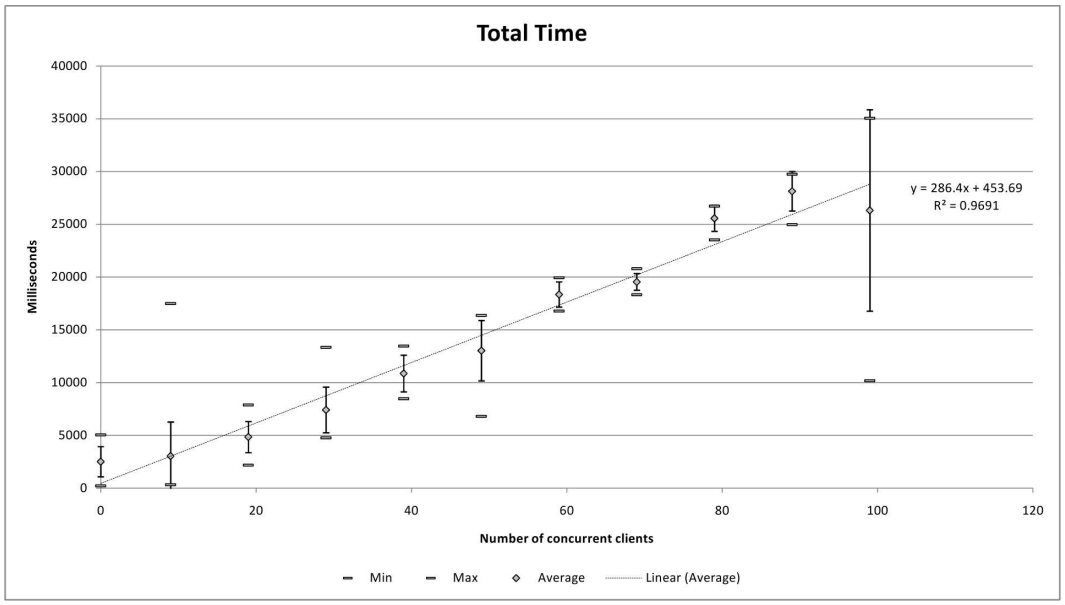


Figure B-4: Total time of the scalability experiment.

Appendix C

Detailed Metrics of the Scalability Experiment on the Synthetic Testbed

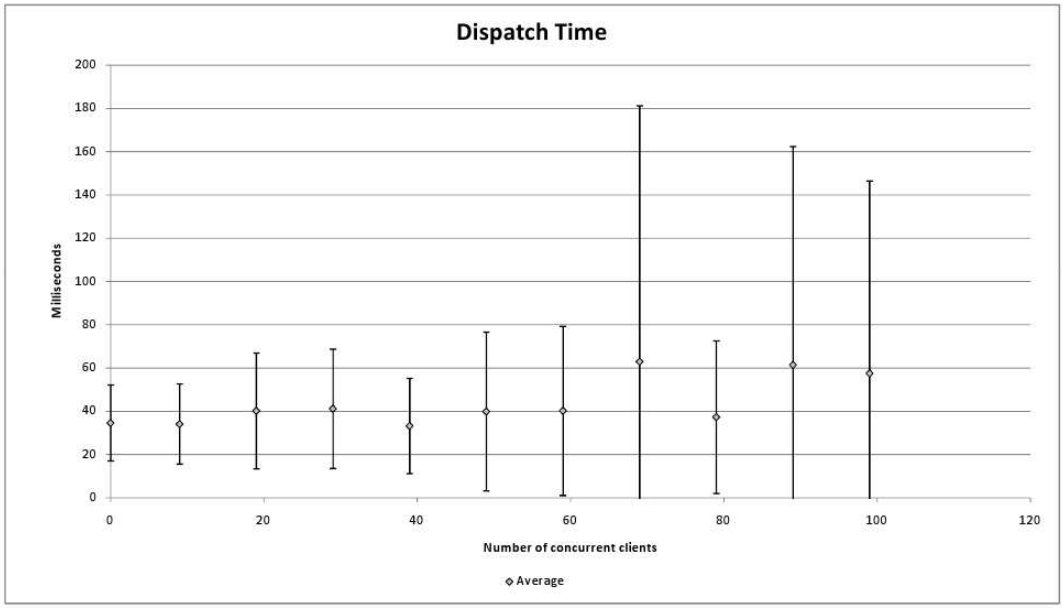


Figure C-1: Dispatch time of the small-scale scalability experiment on the synthetic testbed.

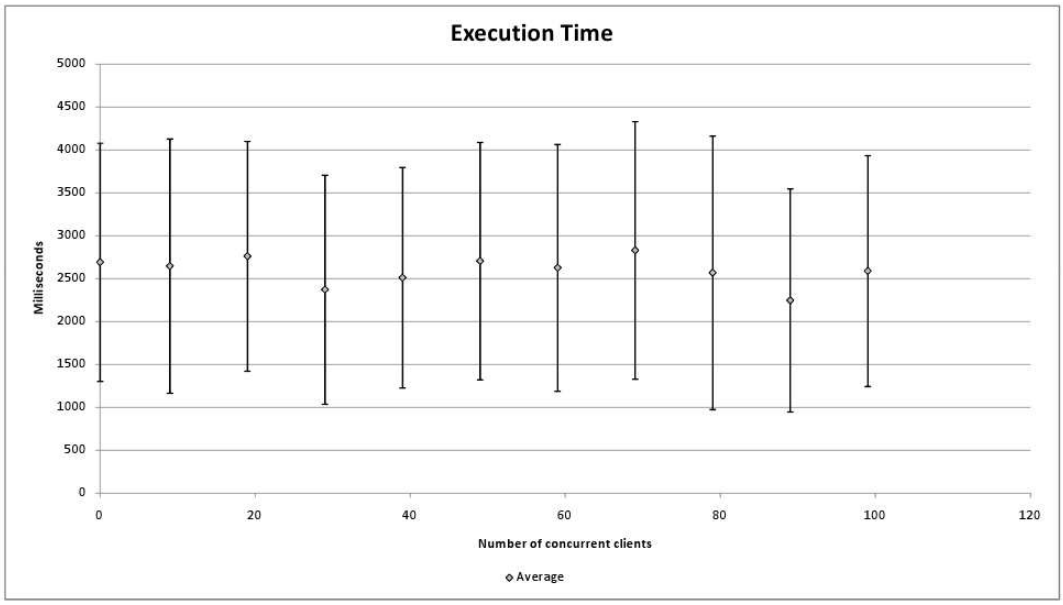


Figure C-2: Execution time of the small-scale scalability experiment on the synthetic

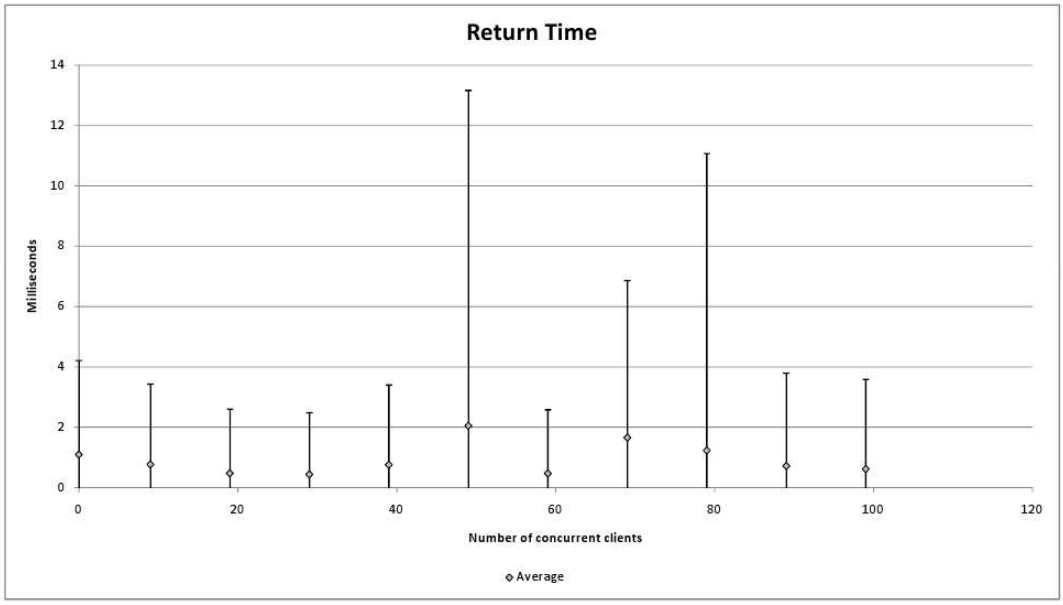


Figure C-3: Return time of the small-scale scalability experiment on the synthetic testbed.

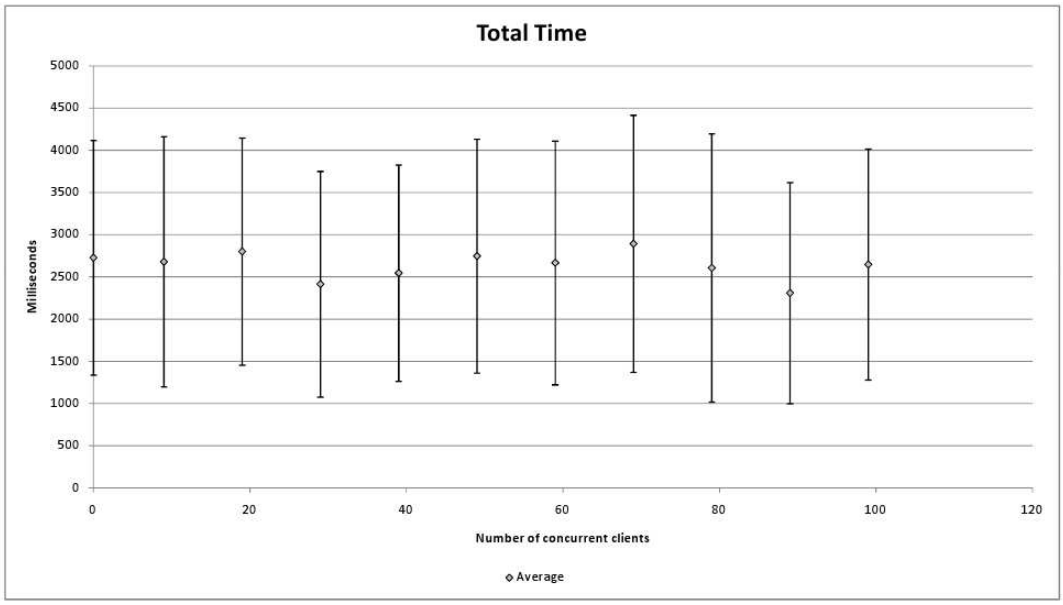


Figure C-4: Total time of the small-scale scalability experiment on the synthetic

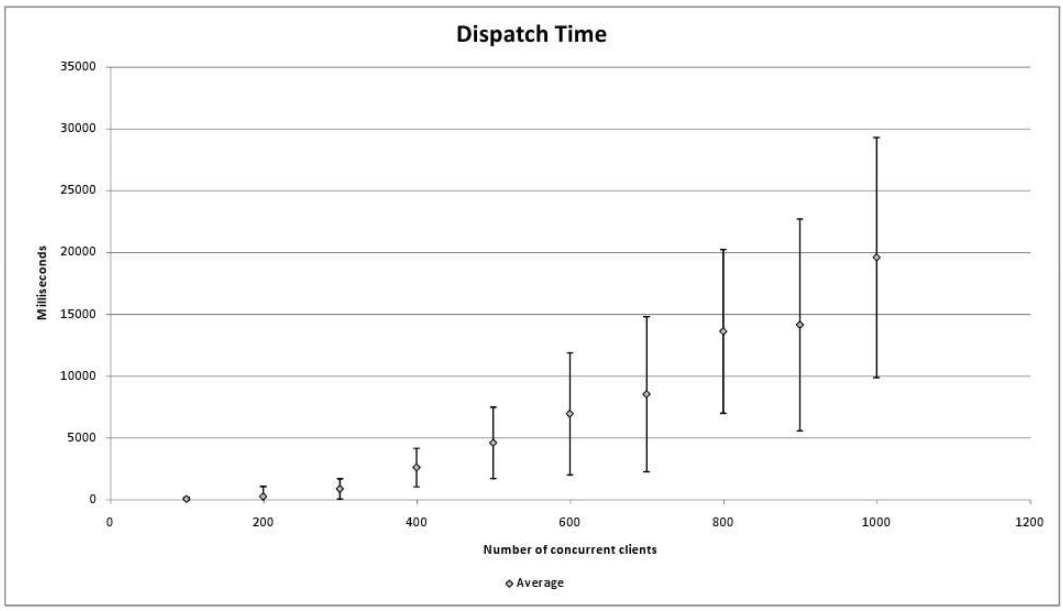


Figure C-5: Dispatch time of the large-scale scalability experiment on the synthetic testbed.

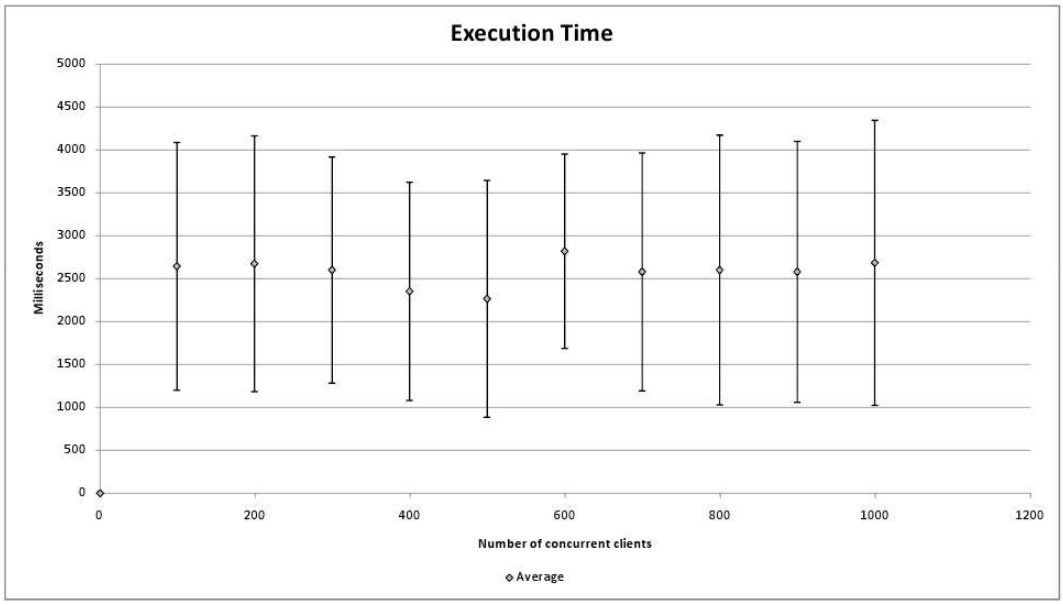


Figure C-6: Execution time of the large-scale scalability experiment on the synthetic

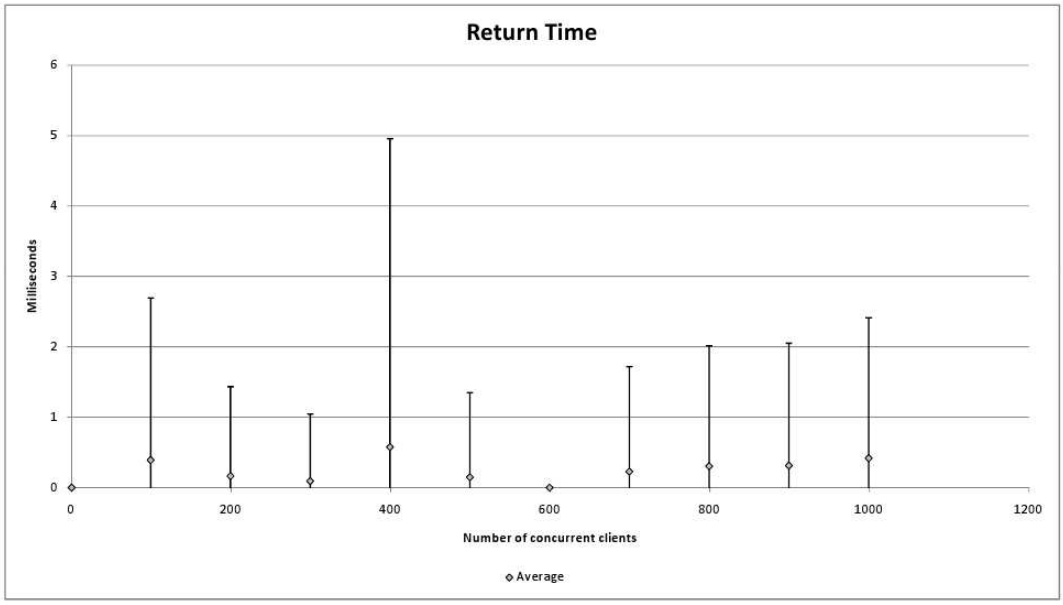


Figure C-7: Return time of the large-scale scalability experiment on the synthetic testbed.

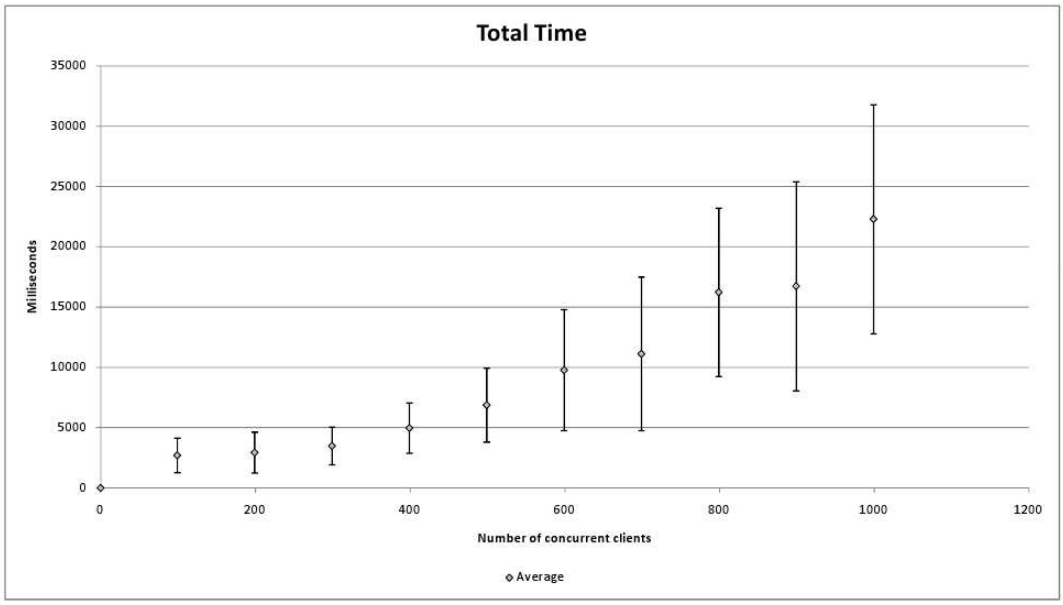


Figure C-8: Total time of the large-scale scalability experiment on the synthetic testbed.

Appendix D

Detailed Metrics of First Behavioural Experiment

Detailed results of the second behaviour experiment (described in Section 4.4.4) are shown in Figures D-1, D-2, D-2 and D-4.

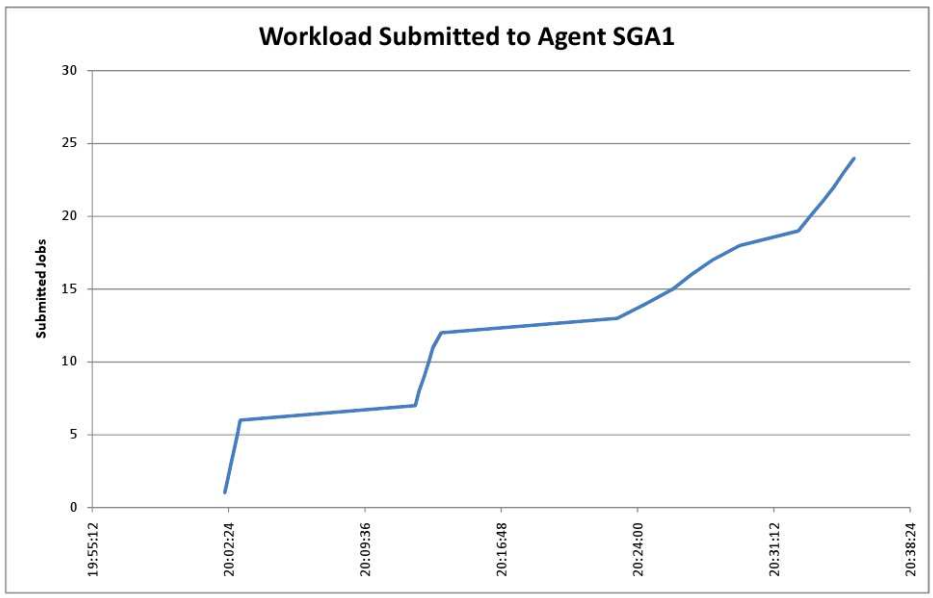


Figure D-1: Workload.

Figure D-1 shows the number of jobs submitted to agent *SGA1*.

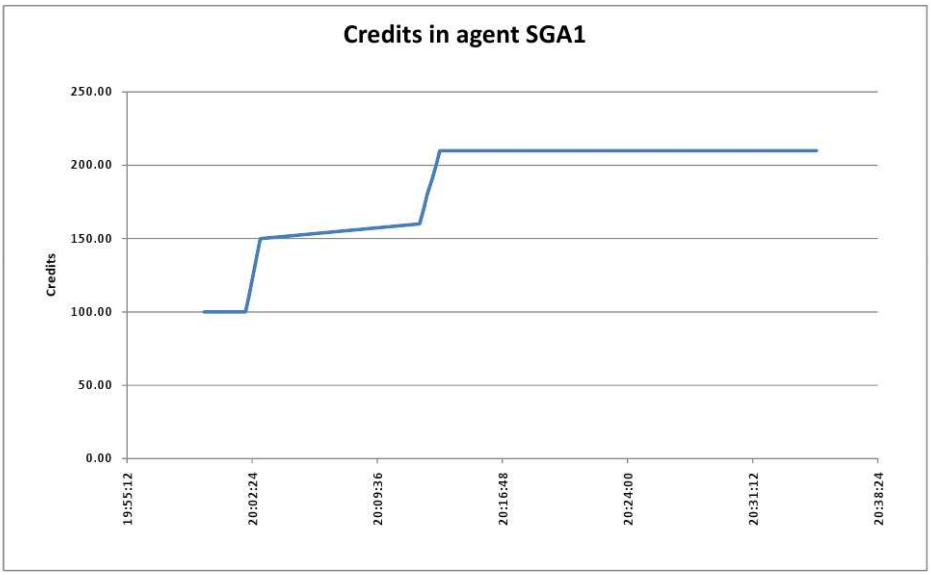


Figure D-2: Endowment of SGA1.

Figure D-2 shows the number of credits available to agent *SGA1*. Credits grow during the first two batches (*phase A* and *phase B*, detailed in Section 4.4.4) of jobs according to the policies that determine its behaviour (described in Section 4.4.4), in fact agent *SGA1* receives a payment for each job but it bears no costs for its execution as the first batch is performed by agent *PGA1* which is directly controlled while the second batch of jobs is executed through a Pub Topology that links agent *SGA1* with agent *PGA2*; in this case no credit is consumed in agent *SGA1* but tokens granted to it decrease in agent *PGA2* (detailed in Figure D-4). During *phase C* of the experiment, agent *SGA1* purchases job execution rights from agent *SGA3* (whose credits are detailed in Figure E-3) at the same price it receives, accordingly the amount of credits available to agent *SGA1* remains constant while the credits of agent *SGA3* grows.

Figure D-3 shows the number of credits available to agent *SGA3*. Credits remain

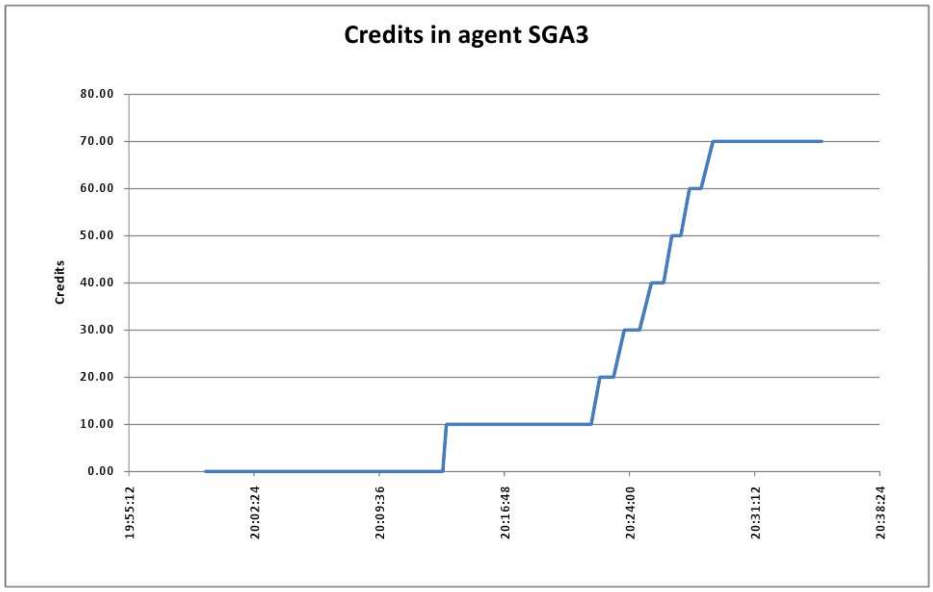


Figure D-3: Endowment of SGA3.

constant during the first two batches (*phase A* and *phase B*, detailed in Section 4.4.4) as the agent is not involved in the execution of any job. During *phase C* of the experiment, agent *SGA3* executes jobs for agent *SGA1* (whose credits are detailed in Figure D-2) accordingly the amount of credits available to agent *SGA3* grows.

Figure D-4 shows the number of tokens granted to agent *SGA1* The number of tokens remain constant during the first batch (*phase A* detailed in Section 4.4.4) as the agent is not involved in the execution of any job. During *phase B* of the experiment, agent *PGA2* executes jobs for agent *SGA1* accordingly the amount of tokens granted to agent *SGA1* decreases. During *phases C* of the experiment, agent *PGA2* is not involved in production and the amount of tokens granted to agent *SGA1* remains zero.

Figure D-4: Execution tokens granted by PGA2 to SGA1.

Appendix E

Detailed Metrics of Second Behavioural Experiment

Detailed results of the second behaviour experiment (described in Section 4.4.5) are shown in Figures E-1, E-2, E-3, E-4 and E-5.

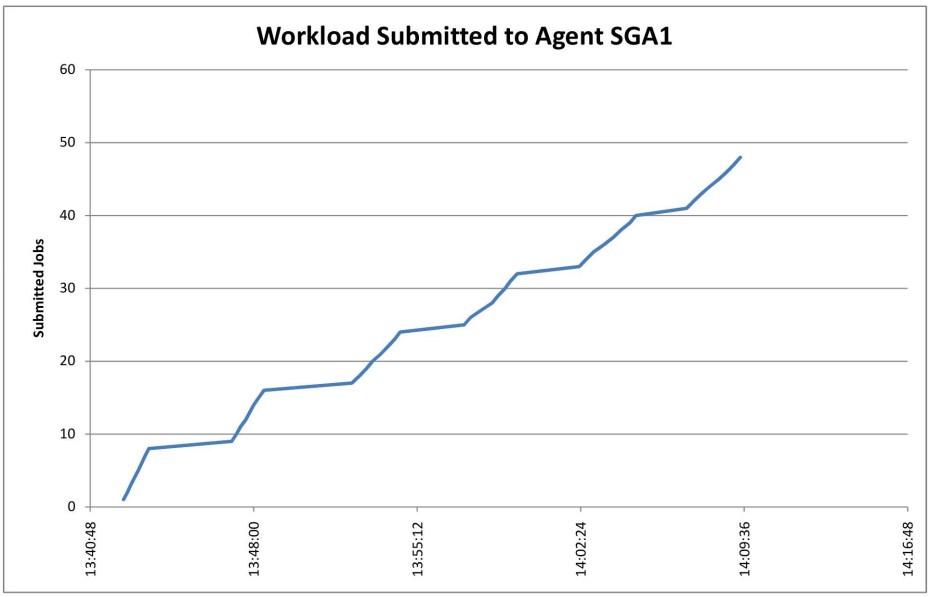


Figure E-1: Workload.

Figure E-1 shows the number of jobs submitted to agent *SGA1*.

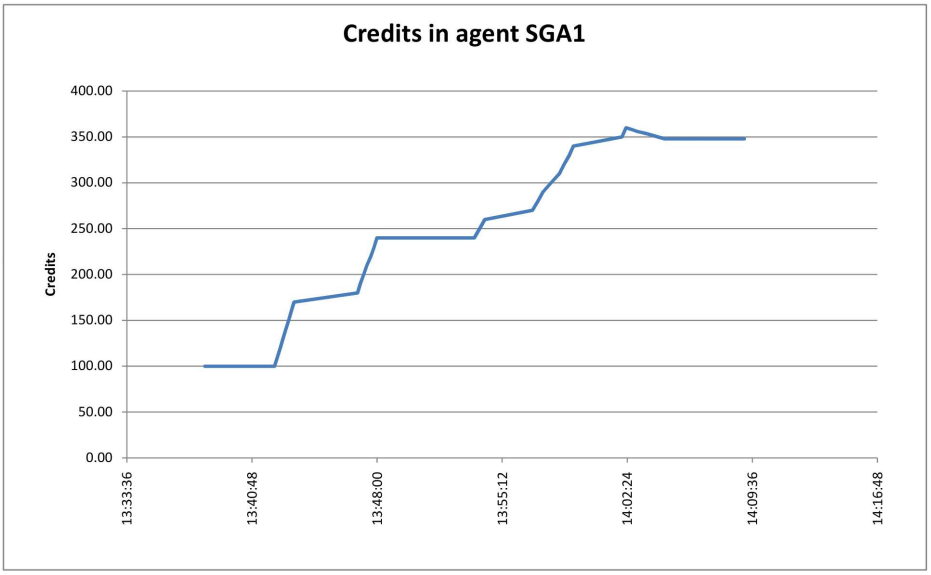


Figure E-2: Endowment of SGA1.

Figure E-2 shows the number of credits available to agent *SGA1*. Credits grow during the first two batches (*phase A* and *phase B*, detailed in Section 4.4.5) of jobs according to the policies that determine its behaviour (described in Section 4.4.5), in fact agent *SGA1* receives a payment for each job but it bears no costs for its execution as the first batch is performed by agent *PGA1* which is directly controlled while the second batch of jobs is executed through a Pub Topology that links agent *SGA1* with agent *PGA2*; in this case no credit is consumed in agent *SGA1* but tokens granted to it decrease in agent *PGA2* (detailed in Figure E-5). During *phase C* of the experiment, agent *SGA1* purchases job execution rights from agent *SGA3* (whose credits are detailed in Figure E-3) at the same price it receives, accordingly the amount of credits available to agent *SGA1* remains constant while the credits of agent *SGA3* grows. During *phase D* of the experiment, agent *SGA1* purchases job execution rights from agent *SGA4* (whose credits are detailed in Figure E-3) at

a higher price, accordingly the amount of credits available to agent *SGA1* decreases while the credits of agent *SGA4* (detailed in Figure E-4) grows.

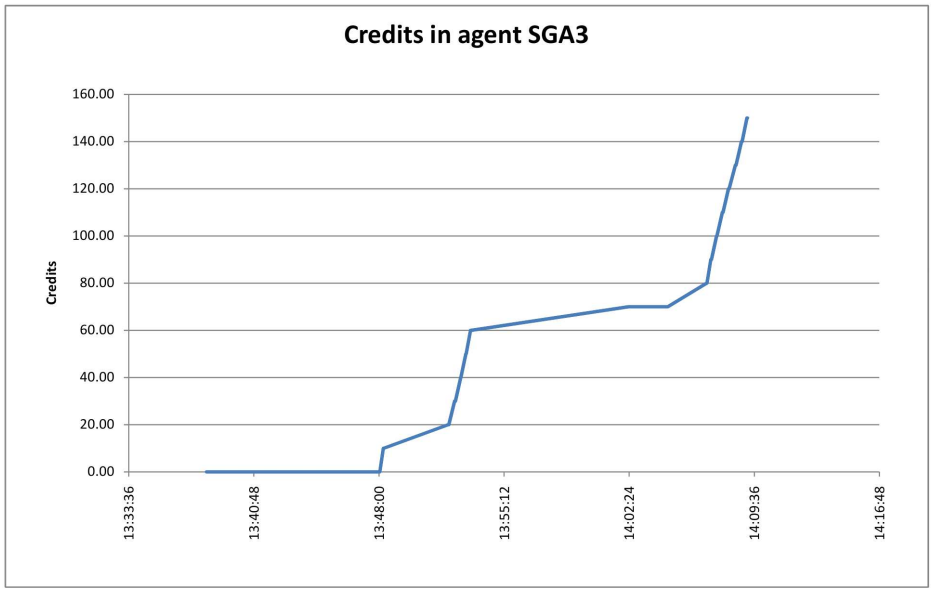


Figure E-3: Endowment of SGA3.

Figure E-3 shows the number of credits available to agent *SGA3*. Credits remain constant during the first two batches (*phase A* and *phase B*, detailed in Section 4.4.5) as the agent is not involved in the execution of any job. During *phase C* of the experiment, agent *SGA3* executes jobs for agent *SGA1* (whose credits are detailed in Figure E-2) accordingly the amount of credits available to agent *SGA3* grows. During *phase D* of the experiment, agent *SGA3* is not involved in production and the amount of its credits remain constant.

Figure E-4 shows the number of credits available to agent *SGA4*. Credits remain constant during the first three batches (*phase A*, *phase B* and *phase C*, detailed in Section 4.4.5) as the agent is not involved in the execution of any job. During *phase D* of the experiment, agent *SGA4* executes jobs for agent *SGA1* (whose credits are detailed in Figure E-2), accordingly the amount of credits available to agent *SGA4*

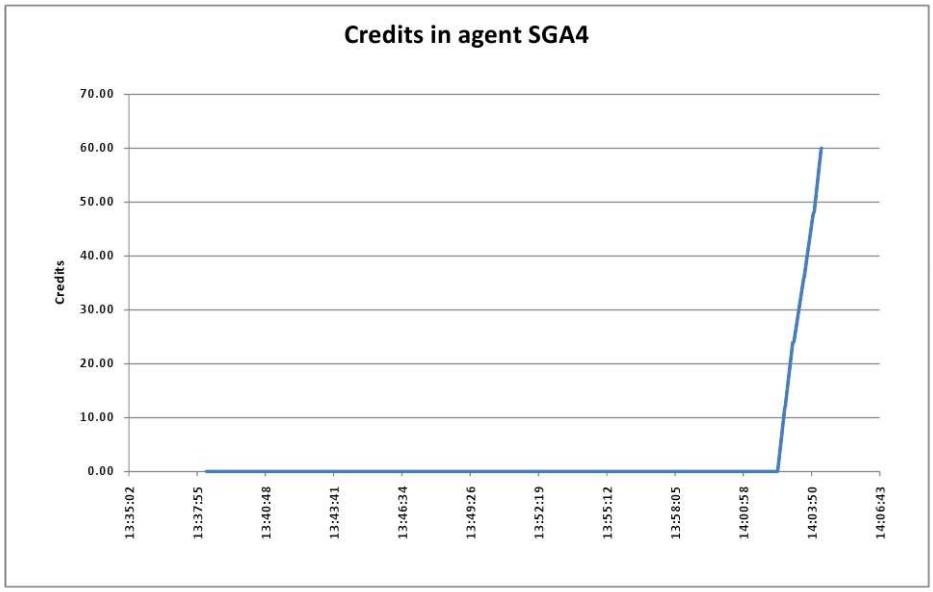


Figure E-4: Endowment of SGA4.

grows while the amount of credits of agent SGA1 decreases because the price paid by agent *SGA1* to agent *SGA4* is greater to the price paid to agent *SGA1*.

Figure E-5 shows the number of tokens granted to agent *SGA1* The number of tokens remain constant during the first batch (*phase A* detailed in Section 4.4.5) as the agent is not involved in the execution of any job. During *phase B* of the experiment, agent *PGA2* executes jobs for agent *SGA1* accordingly the amount of tokens granted to agent *SGA1* decreases. During *phases C and D* of the experiment, agent *PGA2* is not involved in production and the amount of tokens granted to agent *SGA1* remains zero.

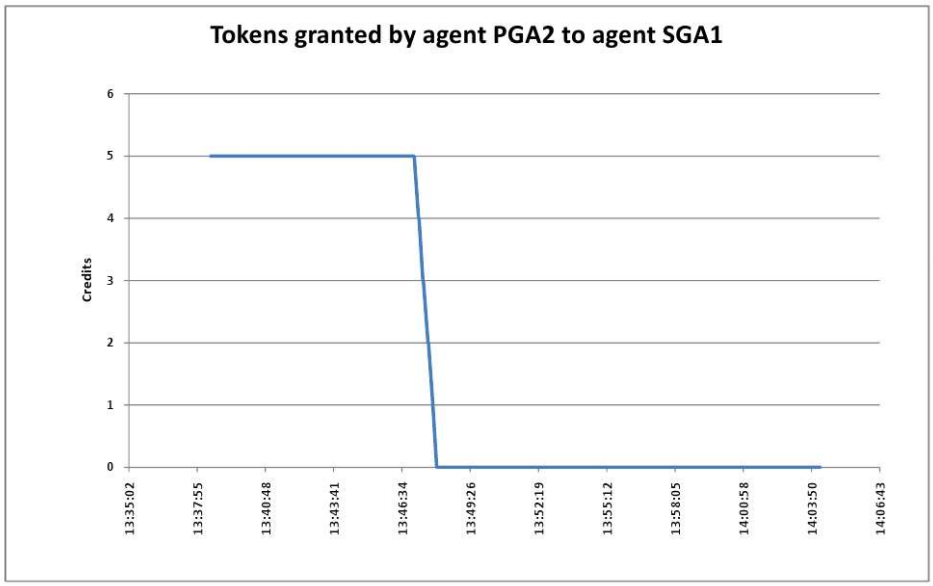


Figure E-5: Execution tokens granted by PGA2 to SGA1.

Bibliography

[1] ArguGRID Project webpage. http://www.argugrid.eu/, Last visited on April 2009.

[2] Assess grid project webpage. http://www.assessgrid.eu/, Last visited on April 2009.

[3] The Boinc Project. http://boinc.berkeley.edu/, Last visited on April 2009.

[4] CATNETS project webpage. http://www.catnets.uni-bayreuth.de/, Last visited on April 2009.

[5] Condor Project webpage. http://www.cs.wisc.edu/condor/, Last visited on April 2009.

[6] Corba webpage. http://www.corba.org/, Last visited on April 2009.

[7] Eclipse Project webpage. http://www.eclipse.org/, Last visited on April 2009.

[8] Global Grid Forum webpage. http://www.globalgridforum.org/, Last visited on April 2009.

[9] Grid Economy Project webpage. http://www.buyya.com/ecogrid/, Last visited on April 2009.

[10] Grid Market Directory webpage. http://www.gridbus.org/gmd/, Last visited on April 2009.

[11] Grid Service Broker webpage. http://www.gridbus.org/broker/, Last visited on April 2009.

[12] GridBus Project webpage. http://www.gridbus.org/, Last visited on April 2009.

[13] Haskell Project webpage. http://www.haskell.org/, Last visited on April 2009.

[14] Libra webpage. http://www.gridbus.org/libra/, Last visited on April 2009.

[15] Maui documentation webpage. http://www.dcsc.sdu.dk/docs/maui/mauidocs.html, Last visited on April 2009.

[16] Nimrod-G webpage. http://www.csse.monash.edu.au/~davida/nimrod/, Last visited on April 2009.

[17] Open Grid Forum webpage. http://www.ogf.org/index.php, Last visited on April 2009.

[18] Open LDAP webpage. http://www.openldap.org/, Last visited on April 2009.

[19] SORMA Project webpage. http://www.sormaproject.eu/, Last visited on April 2009.

[20] Ajith Abraham, Rajkumar Buyya, and Biakunth Nath. Nature Heuristics for Scheduling jobs on Computational Grids. In *Proc. 8th IEEE International Conference on Advanced Computing and Communications (ADCOM 2000)*, pages 45–52, Cochin, India, December 2000.

[21] A.Guarise, G. Patania, and R.M.Piro. DGAS User's Guide. https://edms.cern.ch/file/571271/1/EGEE-DGAS-HLR-Guide.pdf, Last visited on April 2009.

[22] Vassil N. Alexandrov, Ashish Thandavan, and Péter Kacsuk. Using P-GRADE for Monte Carlo Computations in a Distributed Environment. In Bubak et al. [26], pages 475–482.

[23] Fabrizio Bacini and Stefano Baco. WP1 - WMS Software Administrator and User Guide. https://edms.cern.ch/file/335068/0.2/DataGrid-01-TEN-0118-0_2-Document.pdf, January 2002.

[24] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. SOSP '03: 19th ACM symposium on Operating systems principles, Bolton Landing, New York, USA*, pages 164–177. ACM, October 2003.

[25] Jeffrey M. Bradshaw. An Introduction to Software Agents. *Software Agents, MIT Press, Cambridge, MA, USA*, pages 3–46, 1997.

[26] Marian Bubak, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors. *Computational Science - ICCS 2004, 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part IV*, volume 3039 of *Lecture Notes in Computer Science*. Springer, 2004.

[27] Stephen Burke, Simone Campana, Antonio Delgado Peris, Flavia Donno, Patricia Mendez Lorenzo, Roberto Santinelli, and Andrea Sciaba. gLite Users Guide. https://edms.cern.ch/document/722398/1.2, April 2008.

[28] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economic Models for Resource Management and Scheduling in Grid Computing. *Special Issue on Grid Computing Environments, The Journal of Concurrency and Computation, Practice and Experience (CCPE), Wiley Press*, pages 1507–1542, May 2002.

[29] Rajkumar Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing*. PhD thesis, Univ. Melbourne, Australia, Last visited on April 2009.

[30] K. Cassidy, J. Walsh, B. Coghlan, and D. Dagger. Using Hyperbolic Geometry for Visualisation of Concept Spaces for Adaptive e-Learning. In *Proc. 4th International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems (AH2006), Dublin, Ireland*, pages 421–426, June 2006.

[31] S. Childs, B. Coghlan, J. Walsh, and D. OCallaghan. A virtual TestGrid, or how to replicate a national Grid. In *Proc. ExpGrid workshop at HPDC2006, 15th IEEE International Symposium on High Performance Distributed Computing, Paris - France*, pages 47–54, February 2006.

[32] K. Czajkowski, F. D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The WS-Resource framework, version 1.0. http://www.oasis-open.org/committees/download.php/6796/ws-wsrf.pdf, March 2004.

[33] Antony Davies. Computational intermediation and the evolution of computation as a commodity. *Applied Economics*, 36(11):1131–1142, June 2004.

[34] Jared Diamond. *Guns, Germs, and Steel: The Fates of Human Societies*. W. W. Norton & Company, April 1999.

[35] Erik Elmroth and Peter Gardfjall. Design and Evaluation of a Decentralized System for Grid-wide Fairshare Scheduling. In *Proc. E-SCIENCE '05: 1st International Conference on e-Science and Grid Computing*, pages 221–229, Washington, USA, May 2005. IEEE Computer Society.

[36] Carsten Ernemann, Volker Hamscher, and Ramin Yahyapour. Economic Scheduling in Grid Computing. In *Scheduling Strategies for Parallel Processing*, pages 128–152. Springer, 2002.

[37] Torsten Eymann, Werner Streitberger, and Sebastian Hudert. CATNETS - Open Market Approaches for Self-organizing Grid Resource Allocation. In Jorn Altmann and Daniel Veit, editors, *Proc. GECON'07, Rennes, France*, volume 4685 of *Lecture Notes in Computer Science*, pages 176–181. Springer, August 2007.

[38] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. www.globus.org/alliance/publications/papers/ogsa.pdf, Last visited on April 2009.

[39] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, August 2001.

[40] Jan Foster, Carl Kasselman, Craig Lee, Bob Lindell, Klara Nahrsted, and Alain Roy. A Distributed Resource Management Architecture that Supports Advanced Reservations and Co-Allocation. In *Proc. 7th International Workshop on Quality of Service, London, UK*, pages 27–36, April 1999.

[41] John K. Galbraith. *The Affluent Society.* Mariner Books, October 1998.

[42] P. Gardfjll, E. Elmroth, L. Johnsson, O. Mulmo, and T. Sandholm. Scalable Grid-wide Capacity Allocation with the SweGrid Accounting System (SGAS). *Concurrency and Computation: Practice and Experience, published on line: http://www3.interscience.wiley.com/cgi-bin/fulltext/119816803/PDFSTART*, June 2007.

[43] Francesco Giacomini and Francesco Prelz. Definition Of Architecture, Technical Plan And Evaluation Criteria For Scheduling, Resource Management, Security And Job Description. https://edms.cern.ch/file/332413/1/datagrid-01-d1.2-0112-0-3.pdf, Last visited on April 2009.

[44] gLite Project. Official Documentation for LFC and DPM. https://twiki.cern.ch/twiki/bin/view/LCG/DataManagementDocumentation/, Last visited on April 2009.

[45] gLite Project. Storage Management Group webpage. http://sdm.lbl.gov/srm-wg/, Last visited on April 2009.

[46] J. Gomes, M. David, J. Martins, L. Bernardo, J. Marco, R. Marco, D. Rodrguez, Jos Salt, S. Gonzalez, Javier Snchez, A. Fuentes, Markus Hardt, A. Garca, P. Nyczyk, A. Ozieblo, Pawel Wolniewicz, M. Bluj, Krzysztof Nawrocki, Adam Padee, Wojciech Wislicki, C. Fernndez, J. Fontn, A. Gmez, I. Lpez, Yannis Cotronis, Evangelos Floros, George Tsouloupas, Wei Xing, Marios D. Dikaiakos, Jn Astalos, Brian A. Coghlan, Elisa Heymann, Miquel A. Senar, G. Merino, C. Kanellopoulos, and G. Dick van Albada. First Prototype of the Crossgrid Testbed. In F. Fernndez Rivera, Marian Bubak, A. Gmez Tato, and Ramon Doallo, editors, *Proc European Across Grids Conference, Santiago de Compostela, Spain*, volume

2970 of *Lecture Notes in Computer Science*, pages 67–77. Springer, February 2003.

[47] Ladislav Hluch, Ondrej Habala, Martin Maliska, Branislav Simo, Viet D. Tran, Jn Astalos, and Marian Babik. Grid Based Flood Prediction Virtual Organization. In *Proc. e-Science'06, Washington, DC, USA*, page 4. IEEE Computer Society, October 2006.

[48] James J. Kennedy John P. Morrison, David A. Power. An Evolution of the WebCom Metacomputer. *J. Math. Model. Algorithms*, 2:263–276, 2003.

[49] Chaitanya Kandagatla. Abstract Survey and Taxonomy of Grid Resource Management systems. http://arxiv.org/pdf/cs/0407012, Last visited on April 2009.

[50] John Kenneth Galbraith. *A History of Economics (The Past as the Present).* Penguin, Harmondsworth Eng., 1991.

[51] Jens Kenschik. Hypergraph webpage. http://hypergraph.sourceforge.net/, Last visited on April 2009.

[52] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software Practice and Experience*, (2):135–164, 2002.

[53] John Lamping and Rao Ramana. Visualizing Large Trees Using the Hyperbolic Browser. In *Proc. Conference on Human Factors in Computing Systems, Vancouver, Canada*, pages 388–389. ACM, April 1996.

[54] Gareth J. Lewis, Gergely Sipos, Florian Urmetzer, Vassil N. Alexandrov, and Péter Kacsuk. The Collaborative P-GRADE Grid Portal. In *Proc. ICCS'05, 5th International Conference on Computational Science, Atlanta, USA, Part III*, volume 3516 of *Lecture Notes in Computer Science*, pages 367–374. Springer, May 2005.

[55] Lijuan Lu and Yuhui Deng. A Time division pricing based economic model to enhance qos and resources utilization of grid services. *Proc. ChinaCom '06. 1st*

International Conference on Communications and Networking in China, Beijing, China, pages 1–4, October 2006.

[56] Solomon M. The ClassAd Language Reference Manual. http://www.cs.wisc.edu/condor/classad/, Last visited on April 2009.

[57] Daniel Minoli. *A Networking Approach to Grid Computing*. Wiley-Interscience, 2004.

[58] J. P. Morrison. *Condensed Graphs: Unifying Availability-Driven, Coercion-Driven, and Control-Driven Computing*. PhD thesis, Technische Universiteit Eindhoven, October 1996.

[59] J. P. Morrison, Power D, and Kennedy J. Load balancing and fault tolerance in a condensed graphs based metacomputer. *Journal of Internet Technologies, Special Issue on Web based Programming*, (3):221–234, December 2002.

[60] J. P. Morrison, J. J. Kennedy, and D. A. Power. WebCom: A Web Based Volunteer Computer. *The Journal of Supercomputing*, 18:47–61, 2001.

[61] Anjan Mukherji. *Walrasian and Non-Walrasian Equilibria: An Introduction to General Equilibrium Analysis*. Oxford University Press, USA, 1990.

[62] T. Munzner and P. Burchard. Visualizing the structure of the world wide web in 3d hyperbolic space. In *Proc. VRML 1995 Symposium, San Diego, California, USA*, pages 33–38, December 1995.

[63] J. Nakai and R. F. Van Der Wijngaart. Applicability of Markets to Global Scheduling in Grids. pages 1–37, February 2003.

[64] Dirk Neumann, Jochen Ster, Arun Anandasivam, and Nikolay Borissov. Sorma - building an open grid market for grid resource allocation. In Jorn Altmann and Daniel Veit, editors, *Proc. GECON'07, Rennes, France*, volume 4685 of *Lecture Notes in Computer Science*, pages 194–200,. Springer, August 2007.

[65] Jens Nimis, Arun Anandasivam, Nikolay Borissov, Garry Smith, Dirk Neumann, Niklas Wirstrom, Erel Rosenberg, and Matteo Villa. Sorma - business cases for an open grid market: Concept and implementation. In Jorn Altmann, Dirk Neumann, and Thomas Fahringer, editors, *Proc. GECON'08, Las Palmas, Spain*, volume 5206 of *Lecture Notes in Computer Science*, pages 173–184. Springer, August 2008.

[66] Organization for the Advancement of Structured Information Standards OASIS. MUWS 1.1 Part 1 Specification. http://www.oasis-open.org/committees/download.php/20576/wsdm-muws1-1.1-spec-os-01.pdf, Last visited on April 2009.

[67] David O'Callaghan and Brian Coghlan. Bridging Secure WebCom and European DataGrid Security for Multiple VOs over Multiple Grids. In *Proc. ISPDC 2004*, pages 225–231, Cork, Ireland, July 2004. IEEE Computer Society.

[68] Francesco Pacini and Peter Kunszt. JDL Attributes Specification (submission through Network Server). https://edms.cern.ch/file/555796/1/EGEE-JRA1-TEC-555796-JDL-Attributes-v0-4.pdf, Last visited on April 2009.

[69] Francesco Pacini and Alessandro Maraschini. Job Description Language (JDL) Attributes Specification (Submission through the WMProxy Service). https://edms.cern.ch/file/590869/1/EGEE-JRA1-TEC-590869-JDL-Attributes-v0-4.pdf, Last visited on April 2009.

[70] M. A. Pettipher, A. Khan, T.W. Robinson, and X. Chan. Review of Accounting and Usage Monitoring Final Report. http://www.jisc.ac.uk/publications/publications/accountingusagereport.aspx, Last visited on April 2009.

[71] G. Pierantoni, E. Kenny, and B. Coghlan. An Agent-based Architecture for Grid Societies. In *Proc. PARA'06, Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, Umeå, Sweden, Revised*

Selected Papers, volume 4699/2008 of *Lecture Notes in Computer Science*, pages 830–839. Springer, June 2006.

[72] G. Pierantoni, E. Kenny, B. Coghlan, O. Lyttleton, D. O'Callaghan, and G. Quigley. Interoperability using a Metagrid Architecture. In *Proc. HDPC'06, Paris, France*, pages 23–30, June 2006.

[73] G. Pierantoni, O. Lyttleton, D. O'Callaghan, G. Quigley, E. Kenny, and B. Coghlan. Multi-Grid and Multi-VO Job Submission based on a Unified Computational Model. In *Proc. Cracow Grid Workshop (CGW05)*, pages 341–349, Kracow, Poland, November 2005.

[74] Rosario M. Piro, Michele Pace, Antonia Ghiselli, Andrea Guarise, Eleonora Luppi, Giuseppe Patania, Luca Tomassetti, and Albert Werbrouck. Tracing Resource Usage over Heterogeneous Grid Platforms: A Prototype RUS Interface for DGAS. In *Proc. E-SCIENCE '07: 3rd IEEE International Conference on e-Science and Grid Computing*, pages 93–101, Washington, DC, USA, 2007. IEEE Computer Society.

[75] P-Grade portal project. PROVE Presentation. http://www.lpds.sztaki.hu/pgrade/p_grade/pgrade/sld018.html, Last visited on April 2009.

[76] EGRID Project. STORM webpage. http://www.egrid.it/about/collaborations/storm_description/, Last visited on April 2009.

[77] Unicore Project. Unicore webpage. http://www.unicore.eu/index.php, Last visited on April 2009.

[78] Tim Puschel, Nikolay Borissov, Mario Macas, Dirk Neumann, Jordi Guitart, and Jordi Torres. Economically Enhanced Resource Management for Internet Service Utilities. In Boualem Benatallah, Fabio Casati, Dimitrios Georgakopoulos, Claudio Bartolini, Wasim Sadiq, and Claude Godart, editors, *WISE*, volume 4831 of *Lecture Notes in Computer Science*, pages 335–348,. Springer, December 2007.

[79] Raman Rajesh, Livny Miron, and Solomon Marvin. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proc. 7th IEEE International Symposium on High Performance Distributed Computing, Chicago, Illinois, USA*, pages 28–31, July 1998.

[80] Raman Rajesh, Livny Miron, and Marvin Solomon. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In *Proc. 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12 '03), Seattle, Washington, USA*, pages 80–89, June 2003.

[81] Rajesh Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, The University of Wisconsin, Last visited on April 2009. Supervisor: Miron Livny.

[82] Keith Rochford, John Walsh, Eamonn Kenny, and Brian Coghlan. A Standards-Based Architecture for Grid Service Management. In *Proc. ISPDC '07, 6th Parallel and Distributed Computing, Hagenber, Austria*, page 24, USA, July 2007. IEEE Computer Society.

[83] Ed Seidel, Gabrielle Allen, André Merzky, and Jarek Nabrzyski. Gridlab: a grid application toolkit and testbed. *Future Gener. Comput. Syst.*, 18(8):1143–1153, 2002.

[84] William F. Sharpe. *The Economics of Computers*. Columbia University Press, New York, NY, USA, 1972.

[85] Jahanzeb Sherwani, Nosheen Ali, Nausheen Lotia, Zahra Hayat, and Rajkumar Buyya. Libra: a computational economy-based job scheduling system for clusters. *Software Practice and Experience*, 34(6):573–590, 2004.

[86] A. Sim and V. Natarajan. OGF — Grid Interoperability Now (GIN) webpage. https://forge.gridforum.org/projects/mgi/, Last visited on April 2009.

[87] Joseph Stiglitz. *Globalization and its discontents*. Penguin, London, 2002.

[88] Joseph E. Stiglitz. *The Roaring Nineties: Why We're Paying the Price for the Greediest Decade in History*. Penguin Books Ltd, June 2004.

[89] PJ Strange, T Antoni, F Donno, H Dres, G Grein, G Mathieu, A Mills, D Spence, T Min, and M Verlato. Global GRID User Support : The Model and Experience in LHC Computing GRID. http://epubs.cclrc.ac.uk/work-details?w=35266, Last visited on April 2009.

[90] Francesca Toni. E-business in ArguGRID. In *Proc. GECON'07, Rennes, France*, pages 164–169, August 2007.

[91] Globus Toolkit Project. Globus GRAM webpage. http://www.globus.org/toolkit/docs/2.4/gram/, Last visited on April 2009.

[92] Globus Toolkit Project. Globus Toolkit webpage. http://www.globus.org/toolkit/, Last visited on April 2009.

[93] Globus Toolkit Project. Resource Specification Language webpage. http://www-fp.globus.org/gram/rsl_spec1.html, Last visited on April 2009.

[94] Sven van de Berghe, Gilbert Netzer, Rosario Piro, Morris Riedel, and Davy Virdee. JRA1 Accounting Milestone: Preliminary design documents. omii-europe.com/OMIIEurope/News/OMIIEuropeMJRA1.4.pdf, April 2009.

[95] Maoguang Wang, Zhongzhi Shi, and Shifei Ding. Hierarchical policy for agent grid collaboration. In *GCC '07: Proceedings of the Sixth International Conference on Grid and Cooperative Computing*, pages 236–241, Washington, DC, USA, 2007. IEEE Computer Society.

[96] R. Wolski, J. Brevik, J. S. Plank, and T. Bryan. Grid Resource Allocation and Control Using Computational Economies. pages 747–772, 2003.

[97] Rich Wolski, James S. Plank, John Brevik, and Todd Bryan. Analyzing Market-Based Resource Allocation Strategies for the Computational Grid. *International Journal High Performamce Computing Application*, 15(3):258–281, 2001.

[98] Chunling Zhu, Xiaoyong Tang, Kenli Li, Xiao Han, Xilu Zhu, and Xuesheng Qi. Integrating Trust into Grid Economic Model Scheduling Algorithm. In *Proc. OTM Confederated International Conference, Montpellier, France*, volume 2, pages 1263–1272. Springer Berlin, October 2006.