Social profiles for dynamically configurable agents in large scale cloud, grid, and heterogeneous infrastructures

by

Peter D. Lavin

Master of Science

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

Doctor of Philosophy

School of Computer Science and Statistics,
University of Dublin,
Trinity College

2014

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 Peter D. Lavin.

Signature of Author . . . .

Peter D. Lavin April, 2013

Abstract

A vast amount of computing resources are available throughout the world today. These are distributed worldwide, and are heterogeneous in platform, origins, motivations, ownership and control. Many large computational challenges also exist, which if addressed, would require a great amount of computing resources, often too much to be addressed by one single type of computing resource.

In attempting to assemble enough resources to address these large challenges, two difficulties present; interoperability, and the diversity found in the social and economic backgrounds that both the work and resources may have. Multiple distributed agents offer an interoperability solution by wrapping services which need to interact with each other. Agents can represent computing resources and work, modelling and executing the types of interactions needed to make resource allocation decisions. In an ever changing resource allocation environment, agents need to be easy to create, configure and reconfigure.

Social Grid Agents (SGAs) are designed to address resources allocation in grid computing. Being able to operate and communicate in a distributed heterogeneous environment, they offer a solution for interoperability. SGAs interact with one another and enforce the work allocation policies of the resources they represent. This is done using the Boolean matching features of the Classad language. But matching of subtle and nuanced social and economic backgrounds presents a challenge to Boolean matching mechanism. To overcome this challenge, agents need a mechanism which can describe and differentiate between the social and economic differences among the resources and work which they are allocating.

This thesis extends previous research in Social Grid Agents. It enhances existing

SGA technology to become more scalable and configurable. It also proposes using textual descriptions in unstructured language to describe the social and economic outlooks that computing resources and workloads may have. To measure the similarity between these descriptions, search technology is integrated into the agents. These descriptions are called ‘social profiles’.

This thesis provides novel mechanisms to improve scalability, configurability and programmability of SGAs. Serialisation and class loading techniques are employed to achieve dynamic configurability of ‘live’ agents. An extensible and versatile format for the social profiles is developed. This thesis also provides mechanisms to exchange social profiles using distributed search technology and serialisation. Embedded search technology in SGAs allows quantitative similarity measurements to be determined for social profiles, yielding information useful in resource allocation decisions. An extensive set of experimental case studies demonstrate the efficacy of these techniques.

Acknowledgements

I would like to thank all my family and friends who supported and helped me during my time in Trinity College, and also my colleagues in the Computer Architecture and Grid group, Eamonn, John, Gabriele, Kathryn, David, Stuart and Kim.

Special thanks to my supervisors, past and present, Brian Coghlan and David Gregg, for patient proofreading, advice and support.

Dedications

The work of thesis is dedicated to some of the people who helped, supported and inspired me over many years.

To my parents, John and Bernadette, for unending hard work and sacrafices made so that my siblings and I could have everything we needed. To Mary Fitzmaruice, for her always present generous spirit and support. To Josephine Lawlor, who sowed the seeds of whatever ‘can do’ attitude I may have, something perhaps useful while working towards a doctoral thesis. Finally to my late aunt Sr. Pauline Lavan, for her vision, character and self-belief, things she wanted everyone to have. I hope that something from you all has rubbed off on me - thank you.

Contents

1 Introduction **1**

1.1 Computing Resources 1

1.1.1 Candidate Work for Resources 3

1.1.2 Interoperability 3

1.2 Software Agents, Interoperability and Resource Allocation 4

1.2.1 Social Grid Agents 4

1.2.2 Changes in Resources and SGAs 5

1.3 Determining Similarity Between Resources 6

1.4 Research Problem 7

1.4.1 Research Aims and Questions 7

1.5 Layout of this Thesis 9

1.6 Publications from this Thesis 10

2 Computational Resources **11**

2.1 Introduction 11

2.2 Academic and Scientific Research Grids 11

2.2.1 An Example Grid and Middleware – EGI and gLite 12

2.2.2 Specialty Grids 17

2.2.3 Condor and HT Condor 18

2.2.4 Academic Resources Summary 19

2.3 Cloud: Commercially Available Resources 19

2.3.1 Hybrid Clouds 21

2.3.2 Cloud: Advantages and Concerns 21

2.3.3 Social and Economic Considerations of Clouds 22
2.4 Public Resourced Computing 22
2.4.1 BOINC 23
2.4.2 BOINC Technical Details 24
2.4.3 Ownership and Control 25
2.4.4 Trust, Security and Malicious Behavior 25
2.4.5 World Community Grid 26
2.4.6 Social and Economic Considerations in PRC 26
2.4.7 XtremWeb 27
2.5 Interoperability Between Resources 28
2.5.1 Interoperability of Grid and Volunteered Resources 28
2.5.2 The EDGeS Project 29
2.5.3 The Lattice Project 30
2.6 Different Types of Candidate Computing Work 31
2.6.1 Academically-Resourced Work 31
2.6.2 Commercially-Driven Work 31
2.6.3 Publicly-Resourced Work 32
2.7 Work Allocation 32
2.7.1 The Classad Language 32
2.7.2 Resource Allocation in Grids 34
2.7.3 Resource Allocation in Volunteer Computing 35
2.7.4 Resource Allocation in Commercial Cloud 36
2.8 Summary 36

3 Software Agents **38**
3.1 Introduction 38
3.1.1 Definitions of Software Agents 38
3.1.2 Agents for Wrapping Other Technologies 41
3.1.3 Mobile Software Agents 41
3.1.4 Agent Communication Languages 42
3.1.5 Agent Oriented Design – Discussion 44

3.2 Social Grid Agents 46

3.2.1 SGA Internal Architecture 47

3.2.2 Classad in SGAs 48

3.2.3 SGA Communications 49

3.2.4 SGAs and Topologies 50

3.2.5 Migrating SGAs to New Web Service Technologies 50

3.3 SGAs compared to Similar Technologies 51

3.3.1 Example of a Similar Agent Technology 51

3.3.2 SGAs and Mobility 52

3.3.3 Web Service Orchestration 52

3.4 What Next for Social Grid Agents? 53

4 A Proliferating Multiple Agent System 54

4.1 Introduction 54

4.1.1 Multiple Agent System Planning 55

4.1.2 Chapter Layout 55

4.2 A Proliferatable Multi-Agent System of SGAs 56

4.2.1 Basic Agent Strategy 56

4.2.2 Reproduction of Basic SGAs 57

4.2.3 Addition of Binary Abilities Using Serialisation 57

4.3 Making a Multiple Agent System of SGAs 59

4.3.1 Details Needed for a single SGA Deployment 59

4.3.2 Agent Configuration Files for SGAs 60

4.4 Testing All Steps Together 61

4.4.1 Test Parameters and Preparation 62

4.4.2 Execution of the ‘MAS Creation’ Script 62

4.4.3 Removal of SGAs from an MAS 64

4.5 Results and Observations 64

4.5.1 MAS Creation 64

4.5.2 Memory Limitations 64

4.6 Abilities as Classes and Instances 65

4.6.1 Important Concepts of Agent Configuration 66

4.6.2 Loading Class Instances in SGAs 70

4.6.3 Configuration Information for Classes and Instances. 72

4.7 Automated Creation of Abilities Configuration Files 74

4.7.1 Analysing and Specifying SGA Configuration 74

4.7.2 Serialised Object File Creation 76

4.7.3 End-to-End Overview of an MAS Creation Framework 78

4.8 Including External Abilities Configuration Files 81

4.9 Other Developments in SGAs 82

4.9.1 Making Agents Within an MAS – Discussion 82

4.9.2 General Purpose Message Sending Format for SGAs 83

5 Social Profiles in SGAs 86

5.1 Introduction 86

5.2 Social Evaluations — A Discussion 87

5.2.1 Social and Economic Differences 88

5.3 Modeling and Measuring Meaning 90

5.3.1 A Model of Meaning – The Vector Space 90

5.4 Lucene as a VSM Implementation 93

5.4.1 The Inverted Index 93

5.4.2 Lucene Documents 93

5.4.3 Lucene Indexing Options 95

5.4.4 Querying the Vector Space for Documents 96

5.4.5 Cosine Similarity 97

5.4.6 Lucene Scalability and Efficiency 99

5.5 Social Profiles 100

5.5.1 Content of a Social Profile 101

5.5.2 Structure of Social Profile 102

5.5.3 XML as a Social Profile Container 104

5.5.4 Extensibility of A Social Profile 104

5.5.5 An Extensibility Solution 105

5.6 Search in Other Areas of Work Allocation 108
5.7 Other Methods of Evaluating Text 109
5.7.1 Sentiment Analysis 109
5.7.2 Fuzzy Logic 109
5.7.3 Relevance to Social Profiles and SGAs 110
5.8 Summary 110

6 Usage Experiments 112
6.1 Overview of Experiments 112
6.1.1 Experiment Summaries 112
6.2 Environment for Experiments 116
6.3 Experiment 1: MAS Creation Functionality 117
6.3.1 Description of the System to be Created 118
6.3.2 Experiment Environment Considerations 120
6.3.3 Running the Experiment 123
6.3.4 Evaluation of Outcome 123
6.3.5 Summary of Experiment Result 126
6.4 Experiment 2: Indexing, Retrieval and Similarity 127
6.4.1 Experiment Description and Layout 127
6.4.2 Index Preparation 129
6.4.3 Part (A) — Using Designed and Designless Queries 132
6.4.4 Part (B) — More-like-this Queries 141
6.4.5 Analysis 143
6.4.6 Outcome and Evaluation 143
6.4.7 Discussion 145
6.5 Experiment 3: Integrating Search into SGAs 146
6.5.1 Criteria for Evaluation 146
6.5.2 Experiment Description and Layout 147
6.5.3 Technical Requirements — Libraries 148
6.5.4 Requesting and Returning a Social Profile 149
6.5.5 Querying Another SGA's Index 152

6.5.6 Calculating Cosine Similarity in an SGA 155
6.5.7 Overall Evaluation 157
6.5.8 Discussion 158
6.6 Experiment 4: Using Similarity for Resource Allocation 159
6.6.1 Criteria for Success 159
6.6.2 Experiment Description 159
6.6.3 Abilities Required 160
6.6.4 Evaluation of Price and Similarity 160
6.6.5 Information Matrix and Decision Logic 161
6.6.6 Sequence for Price and Similarity Evaluation 163
6.6.7 Using Price and Similarity Values in a Classad Function . . . 163
6.6.8 Outcome 163
6.6.9 Evaluation 164
6.6.10 Discussion 164
6.7 Experiment 5: External JARs Over HTTP 167
6.7.1 Experiment Description 167
6.7.2 Criteria for Success 168
6.7.3 Procedure for Experiment 168
6.7.4 Running the Experiment 171
6.7.5 Evaluation and Contributions 172
6.8 Experiment 6: Using Includes URLs in ACFs 173
6.8.1 Criteria for Success or Failure 173
6.8.2 Experiment Description and Methodology 173
6.8.3 Outcome 174
6.8.4 Evaluation 176
6.8.5 Other Prototype Tests for Included URL in ACFs 176
6.8.6 Discussion 177
6.9 Experiment 7: Profile Structure Class 178
6.9.1 Objectives of Experiments 178
6.9.2 Adding Local Information to the Indexed Social Profile 179

6.9.3 ProfileStructure Class Size 182
6.9.4 Evaluation 183
6.9.5 Discussion 183
6.9.6 Contribution 185

7 Conclusions 186

7.1 Scalability and Configurability of SGAs 187
7.1.1 Creating Multiple SGAs 188
7.1.2 Flexible Agent Configuration 188
7.1.3 Discussion 190
7.2 Integrating Search into Distributed Agents 191
7.2.1 Encapsulated Binary Schema 192
7.2.2 Similarity Measurements 193
7.2.3 Discussion 194
7.3 Final Thoughts 195

A Sample gLite Commands 196

B Class Loading Issues Specific to Tomcat 198

B.1 JVM and Tomcat Class Loader Architecture 198
B.1.1 Loading De-Serialized Classes in Tomcat 199
B.1.2 Usage of Both Solutions 200

C Inventory List of Classes Required for All SGAs in Experiment 201

D Abridged List of Classes and JAR files in a Basic SGA WAR Package 202

E Log Outputs for Wallet Ability Configuration Details 203

F Log Outputs for JSAsyncProcessor Ability Configuration Details 204

G Sample Outputs of Trial Messages to SGA Experiment 205

H Outcomes of Term Query Searches 206

I Output from Includes in XML Abilities Files **208**

J Output from URL Class Loader HTTP Requests **210**

K Statistical Charts for Experiment 2 in Chapter 6 **212**

L Issues with SGAs in GT4 Technology **215**

L.1 Globus Implementation and SGAs 215

M Discussion on CXF and Tomcat Frameworks **217**

M.1 CXF — Tomcat and Globus Toolkit 4 Comparison 217

M.1.1 Archive File Structure and Storage 218

M.1.2 File Expansion on Host Machine 218

M.1.3 Disk Usage on Host Machine 218

N Sample ClassAd Expressions **219**

Bibliography **221**

List of Figures

2.1 A simple Job Description Language (JDL) file. 15

2.2 A Simplified Grid Overview. 16

2.3 Simple used car advertisements with equivalent Classad expressions. . 33

3.1 Abstract architecture of an SGA. 48

4.1 Structure of abilities configuration XML file. 61

4.2 Sample line from an MAS Plan File (MPF), columnised for clarity. . 62

4.3 Sequence for MAS Creation 63

4.4 Logic for retrieval of a class definition or an instance from the SGA data-structure 67

4.5 Sequence for creating a class instance in a hard-coded in the SGA of Pierantoni [107]. 68

4.6 Sequence for deserialising a class instance from serialised content in an XML ACF file in the SGA, as proposed in this thesis. 71

4.7 Methods used in SGAs to add binary abilities. 73

4.8 Sample lines from the abilities blueprint CSV file used to specify details of what is to be added to the XML abilities configuration file (ACF). 75

4.9 Mapping of abilities blueprint CSV file line to an ability element in the abilities configuration XML abilities file. 79

4.10 Summary of all parts which contribute to creation of a Multiple Agent System. 80

4.11 Structure of XML abilities configuration file with included URLs. . . 82

5.1 Sample Sentence. 91

5.2 Vector for Sample Sentence in figure 5.1. 91

5.3 Term Frequency-Inverse Document Frequency Equation. 92

5.4 A Simple Lucene Document Schema. 94

5.5 Sequence for social profile indexing using a class encapsulated in the XML file. 107

6.3 Layout of the eight SGAs to be recreated using the MAS creation mechanism. 119

6.13 General layout for SGAs used in Lucene integration experiments. . . 150

6.14 Sequence of messages for simple request and return of social profiles between SGAs. 151

6.15 Sequence of messages for sending a social profiles from one SGA to another. 152

6.16 Cosine-similarity calculation for an SGA's profile and that of a remote SGA. 157

6.17 SGA multiple-agent system for using similarity for resource allocation. 160

6.19 Logic for resource selection based on price and similarity measurements.162

6.20 Sequence for assembling information needed for price and similarity evaluation. 165

6.21 Classad function statements to implement the logic of figure 6.19. . . 166

6.22 Conventional Java Code with Import of Classes. 169

6.23 Java Code for Remote Classes. 170

6.24 SGA access to a remote JAR file using URL class loader. 171

6.25 Scenario for distributing all 11 required abilities over several ACFs. . 175

6.26 Scenario for sharing ACF abilities files between agents. 175

1.1 Sample gLite commands 197

10.1 Output from URL Class Loader HTTP Requests 211

11.1 Graphical summary of data from randomly generated queries for category 1, section 6.4. 213

11.2 Plot from Anderson-Darling Normality test on data from randomly generated queries for category 1, section 6.4. 214

Chapter 1

Introduction

To introduce the problem area addressed in this thesis, let us consider two things. Firstly, many current computational challenges require a large amount of computing resources, often too large to be addressed by one single type of computing resource. Secondly, throughout the world today, there is also a vast amount of computing resources available. Let us suppose that some of these resources can be assembled in a way which will make it possible to address some of these large bodies of work. The problems arising in trying to assemble enough of these resources to address such challenges are in the domain of resource and work allocation.

1.1 Computing Resources

Let us first examine some of these different computing resources, and in particular, their origins, the reasons why they exist and what they are used for.

Of those resources that are available, academic research grids and commercially available cloud computing clusters are typical examples. The total size of all these resources is hard to estimate. But as an indication, the European Grid Infrastructure (EGI) [1] alone provides over 320000 logical CPUs and 340 petabytes of storage. That infrastructure processes up to 1.7 million individual grid jobs per day, typically doing work which originates in the scientific and academic community. The actual hardware which makes up this infrastructure is owned by many national research institutions.

These institutions are located all over the world and they collaborate to support, use and share their resources.

Commercially available cloud computing first appeared in the mid-2000s. This service provides scalable computing to users on a commercial basis. Leading providers are Amazon, RackSpace and Hewlett Packard [2, 3, 4, 5]. Secure storage and computing services are offered to individuals or enterprises, with capacity ranging up to petabytes of storage and tens of thousands of available CPUs.

In addition to Grid and Cloud, public resourced computing (PRC) or volunteer computing is a popular internet-wide phenomenon where members of the public donate the use of their computers for use in research projects [47, 45]. Sometimes knows as *desk-top grids*, many individuals' computers, all working in parallel, carry out small portions of a large body of work. Members of the public are motivated to do this, often by a desire to support and participate in scientific research projects which appeal to them. Humanitarian and medical research, as well as difficult and protracted problems in maths and science are popular themes of publicly resources projects. Connectivity between the projects and their many volunteers is provided by middleware, typically the *Berkeley Open Infrastructure for Network Computing* or BOINC. Volunteered computing provides a substantial resource. There are frequently in excess of 1.2 million volunteered computers using the BOINC middleware, making, in effect, an aggregated system with computing capacity to do 8600 TeraFLOPS[1].

All of the above resources are heterogeneous in technological and physical form. They are also in geographically distributed locations. Furthermore, many of them owe their existence to a wide range of motivations and objectives, each having different origins, ownership and control. It can be said that these resources exist for a wide range of different *social* and *economic* reasons.

[1] Source: BOINC website statistics, retrieved April 2013, http://boinc.berkeley.edu and http://boincstats.com/en/stats/-1/project/detail/overview/.

1.1.1 Candidate Work for Resources

Something similar can be said of the many types of work which may be carried out on these resources. While work may be suitable for one type of resource, it may be infeasible or unacceptable to process it on another. For example, it is often the case that humanitarian work does not attract the funding or investment necessary to purchase commercial resources, and hence, this work is often only undertaken by volunteer computing projects. Consumers of commercially provided storage or computing may have reservations about allowing their data be processed in the relatively insecure environment of volunteered resources [116]. As a further example, governments that invest in computer resources for academic institutions do so for the long-term benefit of society. The resources are owned and controlled by the institutions and are not usually available for commercial work.

1.1.2 Interoperability

Let us return to the idea that some of these resources could be usefully brought together and employed to address a large computing task. For that to happen, computer systems, all of which may be different, need to be able to communicate and interact with each other in a way that allows them each to work correctly; this is called 'interoperability'. The problem of achieving interoperability between disparate and heterogeneous resources is still an active research area and there are many solutions which allow different resources to interact with each other. For example, the EDGeS project [122] has developed an interoperability solutions for a number of different desk-top grids and academic grid resources. The Lattice project [100, 6] offers a comparable solution where grid and BOINC resources are also presented alongside each other with a uniform interface. In both these solutions, the details required by one system are mapped to the other using a communications format that both systems understand. More recently, the EDGeS project is being extended by the SCI-BUS project [65], ultimately aiming to connect academic grids, volunteered computers, and commercial clouds.

1.2 Software Agents, Interoperability and Resource Allocation

Interoperability: Software agents are distributed applications which communicate and interact with each other and are a mainstream solution for interoperability [70, 130]. Agents are suited to this problem because they can be co-located with the heterogeneous resources which need to be made interoperable. Agents can be programmed to become 'wrappers' for the resource with which they are co-located, effectively becoming a layer above the underlying resources. Message interactions between agents can make the necessary translations to allow their underlying resources to work together.

Resource Allocation: Software agents also offer a solution for resource allocation [80, 70], modelling and executing the types of interactions needed for making resource allocation decisions. That is, agents can be used to represent the different entities which are involved in making such decisions, be those entities producers or consumers of computing resources.

1.2.1 Social Grid Agents

Social Grid Agents (SGAs) [107, 108] are an agent technology which were designed primarily for resource allocation. Their development is motivated by the diversity and complexity in Grid computing and the view that full interoperability using standardisation is difficult and unlikely in the near future.

SGAs have two features which are important in resource allocation and interoperability. First, they are designed to operate in a distributed computing environment and can be co-located with the services which produce and consume resources they allocate. Second, a co-located SGA can act as a 'wrapper' for one or more of these services, thereby providing interoperability between those services.

SGAs are implemented as web-applications and communicate by sending SOAP messages. Among other things, these messages can contain the resource allocation policies of the agent's owner. Work allocation decisions can be based on the price, available credit and the value of Grid services.

Social Grid Agents are agnostic to the types of resources they interact with or the types of resources they are used to allocate. The modular architecture is extensible, allowing the technology to be extended to encompass other types of resources, including cloud providers and desk-top grids.

SGAs and Classad Matching

The Classad match-making mechanism [109, 7] is commonly used to describe computer resources and work for matching and allocation. This versatile matching approach allows producers to describe the capabilities of their resources, and consumers to describe the requirements of their jobs. Attributes of both resources and work are expressed as strings, integers or 'functional expressions'. The matching process evaluates and matches attributes and requirements using Boolean matching; that is to say, requirements are either satisfied or not. This is a suitable mechanism where the attributes described are technical details, such as memory or CPU specifications. Classad is widely used as a matching approach in grid computing resource allocation [55, 85].

SGAs use Classad extensively, both for communications and expressing and matching resource allocation policies. However, Classad is not designed to describe the many *social* and *economic* differences found between the different types of computing resources and work described in section 1.1. These differences are subtle and nuanced and Social Grid Agents need an additional approach to complement the existing Boolean matching mechanism that they already use.

1.2.2 Changes in Resources and SGAs

Continuous change is an inherent feature of the work allocation environment. Arrival of new resources, removal of others, as well as changes and addition of new features to existing ones are all to be expected. Agents interacting with these resources also need to be able to deal with frequent change.

SGAs would greatly benefit from a mechanism which allows them to be quickly and easily created and deployed in the environment in which they are to be used. The capability to update the abilities configured in each agent would allow them to adapt to changes in their underlying services. In addition, configurability would make SGAs more usable and flexible. It would allow them to be altered and re-configured to address new or different tasks.

1.3 Determining Similarity Between Resources

Security of data and processing speed may be important for some consumers. Return on investment and profitability will be vital for a commercial resource provider. Knowing that their donated home PC is doing something of humanitarian value may be an absolute requirement for a 'computer volunteer'. Others volunteer their computer time to help find signs of extra-terrestrial life [8].

The challenge for anyone attempting to match producers and consumers of resources, where such a myriad of considerations may be at play cannot easily be addressed using Boolean matching. Matching all the possible combinations of different social and economic outlooks presents a complex problem; maybe these considerations can never be matched exactly. Perhaps the best that can be done is to determine what similarity there is between the different entities involved in resource allocation.

One interesting approach is simply describing, using free language, all the attributes of the entity which are relevant to its resource allocation needs. Using a textual description puts the full resources of the language used at the disposal of the person writing the description.

In producing a comprehensive textual description, search engine technology then offers what may be a useful line of attack on this problem. In language usage, entities which are similar are often described using the same words. Similarities between entities can therefore be measured by measuring similarity between the textual descriptions which describe them. Using the frequencies with which words occur in textual descriptions, search technology is one mechanism which provides such measurements.

1.4 Research Problem

In the context of using agents in resource allocation, two areas are examined in this thesis. Returning to the idea that some of the resources described in section 1.1 can be assembled to address a large computing challenge, both areas are examined with the aim of facilitating this happening in the future. These areas are:

- Enhancing the ease with which SGAs can be created, deployed and configured in the environment in which they are intended to be used.
- Integrating search technology into SGAs, thereby providing them with a mechanism to determine a measurement of similarity between the resources they allocate.

The motivation for enhancing SGAs in these ways is to enable the creation of a system of agents for deployment alongside producers and consumers of computing resources in their environment. These agents will represent and act on behalf of these entities. Each agent will have its own textual description of the entity it represents. Similarity between these descriptions can then be considered when making resource allocation decisions.

1.4.1 Research Aims and Questions

This thesis explores many aspects of the two areas set out above.

SGA Creation and Configurability

For ease of creation and configuration of agents, this thesis introduces a mechanism which allows SGAs to be quickly created and deployed in large numbers. In existing SGAs, agent abilities need to be hardcoded during development, each one having a fixed set of abilities. This renders it only suitable for its original purpose or task. Using the mechanism proposed in this thesis, each agent is initially created and run using a minimal set of 'hardcoded' abilities. They can then have functionality added dynamically at any time during their lifetime, making them extensible and much more flexible. Crucially, agent functionality can be added without needing to stop and rebuild an agent.

For scalability and configurability of SGA, the following questions are examined:

1. Can the same behaviour and functionality found in a hardcoded SGA be replicated in one that is created using the automated approach proposed in this thesis?

2. Is this method of creation proposed by this thesis better, more flexible, easier and faster that what preceded it?

3. Does the approach used make SGAs more usable?

4. Can the specification for an agent's abilities be shared and reused in many agents, giving rise to improved scalability and efficiencies?

5. How scalable and adaptable is the proposed mechanism in terms of adding abilities which need external libraries not previously used in an SGA? What are the benefits of being able to do this?

Integrating Search Technology into SGAs

This thesis also uses textual descriptions to describe the different social and economic attributes which the consumers and producers of resources may have. In this thesis, these 'social and economic' descriptions are called *social profiles*. To use these social profiles within a multiple-agent system, both the social profiles and the search functionality to exploit them must be integrated into SGAs. In examining this, the following questions are addressed (numbering continues from above):

6. Can search technology and its functionality be successfully integrated with the SGA distributed environment?

7. Can social profiles be easily transferred between SGAs? Can one SGA search for a social profile in another SGA?

8. Can similarity between textual descriptions be determined using search technology? Can this be done within an agent?

9. Are social profiles scalable in terms of how many different types of information they could contain?

10. Along with social and economic textual descriptions, what other information can be incorporated into social profiles?

1.5 Layout of this Thesis

This thesis is laid out as follows:

- Chapter 2 examines many different types of computing resources and work which may be potentially carried out by them.

- In chapter 3, software agents are examined. Social Grid Agents are further discussed and compared with some other agent technologies.

From this point, the thesis goes in two different directions, each looking at the two areas outlined in section 1.4.

- Chapter 4 introduces a mechanism for creating and configuring SGAs. Scalability, flexibility and configurability improvements are successfully made to Social Grid Agents. Some serialisation and class loading techniques, new to SGAs, are fruitfully employed to achieve this. Creation of multiple agents is transformed from using a manual 'hardcoded' procedure to being semi-automated. Systems created can be increased or reduced in size and abilities can be added, all without having to disrupt or stop the agents.

- Chapter 5 develops the idea of a 'social profile' for describing social and economic differences. Search technology is also successfully integrated into SGAs and tested in a distributed environment. Quantitative similarity measurements between social profile descriptions are explored, as well as scalability and extensibility of the format used for describing the entities.

- The work of chapter 4 and 5 are brought together again in the experiments of chapter 6. This chapter examines the developments of the thesis to determine the feasibility (or not) of what is proposed. Results are presented, and these are complemented with some further discussions about some of the problems which were encountered during developments.

- The thesis concludes with chapter 7, which sums up the findings of chapter 6 and others. An evaluation of this work is presented, pointing to future work, and discusses some limitations of the approaches used.

1.6 Publications from this Thesis

- Work allocations governed by social profiles for large scale heterogeneous infrastructure. Peter Lavin and Brian Coghlan. *In Information Society (i-Society), 2011 International Conference*, IEEE, June 2011 [87].

- Computing resource and work allocations using social profiles. Peter Lavin, Eamonn Kenny and Brian Coghlan. *Computer Science Journal*, 14(2):273293, January 2013 [89].

- Dynamic proliferation of agents in a Multiple-Agent system. Peter Lavin and Brian Coghlan. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference*, IEEE, February 2013 [88].

Chapter 2

Computational Resources

2.1 Introduction

This chapter describes a number of different types of computer resources. Academic and scientific research, commercial and volunteered systems are examined. As well as the technological differences between them, some of the implications of their different origins and backgrounds are also discussed. Interoperability between these systems, their typical workloads and how these are allocated is also reviewed.

2.2 Academic and Scientific Research Grids

Individual universities and scientific institutions often invest in substantial amounts of computing resources for their own research and teaching needs. Although these resources exist within controlled environments (i.e. not directly connected to the public Internet), any single academic institution can agree to aggregate and share its resources with other similar institutes which share its aims. More ambitiously, multiple institutions can aggregate resources using wide-area constructs called Grids.

A *Grid* is a internet-scale distributed computing infrastructure consisting of connected computing resources which are distributed across multiple organisations and administrative domains. The resources are then provided to scientific and academic researchers across these institutions. Typical Grids which have been created for scientific research include the European Grid Infrastructure (EGI) [1], TeraGrid [9] and the Open Science Grid [10].

At a somewhat abstract level, Foster sets out a three point checklist [68] which defines a Grid as a system that:

1. "coordinates resources that are not subject to centralized control"
2. "uses standard, open, general-purpose protocols and interfaces"
3. "delivers nontrivial qualities of service"

At the implementation level, a layer of middleware is used to connect these distributed and heterogeneous systems. From the perspective of an end-user, this middleware presents a standard user interface (or UI) to all users so that jobs can be processed while hiding possible differences in the various systems which make up the collective grid resource.

Grid middlewares can also be thought of as being the 'building-blocks' of grid systems. They provides resource-sharing arrangements allowing the resulting resource pool to be used across several organisations and domains in a secure and stable manner.

2.2.1 An Example Grid and Middleware – EGI and gLite

The EGI infrastructure consists of a number of distributed sites, a typical site being a university campus or scientific research center. All sites included in EGI provide some amount of computing resources, with more extensive sites also providing storage. Storage, consisting of persistent disk and tape storage, may be used to hold files which are necessary to process work or to store the resulting data from that work.

Typically, EGI resources are used to process large bodies of work which have the characteristic of being able to be broken down into individual 'jobs'. These jobs can then be processed in parallel, simultaneously and independent of each other. Depending on the nature of the work, the duration of these jobs may vary from a number of seconds to a number of hours or days.

Both computing and storage resources at every site regularly publish information about their usage and availability to a *Resource Berkeley Database Information Index* which in turn is published to a *Top Level* BDII known as the Grid Information Index Service BDII. This hierarchical system provides information on the overall state of the Grid and also supports service and resource discovery.

Grid Compute Elements

The resources at each EGI site are known as a Compute Element (CE). The CE is made up of a Gatekeeper (or bridge to other sites), a batch management system, a quantity of compute nodes (i.e. the machines that do the actual work) and storage nodes if present. The compute nodes may consist of a number of hardware and virtual machines, or may include some 'specialty hardware' (discussed in sub-section 2.2.2). Within the CE, individual jobs are scheduled, matched and allocated to a suitable CE by a Workload Management System (WMS). Although jobs may be processed independent of each other, EGI also provides a message passing interface between CEs where jobs may need to communicate with other jobs [124].

Grid Storage

EGI provides a distributed storage system for files required for running jobs. Such files are copied from a UI to a Grid Storage Element (SE) where they are registered and given a unique identifier. Files are replicated for redundancy and access optimisation. File permissions can be specified in a similar way to those on a standard file system and these can be used for controlling access to files by different users.

Authentication in Grid Computing

Trust is established between users and participating institutions by Certification Authorities (CAs). CAs work to an agreed set of standards for establishing the identity and bone-fides of a prospective grid end-user. An electronic identity is issued in the form of a signed *X.509 Grid Certificate* which contains a public and private key. When using Grid resources, this certificate allows the user to create a 'proxy' certificate (which is also signed using the CA signature). This 'signed' proxy certificate is then used as proof of identity when accessing Grid resources. For security reasons proxy certificates are valid for a much shorter period than Grid certificates.

Authorisation in Grid Computing Using Virtual Organisations

In grid computing, the *authority* to use aggregated resources is controlled by grid administrators on behalf of, and by agreement with the resource providers. This is done by creating and managing what is known as *Virtual Organisations* (VOs). A group of individual users who are defined by a common set of sharing rules and conditions constitute a VO. Such a group is typically made up of users who have some commonality in their work or research, but which may be distributed across several locations. VO membership is also used to control access to storage resources, including down to the level of access to individual files stored on an SE. The sharing rules are implemented by a component known as the VO Management System (VOMS).

User Submission of Grid Jobs using gLite

gLite [11] (pronounced "gee-lite") is the middleware of the European Grid Initiative (EGI). From the perspective of an end-users, the middleware provides access that grid resource via a command line interface. Primarily, it allows users to submit individual jobs and subsequently retrieve the results. Among other things, it can also be used to discovery available resources and check the status of a job, checking for example if a job is scheduled for processing on a compute node, already in progress or already completed. EGI resources can also be accessed using a *Grid Portal* [49, 12]. These graphical interfaces effectively wrap an underlying gLite UI and facilitate more pervasive and easier access [91]. Some sample gLite commands with explanatory comments are shown in appendix A on page 196.

Before a job may be submitted, it must be described in terms which are useful to the scheduling mechanism in the WMS. Jobs are described using a text file known as a Job Description Language or JDL file (discussed also subsection 2.7.2 on page 35). Details in this JDL file are used by the WMS for matching and scheduling jobs to the available resources. Figure 2.1 shows a simple JDL file which describes a grid job. Within the WMS, a file of similar format is used to describe each CE.

```
[
  Type = "job";
  JobType = "Normal";
  Executable = "GridJobScript.sh";

  # The arguments which are passed to the executable
  Arguments = "dataFile01.txt arg0 arg1 arg2";

  # Specifies files which collect outputs during execution
  StdOutput = "sim.out";
  StdError = "sim.err";

  # Files which will be submitted with the job
  InputSandbox = {"GridJobScript.sh", "dataFile01.txt"};

  # Files which will be retrieved with the results
  OutputSandbox = {"sim.err","sim.out"};

  # A system CPU speed of 2600 MHz is required.
  Requirements=(other.GlueCEInfoCPUSpeed>2600);

  # If more than one resource matches,
  # the resource with the fastest CPU is chosen
  Rank = (other.GlueCEInfoCPUSpeed);
]
```

Figure 2.1: A simple Job Description Language (JDL) file.

Figure 2.2 outlines a simple grid overview, showing multiple users interacting with a grid using a number of UIs. It also shows the layout of the UI, WMS, CE and SE components described above. Using the commands shown in appendix A and these components, the sequence for carrying out work on EGI Grid resources can be

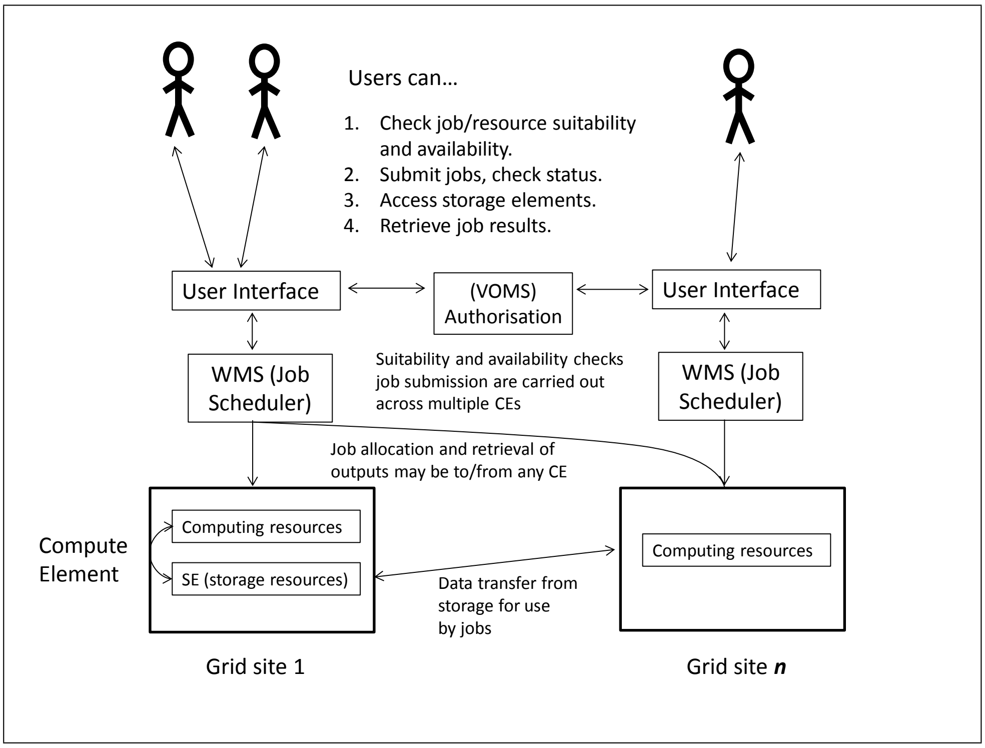

Users can…
1. Check job/resource suitability and availability.
2. Submit jobs, check status.
3. Access storage elements.
4. Retrieve job results.
User Interface
(VOMS) Authorisation
User Interface
WMS (Job Scheduler)
Suitability and availability checks job submission are carried out across multiple CEs
WMS (Job Scheduler)
Job allocation and retrieval of outputs may be to/from any CE
Compute Element
Computing resources
SE (storage resources)
Computing resources
Data transfer from storage for use by jobs
Grid site 1
Grid site *n*

Figure 2.2: A Simplified Grid Overview.

summarised as follows:

1. The body of work prepared, breaking its data files into small individual parts for storage on the UI machine, or if they are large, on a suitable SE. A job essentially consists of a set of one or more data files.

2. A individual JDL file is prepared for each job.

3. The job is submitted (by referencing its JDL file) on the UI or portal machine.

4. The matching and scheduling of a given job to a compute node is carried out by the WMS, this may be on local resources or on a CE at another site.

5. During and after this time, the status of the job may be monitored until successfully completed.

6. The result of the job is then retrieved by the user. Results may also transferred to grid storage and retrieved later.

7. For large numbers of grid jobs, the entire process may be automated.

2.2.2 Specialty Grids

Specialty Grids are grids (or part thereof) which have special form and function due to the type of hardware used or other features of services provided. This sets them apart from the conventional systems described earlier in this chapter. For example, with processor speed increases slowing down due to cooling and other physical restrictions [60], the trend towards multiple core processors and specialised processing units for increasing cycle throughput has become more established. Grid researchers are keen to exploit 'many-core' processors (some of which are available in commodity computing equipment). Some of these are briefly reviewed here to indicate the potential complexity which may be involved in integrating them with other systems.

General-Purpose Computing on Graphics Processing Units (GPGPUs)

Graphics Processing Units, sometimes called visual processing units (GPUs or VPUs) are specialised processors designed specifically to accelerate the generation of images for display on computer monitors. Graphic computation involves highly parallel vector and matrix calculations. The increasing popularity of computer games has driven many improvements in this area and this has led scientists to study to use of GPUs for general-purpose (non-graphical) computing. GPUs may be integrated in to the system motherboard or may be located in dedicated graphics cards. Where there is more than one GPU fitted, these can be exploited in parallel. Compute Unified Device Architecture (CUDA) [71] and Open Computing Language (OpenCL) [83] are examples of APIs used for programming and access to GPUs.

Commodity Game Console Grids

The Cell Broadband Engine was developed by Toshiba, IBM and Sony, the latter using it in the PlayStation III (PS3) game console. Based on PowerPC architecture, these game consoles originally had an open system configuration which permitted the Linux operating system to be installed. This allowed the game console to be also used as a personal computer. This in turn spawned several developments of programs which were designed specifically to exploit the multiple processors available in these systems. Some of these were unrelated to gaming and graphics generation. One such program is eHiTS (electronic High Throughput Screening) from Symbiosys [13] which exploits the multiple processor architecture to speed up the task of screening potential drug compounds against known disease proteins.

Specialty Systems and Middleware

For these specialty systems to become part of a grid resource, the grid middleware must be adopted (or 'ported') to suit the operating systems and hardware platforms on which they run. Although they may be fundamentally different types of resources to others in a given grid, being able to run the same middleware means that they are presented to a user as a 'grid computing resource'.

2.2.3 Condor and HT Condor

HTCondor is a cycle-harvesting system designed to support high-throughput computing [117]. Within the confines of a university or institution, it is used to integrate dedicated clusters of nodes (often associated with grid computing) and non-dedicated desk-top machines. For the non-dedicated machines, the owners of individual machines allow HT Condor to launch an externally submitted job when the machine becomes idle. A collection of available machines is termed a *HT Condor Pool* and under the HTCondor model, can be considered a single computing resource.

When the HT Condor pool manager detects that a machine has become idle (i.e. no keyboard activity or low CPU usage), it takes an unexecuted job from a queue it maintains and assigns it to the idle machine for execution. If keyboard activity restarts, HT Condor detects this activity and *evicts* the external job. Thus, machine owners maintain exclusive access to their own resources and HTCondor only uses them when they would be otherwise idle.

Middleware and Job Submission

Similar to gLite and EGI resources, HTCondor is essentially middleware which is installed on the resources which it harvests. Although not generally as large in scale as, for example EGI resources, the process of submitting and matching jobs to resources using a UI is largely the same. A fundamental difference between EGI and HTCondor resources is that some of the CPUs used by HTCondor are not dedicated and used exclusively for processing jobs grid.

2.2.4 Academic Resources Summary

Funding for most academic resources comes from national or international bodies. This brings with it a level of accountability which demands that resources be used for tasks which are mainly of public interest and benefit. It could also be said that the security arrangements (which are there to protect the resources and all users) have become a barrier to entry. It is perhaps understandable that, where institutions share valuable and expensive resources, an agreed regime with a high level of security, authorisation and authentication will prevail.

Regarding specialty grids, and the GPUs and game consoles they exploit, it was perhaps inevitable that the high capacity processors found in these would be put to uses other than those originally intended, especially in academic research. Later, when looking at public resourced computing (section 2.4), we will see how similar equipment has been also used as a resource which has a somewhat different social and economic background.

2.3 Cloud: Commercially Available Resources

Cloud computing is the delivery of available computing resources using network connectivity [48]. This type of service allows a consumer to pay for resources from an external provider company on as 'as needed basis'. In commerce, consumers are usually businesses who need computing capacity and who are prepared to pay commercial rates for it [48]. However, Cloud Computing has become a popular way to meet capacity needs in other areas such as research and teaching also.

Using Cloud instead of a privately owned resources is somewhat analogous to how electricity was in the past and is currently provided to industry. Having moved away from privately owned generating equipment, today consumers only buy and use the amount of electricity they need from a national power system. Cloud providers aim to replicate this shift in computing capacity provision by offering compute power, bandwidth and storage on demand.

Services are provided in a number of models, each one aimed at fulfilling the different needs of consumers. *Infrastructure as a Service, IaaS:* Infrastructure (hardware) is provided for users to create one or more virtual machines (VMs). It is usual that the consumer provides the VM image, thereby controlling what operating system (OS) and software runs on them. Amazon's EC2 [2, 3] is a typical IaaS product, although there are now several.

Platform as a Service, PaaS: This product provides a fixed platform, where the consumer has little or no input into what OS is provided. It is more suited to users who do not have the proficiency to create VM images, and to interact with the virtualisation mechanisms of an IaaS offering. Google App Engine [14] is an example of a commercial PaaS offering.

Software as a Service, SaaS: This type of service provides software which processes, for example, large batches of uniform jobs. The user is provided with access to software without having to install and configure it. The product offered can scale up and down to meet the demand from consumers. Examples are Google Apps [15] and Microsoft Office 365 [16].

Network as a Service, NaaS: Although not strictly in the same categories as other cloud services, networking is an integral part of cloud computing. The ability to purchase network connectivity as required may be used in conjunction with other IaaS or PaaS needs. NaaS services may be used to cope with increases in network traffic due to business cycles or to create ad hoc networks for short term collaborations. Providers may also offer support for virtual private networks (or VPNs) to connect the 'last mile' of the required network. This is an evolving area and *Bandwidth-on-demand* [123] and *OpenNaaS* [17] are examples of developing technologies.

2.3.1 Hybrid Clouds

Flexibility is a common theme in cloud computing. Hybrid clouds have evolved to meet the needs of large consumers who already hold some existing ‘privately owned’ resources and want the flexibility to scale up their capacity by expanding ‘into a cloud’ when demand spikes.

2.3.2 Cloud: Advantages and Concerns

Armbrust [48] outlines three aspects of Cloud computing which make it an attractive proposition for potential consumers.

1. There is the ‘welcome’ illusion of infinite computer resources available on demand, removing the risk of over or under provisioning for anticipated demand.

2. There is no need to make up-front financial commitments for provisioning of resources. These can simply be hired and paid for as needed and then relinquished.

3. From a business perspective, using increasing computing resources can be provided and paid for from ‘operational costs’ as opposed to using a capital investment.

These are immediately visible benefits. Further benefits accrue from avoiding costs of additional staff and facility upgrades.

However, there are some concerns associated with using Cloud Computing. Users may be reticent to co-host their data on infrastructure which is alongside potentially competitive data and applications. Data may also be stored in different jurisdictions where different laws on privacy and national security apply. Furthermore, each provider offers a different and sometimes unique way to interface with their resources. Hence, applications and data may be customised for a particular vendor. This makes moving to a different vendor difficult and expensive, something termed ‘technology lock-in’.

2.3.3 Social and Economic Considerations of Clouds

Cloud Computing is offered as a commercial service. While keeping their usage rates competitive in the market, each vendor is also aiming to make a profit and a return for their investment. This is in contrast to most Grid resources which are largely paid for by government and used by scientists and researchers in institutions which are also (directly or indirectly) funded by governments.

While the technology offered is important, providers are likely to strive to accommodate the needs of the market for their products. This will come naturally as providers compete for consumers.

Over time, these differences may fade as Cloud providers offer pro bono resources for academic and pre-competitive purposes and as Grids begin to use clouds on a large scale as back-end resources. This may lead to the evolution of grid-like federations of Clouds. Grids may also adopt charging as a cost recovery mechanism to assist sustainability.

2.4 Public Resourced Computing

Public Resourced Computing (PRC) or Volunteer Computing is a popular internet-wide phenomenon. It revolves around members of the public donating their computing resources, often home PCs, to support research projects which appeal to them [46]. Also knows as *desk-top grids*, many individuals' computers, all working in parallel, carry out small portions of a large body of work. The resources needed for such work are often measured in hundreds or thousands of CPU years.

The concept of publicly resourced computing was first implemented circa 1995 in the Great Internet Mersenne Prime Search (GIMPS) project [18]; in that initiative, volunteered resources were used successfully to search for new prime numbers. Today, volunteered computing resources are used in a broad range of scientific research areas including biology, climate change prediction, earthquake predictions and physics. Work for projects may come from an ongoing stream of data such as that emanating from the space telescope observations of the SETI project [8] or from a bounded parameter sweep application with a defined number of tasks and expectations around required computing time.

Connectivity between the many volunteers and a projects is provided by middleware. Although there have been a number of developments in this area, the most popular and enduring is the *Berkeley Open Infrastructure for Network Computing* or BOINC [19, 45] from Berkeley UC in the USA.

2.4.1 BOINC

BOINC is an open source middleware used widely in public resourced computing. Similar to grid computing, a workload is suitable for public resourced computing if it can be broken down in to units of work which can be processed individually and in parallel. However, for a given BOINC project, these must be homogeneous in nature.

The terminology used in BOINC pertinent to jobs, work and results is different to the grid environment.

- An individual task from a large body of work is called a *workunit* or *WU*. This is analogous to a *job* in Grid computing.

- A *WU* may be duplicated and computed more than once by a BOINC project and each instance of this duplication is called a *result* (before and after it is computed). The status of a result will change when it is sent to a volunteer node and when it is computed successfully or otherwise.

- A result is complete when it has been computed successfully on a volunteer node and its outcome is returned to the central server (see below).

- A workunit is only complete when enough computed results are returned to allow a canonical and therefore reliable outcome to be established using a voting algorithm.

2.4.2 BOINC Technical Details

The BOINC Server and Client

The center of a project is a computer running the BOINC Server software. For larger projects, a number of central computers may be used. The project administrators must develop a custom application for distribution to volunteered nodes in order to process these workunits. A reasonable level of programming proficiency is required to prepare a BOINC server for use in a project.

Volunteer Node Activity

On joining a project, a volunteer user downloads the BOINC client program and the customised application of the project from the BOINC website [19]. On installation, this program contacts the project server. The volunteer registers with the project and downloads a quantity of workunits. A number of workunits are downloaded for processing. This allows the node to continue processing results even when disconnected from the internet. These results can be uploaded to the project server when next reconnected. In the case of an 'always on' internet connection, each result is processed and the outcome is uploaded without delay to the project BOINC server. Otherwise, processed results are stored on the client machine until next connected to the BOINC server. In general, only a small amount of storage is used on each client node.

Because of its evolution path, Public Resourced Computing does not implement shared storage [119]. PRC was designed for distributed parallel computations where the ratio of required data transfer to CPU hours is small [69] and where continuous connectivity between a central server and its clients is not always guaranteed. The features of the BOINC architecture which are shaped by this are a legacy from a time when 'always on' internet connectivity was not as common as it is today.

2.4.3 Ownership and Control

The nodes on which work is carried are neither *owned* nor *controlled* by the project administrators. Ownership and control of worker nodes ultimately stays with the volunteer owner. The installed client program allows the level of usage on the volunteered user's system to be configured. Disk storage allocated to the BOINC client program, restrictions on usage time and percentage of CPU load are all user configurable.

2.4.4 Trust, Security and Malicious Behavior

Trust is asymmetric insofar as volunteers must place considerable trust in the originators of the project and their software. From the perspective of a project administrator, volunteered resources are generally considered untrusted. Malicious behavior is considered an unavoidable part of the PRC environment and members of the public can engage in such activity with impunity. This attribute of the PRC environment is the primary reason that results are not accepted until a number of identical outcomes are 'agreed' upon in a vote. Malevolent behavior by the public ranges from falsification of results to attempting to replace a project application with software detrimental to volunteer's systems. There are several features in BOINC middleware to counteract this type of activity on client worker-nodes [116] and the BOINC developers at UC Berkeley advocate full use of these.

All stakeholders in BOINC projects rely on the middleware having a strong reputation for safety, ease of use and reliability. The negative implications for a BOINC project which has become harmful to volunteers' systems extend beyond that project and may also impact negatively on other (current and future) BOINC projects.

Private uses of BOINC

On a smaller scale, but using the same BOINC middleware, computing pools consisting of tens or hundreds of desktop computers can also be harnessed in private institutions. This is a similar resource to HTCondor (section 2.2.3) where the utility of idle computers is harvested when available.

2.4.5 World Community Grid

An interesting cross-over between the commercial world and public resourced computing is World Community Grid (WCG) [47, 20]. In this IBM initiative [21], the BOINC (2.4.1) volunteer computing middleware is used to aggregate volunteered resources. WCG takes a close interest in the type of project it allocates resources to and vets project aims as well as application code used.

Every project which benefit from using WCG resources could be created as individual project. However, the aggregating effect provided by WCG, coupled with the additional technical input makes this initiative a more reliable, prolific and centralised source of publicly resourced computing. Another benefit of this paradigm is that private users who are willing to donate their spare CPU cycles are more likely to receive a constant supply of work by joining WCG.

In April 2013, WCG website statistics show 615,523 members providing over 2.2 million devices. In keeping with the social and economic consideration of prospective volunteers, only projects which are *not for profit* and which have potential to bring a *humanitarian benefit* are allocated volunteered resources by the WCG administration committee.

2.4.6 Social and Economic Considerations in PRC

Social and economic considerations are important in PRC. The identity of the ultimate beneficiary of results obtained is likely to be an important consideration to individuals when deciding to donate resources. This means that pro bono research projects are the principal beneficiaries of volunteered computer resources.

The volunteered resources often have a high value but are available at a relatively low cost and the benefits go primarily to the project initiators or owners. Notwithstanding this, the project initiators are dependent on the good will of the public to provide resources. Volunteers must be attracted and retained and this is done using publicity, forums and various other means.

Donated CPU time to a given project is acknowledged only in the form of non-monetary ‘BOINC Credit’. This is allocated to volunteers proportional to the amount of work done on their system. Although completely symbolic, credit allocations are publicly known and appear in league tables on various websites. They appear to act as a stimulus or motivation among communities of volunteers. For example, it is common for volunteers to include their credit data in BOINC related emails and forums.

Technical Considerations

BOINC project developers must cater for diversity and heterogeneity of the computer architecture and operating systems abroad in their potential user pool. Because users cannot be expected to be 'expert' in the applications used, the middleware must be easy to install and use. It must also be as unobtrusive to the resource owner as possible. The volunteer often has very few programming or system administration skills. This means that problems with the client program, application and workunits must be self-healing. Failure (for whatever reason) to process a result sent to a client must be handled gracefully by the client program and server.

2.4.7 XtremWeb

XtremWeb [66, 58] is a peer to peer (P2P) multi-application Internet desktop middleware which permits public resourced computing. It is a P2P system in the sense that every user can not only provide a computing resource but also utilise computing resources provided by others. It differs from BOINC insofar as all participants are able to submit new applications and tasks to the system. XtremWeb is implemented in Java$^{\mathrm{TM}}$.

XtremWeb is organised as a three tier architecture and, like BOINC, but unlike grid computing, it does not use a shared filesystem. Clients are nodes which consume resources. Workers provide these resources and use a *pull* model to obtain tasks. A *coordinator* is the central agent for a local instance of XtremWeb. It manages the system, scheduling and fault tolerance of tasks and workers. The various tasks submitted may differ in nature and in application requirements (as opposed to the homogeneous nature of the tasks or work-units computed within a given BOINC project). In that regard, XtremWeb is similar to Condor [93, 113].

Security and Trust in XtremWeb

Because each user can submit applications (in contrast to BOINC), security and trust requirements are somewhat symmetric. Therefore there cannot be any form of implied or naive trust between users, applications, results or even the coordinator. Security is based on autonomous mechanisms which aim to protect each element of the system from other elements. For example, sand-boxing is used to protect from malicious code, and public and private keys are used to authenticate the coordinator when results of work done are being uploaded. Application code downloading is secured and verified using public-private keys and checksums. Later developments in XtremWeb [58] address the convergence of XtremWeb and grid computing.

2.5 Interoperability Between Resources

In the earlier sections of this chapter, we have looked at a number of different types of computing resources. Interoperability between these systems is another part of the overall picture motivating the research of this thesis. Allocation of work to different resources means that they must be able to interact and interoperate at a technical level. This section looks at the 'nuts and bolts' of how this may be achieved, and what inroads have been made in this area of research.

2.5.1 Interoperability of Grid and Volunteered Resources

Many approaches have been explored in the area of interoperability between various types of resources. Examples are the Lattice, EDGeS and SCI-BUS projects, which each address bridging technology gaps between Grid infrastructure, volunteer middlewares and other types of resources. A common problem encountered in this area is the matching and mapping of messages passed between interacting programs; all of these solutions look at translation solutions for interoperability between systems.

2.5.2 The EDGeS Project

The EDGeS project [22] has developed bridges to link EGI (2.2) to two volunteer or desktop grids which have been created using BOINC (2.4.1) and Xtremweb (2.4.7) middlewares. It aims to run jobs on either EGI Grid or desktop resources in a seamless way.

Desktop and Grid Interoperability in EDGeS: To allow BOINC jobs to be computed on EGI resources, a BOINC client is modified so it appears as a very powerful computer to its server. The modified client does not run jobs pulled from the BOINC server, but instead transfers the data and executable to an EGI UI. Using gLite commands, a wrapper script is used by the client to submit and collect the output of the jobs at the grid UI, from where it is returned to the BOINC client. The disadvantage of this method is that the modified client takes no account of the availability of EGI resources when pulling work from the BOINC server. The BOINC client can also become a single point of failure and become a bottleneck when the quantity of Grid resources available increases [82].

Executing EGI Jobs on Volunteered Resources: In order to compute jobs submitted to EGI on a Desktop grid, the BOINC and XtremWeb (server) resources are presented as an EGI Compute Element. An application repository is used which contains only applications trusted by volunteer nodes. Only jobs which require one of these applications can be submitted to the Desktop grid. A plugin (called *Target Grid Plugin*) appropriate to each type of Desktop grid is used and is responsible for querying job (or workunit) status and retrieval of the output.

Generic Grid to Grid Bridge: Multiple customised bridges are required for interoperability between EGI, BOINC and XtremWeb. The EDGeS project have developed a *Generic Grid to Grid* bridge (G3) which can be used for each case [122]. It contains components which store generic descriptive information on jobs, the application repository and the *Target Grid Plugin* mentioned above.

2.5.3 The Lattice Project

The Lattice Project of Maryland UC [100, 6] has integrated and deployed a range of computing resources on their campus for use in life science applications. Although this project involves several technologies, it uses the Globus Toolkit 4 [23] to handle job management, data movement, and security.

The Lattice Project has developed an interface with BOINC which is presented as a Globus addressable resource. Globus provides a standard web interface called the Globus Resource Allocation Manager (GRAM) which is used for job submission. In effect, the BOINC server appears as a GRAM compliant Globus node. This allows a client to invoke a Grid service with a typical set of arguments (e.g. an executable, data files and paths and names of output files). Jobs are then in turn submitted to a Grid scheduler.

Running Jobs on BOINC in Lattice

Depending on the resource specification, availability and scheduling, the job may be sent to a BOINC project. In this case, the client arguments are mapped to requirements within the BOINC project using the Lattice interface. For example, the executable may be mapped to the application in the BOINC project. Note here that no user code is defined by the creator of the job. Instead, all code which will be used as an application on BOINC resources is contained within the BOINC project and is subject to the security constraints therein.

As Grid resources and BOINC have evolved in very different paradigms, other translations or mappings also need to be addressed. Memory and storage allocation mappings are straightforward. Features within Globus are used to convert maximum CPU usage in minutes (the units used by Globus) to estimated floating point operations (used in BOINC). Specific file staging semantics are also addressed.

It can be seen from both the examples of EDGeS and Lattice projects that interoperability often requires customised developments. Mapping of variables is crucial, but once this is achieved, computing capabilities of the underlying systems are able to process each others' workloads.

2.6 Different Types of Candidate Computing Work

The possibility of interoperability and convergence, and the enhanced resources this provides, has led to the emergence of new and increasingly ambitious computing challenges. Projects which were previously intractable (due to their size) are now becoming more feasible. This section looks at the various types of work which are already processed by the existing resources discussed so far in this chapter.

2.6.1 Academically-Resourced Work

Typically carried out in universities and research institutions, this type of work is often compute intensive and may also have demanding storage requirements. Whereas some large groups maintain their own computing resources, many cannot afford to. Consequently, each group will target whatever aggregated resources best match their social and economic needs.

Projects which have predictable timeframes, like matching new drugs to disease proteins, are commonly performed on aggregated resources within national infrastructure. The WISDOM project [77, 54], searching for new drugs to cure Malaria is one such example.

More demanding workloads are performed within large international collaborations on international infrastructure. Experiments like ATLAS on the Large Hadron Collider (LHC) at CERN [44, 24] produce workloads that place demands on academic grid resources worldwide. For example, from 2007 to 2008, the Irish National Grid Initiative [25] provided over 70 million CPU hours to support this workload.

2.6.2 Commercially-Driven Work

As Cloud computing has evolved almost specifically to meet the demands of businesses, workloads like business processes, storage and web hosting are typical tasks which are addressed by commercially provided resources.

The ‘on-demand’ and scalable attributes of cloud resources make them attractive solutions for short term demand peaks such as end-of-year financial data processing in business. Similarly, services which support demand-driven media, for example coverage of popular sports events, are also common cloud consumers. The flexibility offered allows them to meet demand which is not level-loaded over time.

Cloud computing also offers secure storage. Due to the distributed nature of the resource, storage for commercially valuable data can be offered with the promise of multi-locational backups.

2.6.3 Publicly-Resourced Work

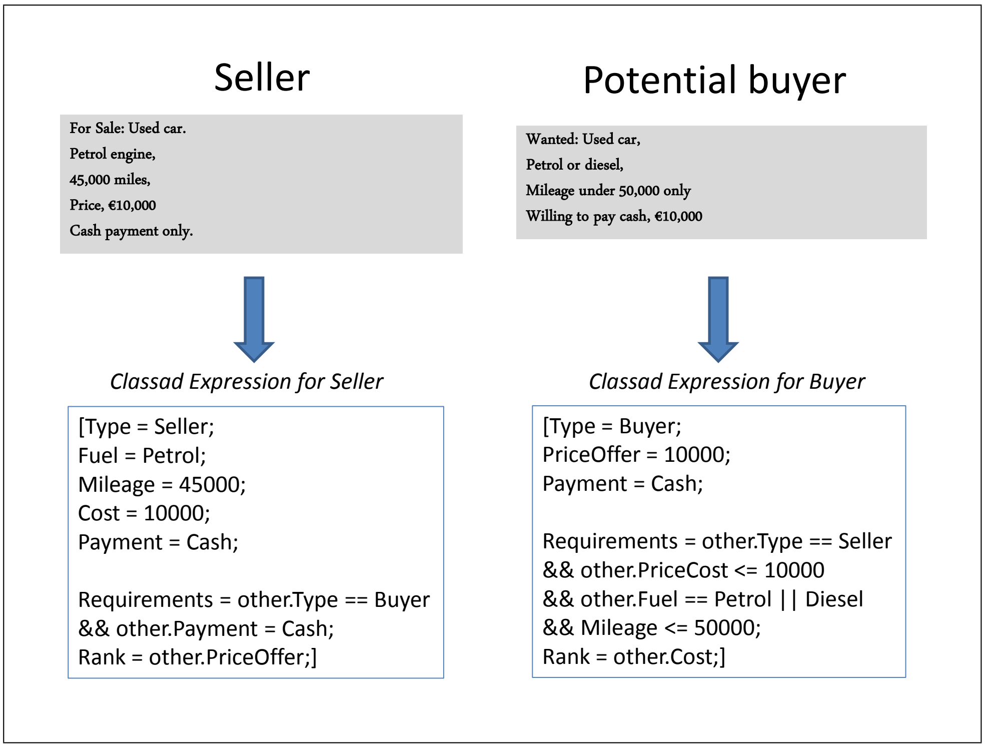
Work which is processed on computing resources which is volunteered by members of the public is often of a humanitarian nature or is considered (by the volunteers) to be beneficial to some sector of society in general. Scientific and medical initiatives, whose chances of success can be described as 'long-shot' does not readily attract funding from commercial companies. Such work is typically attracted to volunteered resources. Searching for drug related solutions to lesser known diseases or those prevalent only in under-developed countries are examples of this type of work.

However, the range of work which finds its way onto volunteered resources is wide. BOINC originally evolved from a the SETI project [8]. This project analyses radiowaves from outerspace for signs of extra-terrestrial signals or signs of life. Maths and science projects also attract a lot of public resources. The Distributed Rainbow Table project [26], for example aims to provide tools for detecting weak hashes from computer passwords.

2.7 Work Allocation

2.7.1 The Classad Language

Classad (an abridgment of Classified Advertisements) is a functional language which was developed in the Condor Project (2.2.3). It is now an important language in distributed computing, especially where resource allocation and matching is needed. An 'advertisement' (also known as a *Classad*) is the working unit of the language. Classads are analogous to the 'for sale' and 'wanted' classified adverts seen in newspapers. Providers specify what they have to offer and requesters specify what they want to procure. For work allocation, an advertisement consists of one or more uniquely named *attribute* and *expression* pairs (or simply expressions), each of which describes properties and requirements of resources and jobs. Both parties list requirements (or constraints) that need to be satisfied, allowing both providers and requesters to express constraints on each other. For illustration, figure 2.3 shows an example of two simple 'used car' classified advertisements and their equivalent Classads which express both what is offered and required by both parties.



For Sale: Used car.
Petrol engine,
45,000 miles,
Price, €10,000
Cash payment only.

Wanted: Used car,
Petrol or diesel,
Mileage under 50,000 only
Willing to pay cash, €10,000

```
[Type = Seller;
Fuel = Petrol;
Mileage = 45000;
Cost = 10000;
Payment = Cash;

Requirements = other.Type == Buyer
&& other.Payment = Cash;
Rank = other.PriceOffer;]
```

```
[Type = Buyer;
PriceOffer = 10000;
Payment = Cash;

Requirements = other.Type == Seller
&& other.PriceCost <= 10000
&& other.Fuel == Petrol || Diesel
&& Mileage <= 50000;
Rank = other.Cost;]
```

Figure 2.3: Simple used car advertisements with equivalent Classad expressions.

The Classad language matches advertisements in a pairwise fashion. The output of this matching process can be either one of TRUE, FALSE, UNDEFINED or ERROR. The undefined result is the outcome of an attempt to match two ads where an attribute is defined in one message but missing in the other. An ERROR outcome may be (among other things) the result of an incorrectly defined attribute. Where there are a number of potential matches, these can be ranked based on values in one or more attributes. The attribute to be ranked is specified by the requester; for example, in figure 2.3, the buyer may look to rank matches in order of ascending mileage.

Classad does not have a fixed schema and attributes can be given any name. The number of attribute and expression pairs is extensible. For matching, the required attributes are included in expressions as *Requirements* along with an attribute which is to be *ranked*. Classads from both providers and requesters should include these two expressions [110]. For other examples of Classad expressions, see appendix N, page 219.

Generality of Classad: Classad can be used as a *general* matching mechanism and is not bound to any type of resources or services being described. Attribute-expression pairs can be used to describe anything which can be described using literals such as strings, integers, etc. This freedom has allowed Classad to be used in other areas, for example in the development of Social Grid Agents (see chapter 3), and within a large number of grid projects [55].

Classad, when taken in toto, is a more versatile and richer language than its use in matching key pairs suggests. Classad expressions can also be used to contain 'payload' text as content. It can also be structured to take on the semantic-content form as seen in some agent communication languages [67, 63]. In the context of using Classad as an agent language, these are discussed further in sections 3.1.4 and 3.3.

2.7.2 Resource Allocation in Grids

The current allocation approach used for grid resources uses mainly Boolean matching and ranking is provided for matches where there are more than one criterion.

In the context of sharing and aggregating grid resources, CPU and storage quotas are agreed between grid site managers. These are then encoded in rules within the Workload Management System (WMS) in Grid infrastructure (see figure 2.2). Personnel representing virtual organisations may request changes in quotas as needs arise. National level operations centers can make final decisions on these allocations. These decisions will be influenced by contractual obligations between institutions, suitability of work and the original reasons behind the provisioning of the resources.

As a further example, SweGrid (the Swedish National Grid Initiative) and its subsidiary SNAC (Swedish National Allocations Committee) is a committee which meets to evaluate and make decisions on large allocation of the resources which it controls. In summary, some criteria used are... "The evaluation is based on scientific merit, need for the resources, feasibility of efficient use of the requested resources, and impact" [27]. The outcome of this is added an automatic fair-share allocation system within their infrastructure.

Job Description Language

At the level of allocating individual grid jobs to actual grid nodes, the European Grid Infrastructure (EGI) (section 2.7.1) employs a *job description language* (JDL) and a resource broker based on the Classad matching tool. This approach derives from the EU DataGrid project [28], further refined in the EGEE, EGEE-II and EGEE-III projects [29]. The language consists of a set of pre-defined attributes which describe both jobs and resources. Examples of these attributes are *executable* name, *JobType*, *ExpiryTime* and *Input*- and *OutputSandbox* to specify the files needed to start the jobs and obtain their output respectively. The GLUE schema, which is an extension of the Classad language [55] is a specification for defining attributes used to describe grid resources and services.

2.7.3 Resource Allocation in Volunteer Computing

At the level of the individual owner of a PC, allocation 'policy' is ultimately dictated by that individual. No formal policy may exist, but it is likely that the owner of the resource will have an affinity towards the work of the chosen project.

In the World Community Grid initiative (2.4.5), administrators take a close interest in the type of project to which it allocates resources. They also vet project aims and provide technical, security and reliability guidance for application code used. In keeping with the social and economic consideration of volunteers outlined in section 2.4.1, only projects which are 'not for profit' and which have a 'humanitarian orientation‘ are allocated volunteered resources by WCG administrators [20]. This allocation process is carried out by WCG personnel who meet on a periodic basis[1].

2.7.4 Resource Allocation in Commercial Cloud

Full free-market forces such as supply and demand, cost effectiveness and ‘return on investment’ apply in allocation of commercial cloud resources. As the cloud market and the technology used matures, the barrier to entry for providers is lowering. Similarly, usage of cloud resources is likely to get easier as user interfaces and APIs evolve. Open-source cloud software such as OpenNebula [30] and Eucalyptus [31] allow individuals and small companies to investigate using privately owned clouds by way of introduction to the technology. This means that in the future, a quite fluid cloud resource market may evolve, and that computing resources will become even more a commodity.

2.8 Summary

The chapter outlines different computing resources, and the many different technologies and motivations which underpin them. There are a number of diverse methods used to aggregate large numbers of CPUs and storage devices. This diversity does not ameliorate the prospects of all these resources ever working together in harmony.

Grid to volunteer interoperability, and desk-top to grid interoperability systems are common. However, interoperability between volunteer computing systems and commercial resources is less common. Perhaps this is because of the relative ‘youth’ of cloud computing, or perhaps because of the social and economic differences between volunteer and commercial cloud resources.

Not all computer resources can be deployed for every type of computing work. In some cases, technical and physical reasons may exist, but as described in section 2.4.5, ideological reasons may also be obstacles. Technical and physical resource matching of work to resources is the subject of extensive and evolving research. However, this

[1] Source: personal email communication with Kevin Reed of WCG and IBM.

is not so for ideological matching. This term *ideology* is used as it encompasses the ideas behind several attributes which may apply to computer work and resources. It is defined by the Oxford English Dictionary [43] for this text as:

> *"A systematic scheme of ideas, usu. relating to politics or society, or to the conduct of a class or group, and regarded as justifying actions, esp. one that is held implicitly or adopted as a whole and maintained regardless of the course of events".*

Further evolution of computing resource matching, and subsequent allocations will mean attempting to address ideological aspects. This idea is further explored in chapter 5.

Agents and their role in resource allocation in this thesis were briefly introduced in chapter 1. In the next chapter, we will return to this topic and look at their background, and in particular Social Grid Agents, the technology used in this thesis.

Chapter 3

Software Agents

3.1 Introduction

The word *agent* is widely used in everyday language. A common meaning is a person who acts on behalf of another person, often managing their business, financial or contractual matters. It is not unusual for the agent to have some knowledge, expertise or locational advantage to offer the 'client' for whom it is acting. It is also common for a 'client' to engage the services of many different agents, each offering their own expertise or services to their client.

3.1.1 Definitions of Software Agents

In computer science, an agent is a piece of software, usually running continuously, that carries out actions or instructions for a client (a human user or another computer system or program) in the environment in which the agent operates. Definitions for the concept of an *agent* are plentiful (perhaps not surprising for a concept which has many definitions outside of computing also). However, several definitions from the literature generally agree and a number are used here to define the concept more completely.

Jennings [78] states that an agent is...

> *"... an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that env-*

> *ironment in order to meet its design objectives"*

Luck [95] defines an agent as...

> *"... an autonomous, problem-solving computational entities capable of effective operation in dynamic and open environments"*

Singh [114] states that...

> *"in the true sense of the word, an agent is a persistent computation that can perceive its environment and reason and act both alone and with other agents. The key concepts in this definition are interoperability and autonomy."*

Weak and Strong Notions of Agents

In terms of complexity of both the software agents themselves and of the actions they carry out, Wooldridge [129] delineates agents along lines of *weak* and *strong* notions of what an agent is.

Weak Agency: An agent having the following attributes exhibits weak notions of agency:

- The ability to operate (at least in some fashion) without human intervention and having a degree of control over one's actions (autonomy).
- The ability to contact and interact, usually using a defined *agent communication language* [59, 96], with other agents (social ability).
- The ability of an agent to sense its environment and to make timely reactions or responses to changes in it. This environment may include an interface with a human user, other agents or the underlying system in which the agent operates.
- The property of being able to embark on one's own goal-driven initiatives (pro-activeness).

Even a simple computer program which exhibits the above traits embodies the concept of an agent and this definition of agency is commonly used in the area of agent-based software engineering [125, 90].

Strong Agency: In areas such as Artificial Intelligence, the term *agent* has a stronger and more specific meaning than that of the weak notions of agency. In addition to the properties outlined above, agents meeting a stronger notion of agency embody concepts which are more usually applied to humans. These include mentalistic notions like knowledge, beliefs, intentions and obligations. The mental state of the agent, and the ability to reason about an action the agent may take are part of this capability.

Agent Knowledge, State, Behaviour and Reasoning – Beliefs Desires Intentions

With its roots in a philosophical model of human reasoning [53], the Beliefs, Desires and Intentions model (BDI) is a commonly accepted model of state and practical reasoning in the agent theory and language community [72]. *Beliefs* represent what the agent knows of the world. However, this knowledge may be imperfect and incomplete (for example one can believe something that is untrue). *Desires* represent some desired end state, some of which will be feasible and some not. *Intentions* are generated by exploring scenarios and possibilities which are consistent with beliefs and desires. Generating *intentions* is generally accepted as being a difficult problem [126, p.25] [128], requiring complicated CPU intensive algorithms to explore the many possible permutations which beliefs and desires may present. Georgeff [72] uses a simple example of how humans can recover from “a missed train, or a an untimely flat tyre” because “we know where we are (through our beliefs) and we remember where we want to get to (through our desires)”. Contrast this scenario with a computer program making similar decisions where there is no provision to automatically recover and use an alternative plan, or to make new discoveries or explore opportunities if they serendipitously present themselves.

Further attributes and features found in more mentalistic agents include the ability to learn, adaption in new environments, negotiation capabilities and determine the veracity of information received.

Other Attributes of Agency

- Mobility is the ability of an agent to move within an electronic network or system.
- Temporal continuity, the ability to execute continually, but more importantly, to be able to maintain its execution state for an extended period. The latter may involve a form of saving of state and the ability to re-assemble this state after a failure or restart of an agent.

3.1.2 Agents for Wrapping Other Technologies

While agents can often carry out tasks in their own right, they may also occupy systems which host other services.

If such agents can invoke their underlying systems while also being able to communicate with other agents, they can be thought of as being 'wrappers' for the underlying services. The agents then become a layer of software above the underlying systems and may even facilitate interoperability between different systems or services. [79, 107].

3.1.3 Mobile Software Agents

A mobile agent is one which can use network infrastructure to migrate from one host system to another. As well as possessing some of all of the characteristics discussed thus far, they represent a class of agent that can transmit its capabilities between systems [103].

Some programming languages lend themselves to this agent migration. Examples are Java and Telescript. Java, because of its 'write once, execute anywhere' attribute supports the notion of heterogeneous execution sites for mobile agents. Telescript [118] is an object oriented language designed for network communications. Central to this language is the ability to send code (e.g. an agent) over a network to a system where it can carry out a task for its originator.

Mobile agent may be chosen over stationary ones for a number of reasons. Where the agent application is small, it may be more feasible to move the application to data as opposed to the other way around, and thereby reducing network traffic. They may also be programmed to act autonomously, reacting to changes such as overcrowding of their present environment. Also, if a signal to shut down a host machine is detectable, a mobile agent may react by evacuating itself from that system.

Migration of an agent is achieved by an agent saving its current data and 'transmitting itself' to its new location, in effect code duplication combined with saved state.

3.1.4 Agent Communication Languages

Agents that need to share knowledge require structured communications and a mechanism via which they can interact. Agent communication languages (ACLs) provide this and facilitate information flow between agents using what is usually an independent transport mechanism. The structure of a communication message requires that it has some content and some information about the meaning (or semantics) of that content. Within an agent, the following components are needed in order to integrate communication:

1. APIs are required to compose, send and receive messages.

2. A naming infrastructure is required for message sending.

3. A set of reserved words to describe message types, each of which will describe a performative act is required.

Items (1) and (2) are reusable for any deployment of a given language implemented in a given agent. Item (3) is largely specific to roles and tasks of a given system of agents. It contains specifications which need to be agreed and shared between interacting agents and which have been planned and designed specifically for the tasks required. It should also take consideration of pragmatic requirements for a communication system and will include basic actions such as send, receive and request. Every agent which communicates in a multiple agent system (MAS) has its own API as mentioned in (1) above.

Conceptually, an ACL is structured in layers, with syntax and content. These can be loosely conceived of as outer and inner layers respectively. The outer layer contains the meaning or semantics of the message. This meaning can apply to the type of message, and how the content is intended to be used (i.e. it is a request, command, reply to a request or command). The outer layer may also encapsulate the message in a format which allows it to be sent and may also contain information like the sender and recipient. The outer layer may be further divided into separate transport information which applies to the message content. The inner layer contains the content of the message. There is no need for the outer layer to be concerned with the content and most ACLs are agnostic about what constitutes the content as long as the language is able to represent and transport it. This separation of concerns lends itself towards using different technologies or implementations for different layers of the communication.

Two common and typical agent communication languages are KQML [67] and FIPA ACL [63, 115]. KQML and FIPA's ACL are superficially the same and have similar structures, as is Classad when deployed as an agent communication language. Early ACLs originated in the artificial intelligence (AI) and agent communities [86].

KQML and FIPA's ACL are designed and can be used to represent a piece of knowledge as content with some semantics associated with that content.

A semantic language (and its vocabulary) is a sub-component of an ACL and it is important that this is standardized. Fixed elements of a semantic language are sender, receiver, performative and content. The remaining elements are often agent and application specific.

Conversation Policies and Protocols

A natural extension of sending and receiving *one* message is the idea of a collection of messages (or conversation) pertaining to one subject or task. A conversation policy

(CP) is a task-oriented shared protocol for a shared sequence of messages carried out in order to accomplish a task.

Originally evolving from the need for a specific response following a particular request, the specification of 'legal' sequences during agent interactions were formalised. These became knows as *conversation policies* and have become a separate research thread. The task could be placing a bid at an auction, requesting information from another agent, etc. [86]. CPs should therefore be specified, agreed and shared among agents. Responses should be expected, mutually exclusive and collectively exhaustive within the range of possible answers. This ensures that no unexpected answer will be sent in reply to a conversation component (which would lead to a 'hang' situation).

To carry out tasks involving interactions with other agents, each sub-task (however small in relation to an overall task) requires a set of messages sent back and forth between a pair of interaction agents in order to complete that task. In agent interactions, the alternative to adhering to a predefined interaction for a conversation is relying on a deep understanding of the domain and the task in hand [86]. In the absence of mentalistic agent capabilities, using a scripted conversation may be adequate and desirable (a drawback of mentalistic capabilities is the level of complexity and logical reasoning required in the agent).

Having a conversation policy means an agent has the capability to carry out that protocol, catering for individual messages. The list of an agent's CPs also describe the list of tasks which are 'doable' by that agent, i.e. the agent's abilities. In the execution of a CP, the emphasis moves from the content of the messages to the outcomes of each message in sequence and the behaviour of the interacting agents in response to these outcomes.

3.1.5 Agent Oriented Design – Discussion

Why Use Agents in a System?

Foster [70] states "An agent-oriented approach to system engineering means decomposing the problem into multiple, interacting, autonomous components that have particular objectives to achieve and are capable of performing particular services".

Individual agents present well defined boundaries and interfaces when deployed in a problem solving scenario. They are often designed to fulfill a specific role and are programmed to achieve the goals of that role. Most problems require more than one agent and this reflects what are often decentralised problems.

Epstein and Axtell [64] argue that trade and exchange between agents provides at least one motive for sharing and trading resources. They demonstrate how trade in a population of agents increases the carrying capacity of the environment in which they live (when taken in toto). It is perhaps intuitive that sharing, trading and co-operation in usage of resources will allow an environment to support more inhabitants than where each head of population must provide for all its own needs, be that a Grid of computer resources or human societies. It is also worth pointing out that trade between agents for required resources introduces dependencies on other agents or entities.

Agents versus Software Objects

Agents are in some ways analogous to software objects in the way that they both facilitate agent-oriented and object-oriented design. Agents can be said to be an abstraction of objects but can be distinguished from software objects in that they are stand-alone entities which are capable of making decisions about their actions and interactions. This is not so for objects.

While agents and objects adhere to principles of information hiding, objects are generally passive until invoked and objects do not encapsulate behaviour choice. An object can be invoked by any of its public methods by another object, while agents use messages (sometimes containing authentication) to communicate and request actions from their peers. Furthermore, an agent can also be enhanced with further functionality, over and above what it may have had when originally created.

Object orientated programming offers only minimal support for specifying relationships between objects other than that provided by static inheritance. Object oriented design aids software development, and likewise the field of agent oriented design offers a route to better design of complex multi agent systems [76, 90].

Agents and Grids

Foster [70] uses the metaphors of *brain* and *brawn* to represent attributes and capabilities of agents and Grid computing respectively. Agents can provide the capability to form teams, negotiate towards obtaining resources necessary to complete a task, etc. (i.e. the brains) while Grid middleware and infrastructure provides tools for access to extensive computing resources and storage of large amounts of data (the brawn). This analogy indicates a convergence of interests where the Grid system provides a robust infrastructure for agents to operate in, and the agents provide flexible and autonomous behaviour within that Grid environment.

3.2 Social Grid Agents

Briefly introduced in chapter 1, Social Grid Agents (SGAs) [107] are an agent technology which originated from the Computer Architecture and Grid Research group in Trinity College, Dublin [32]. Their development is motivated by the diversity and complexity found in allocation of Grid computing resources. SGAs are based on a philosophy which aims to be:

> "flexible and powerful enough to allow SGAs to mimic in Grid computing some of the behaviours that constitute the bare basis of social and economic interactions" [107, p.56].

These motivations now extend to federated clouds and hybrid grid-clouds. For work and resource allocation and interoperability in such distributed computing environments, prototype SGAs have proved useful in two ways [107]:

- They can act as enforcement points for policies pertaining to usage and resource allocations.
- They can act as interoperability wrappers (or interfaces) for different types of services and other resources which may underlie a system of SGAs.

3.2.1 SGA Internal Architecture

SGAs are essentially Java applications which act as message processors, with each agent presenting a uniform interface which receives and returns a message.

Internal Data-structure

Central to SGA architecture is the agent's internal data-structure which is used to store all the abilities which an agent has at its disposal. This data-structure is a Java hashmap[1] with each entry being made up of an *action name* as its *key*, and the *binary* of the ability associated with that action is its corresponding *value*.

When a message is received, its action name is checked against the contents of the data-structure. If a corresponding entry is found, one of these binaries is instantiated and used to process that message. If the internal data-structure does not contain a corresponding entry, the message is deemed to be 'not supported'. This behaviour, in effect causes the data-structure to act as a 'gatekeeper' for what messages an SGA can or cannot process.

Types of abilities

The abilities stored in the data-structure of an SGA are made up of two main groups; *Service-providers* and *Processors*.

Service-providers have the logic or 'knowledge' to deal with a particular type of message. They can carry out a task in the environment of the agent or act as an interface to underlying services.

Processors deal with messages which require a series of sequenced actions and are termed 'agent behaviour engines' [107]. They can invoke other Processors, Service-providers or other SGAs when carrying out a task. Processors can behave *synchronously* by processing a message and immediately sending the output message, or *asynchronously* where the task is of indefinite duration.

[1] This was a design choice by Pierantoni, but other similar mechanisms may also provided the same functionality.

Providers and *Processors*, like the agents of which they are a part, are essentially message processors. They can be thought of as being 'the ability to carry out an action'. In a system of multiple SGAs, relationships, organisation structures and control between agents are also defined by abilities.

Figure 3.1 shows the abstract architecture, showing the different types of abilities and potential message paths between them when processing a received message.

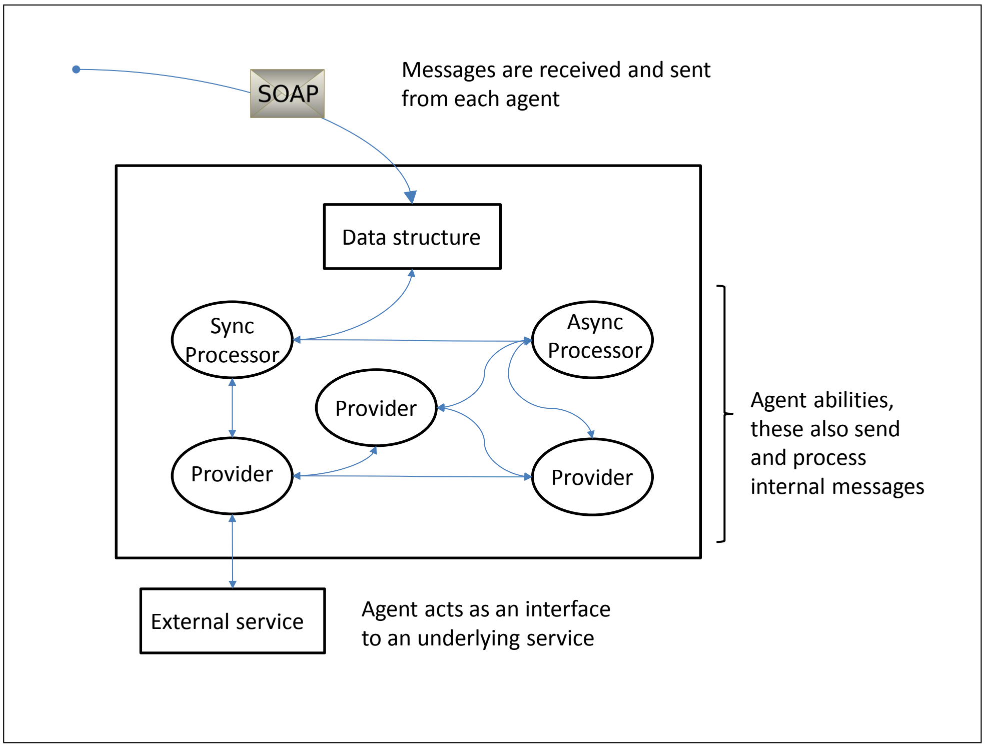

SOAP
Messages are received and sent from each agent
Data structure
Sync Processor
Async Processor
Provider
Provider
Provider
Agent abilities, these also send and process internal messages
External service
Agent acts as an interface to an underlying service

Figure 3.1: Abstract architecture of an SGA.

3.2.2 Classad in SGAs

The functional language Classad [109] from the Condor project [117] plays an integral part in the current SGA implementation. Originally developed for *describing and matching* computer jobs and resources, SGAs exploit its three main features:

- *Structure:* Information expressed in Classad is structured as user-defined *key-value* pairs (e.g. `[CPU=2600]`).

- *Matching & Ranking:* Values are matched against each other, and where there are several matches, these can be ranked based on one or more of these values.

- *Functional Expressions:* Values can be expressed as Functional Programming statements.

Classad was chosen because of these combined features, but in particular for its functional programming capacity. Evaluation of values expressed as functions exhibit *side-effect-freedom* and *referential transparency* [98, p.77]. From the point of view of auditing an action or transactions between a number of agents, freedom from side-effects means that the functions within expressions will evaluate to the same value, regardless of the order in which they are evaluated throughout the agent system. In other words, the history or order of the evaluation is not relevant and so it does not have to be considered. For referential transparency to exist, a function can be replaced by its value without changing the overall function (or expression in the case of Classad).

This offers the potential to audit and prove the outcome from what may be multiple and complicated expressions, particularly those that involve economic actions. This was a principal motivation behind SGA research.

3.2.3 SGA Communications

Classad expressions are used to define SGA communications for both internal and inter-SGA information flows.

The structure of the message is extensible but must at least contain the object or *content* of the message, and the *name* of the action to be carried out on that content. Using the Classad libraries, parts of the message are matched against other Classad expressions which are embedded in various components of the agent. As messages are received and passed through the agent's components, evaluations and matching of outcomes determine how the agent behaves and responds.

In the original work of Pierantoni [107], SGAs were initially implemented as web services in a Globus Toolkit 4.0 (GT4) container [23]. This allows them to be used in distributed locations, utilising SOAP messages over HTTP for communication transport. Classad messages are enveloped in the SOAP messages of that environment.

3.2.4 SGAs and Topologies

Social grid agents can be organised in various topologies and layers. Examples of these are economic, social and production topologies, each with associated information flows.

An agent in the "social layer" can own or control "production agents", the 'utility' of which can be exchanged and traded. This allows the social layer agents to engage in transactions with other social layer agents. Although the name 'Social Grid Agent' is used for all agents in the work of Pierantoni [107], agents in the production layer are often called Production Grid Agents (or PGAs).

3.2.5 Migrating SGAs to New Web Service Technologies

A number of issues arise when running SGAs in the Globus Toolkit framework. These are in the area of ease of deployment and use of external JARs for dependencies for the agents. These issues impact on the ability of an agent owner to deploy and remove agents from an MAS[2]. Further details of these are discussed in appendix L on page 215.

Migration to Newer Web Service Technologies

SGA design and architecture is independent of the platform on which it is run and migration to a different web service platform is both desirable and feasible, and also necessary.

Various web service technologies were examined, leading to SGAs migrating to the CXF framework [33], to be run as a web service in an Apache Tomcat container. Web Archive files (WAR) files were built using standard Maven build tools. This migration immediately addressed the following issues:

- A Tomcat container can automatically deploy or undeploy a service on detecting that a WAR file has been copied into or removed from the appropriate location in its file system. This all happens without the need for a container restart.

[2] For readability and consistency, the indefinite article (an) is used before the acronym *MAS* in this thesis, i.e. it is intended to be read as its individual letters M - A - S.

- All classes and dependencies are packaged into each individual WAR file making it a stand-alone entity for atomic deployment.
- Tomcat containers use a hierarchical system of class loaders where each service is assigned its own class loader at the bottom of that hierarchical tree. This means that dependency classes for each executing SGA are isolated from each other.

A further discussion on some of the differences between using Globus and the CXF framework can be found in appendix M on page 217.

3.3 SGAs compared to Similar Technologies

There are some similarities between Social Grid Agents and other types of agent technologies, in particular in the areas of general attributes and architecture, communications and structure of the language used.

Considering the definitions in section 3.1.1, SGAs do not exhibit reasoning or artificial intelligence attributes and this distinction is made in [107, p.39]. Although not explicitly stated in [107], SGAs should be thought of more as *instruments* owned and controlled by an owner or 'executor'. They are placed in an environment to enforce their owner's policies and wishes.

For communications, the way in which SGAs use Classad conforms to the *key-pair* values semantic-content format. This is similar in structure to the KQML [67] and FIPA [63] [115] languages. In addition to this, key-pair values can be nested as content for other messages.

3.3.1 Example of a Similar Agent Technology

Software agents have become a mainstream technology [127]. For example, the *J*ava *A*gent *DE*velopment Framework or JADE [51] [50] is a typical software agent framework. It can run in multiple Java environments e.g. JSE$^{\text{TM}}$, JEE$^{\text{TM}}$ and JME$^{\text{TM}}$. JADE enables quick and easy development of peer-to-peer multiple agent systems and is a FIPA [34] compliant middleware.

The framework consists of software containers in which agents can run, libraries to create agents and a GUI for user interaction. There can be one or more containers, each on a different system or host. Each container instance must have a 'leader' agent which communicates with other agents in the container, and a directory agent which acts as a *yellow pages* directory for the container. *Behaviours* or binary abilities can be added to an agent at development time or added later as long as the binary can be transported to the agent's host. The framework is open-source and is widely used commercially in the telecommunications industry [51] [121].

There are notable similarities and difference between JADE agents and SGAs. While the modular architecture and design approach is similar, one significant difference is how agents are exposed as web services. Each SGA is an individual service although all SGAs expose a uniform interface and WSDL description. In the case of JADE agents, one web service exposes all the agents in a given container. Multiple interfaces and WSDL descriptions are then exposed for the different services of each agent in the container.

3.3.2 SGAs and Mobility

Although not currently implemented in SGAs [107], it may be possible to save a 'current state' of an SGA by making its internal data-structure persistent. This data when combined with the original application may be sufficient to recreate an SGA in a new location. Currently, SGAs do not have this feature implemented.

3.3.3 Web Service Orchestration

Although conceived from different motivations, creation and configuration of multiple SGAs is similar in concept to orchestration of multiple web services [106]. However, a fundamental difference is that multiple SGAs expose a single uniform interface. This may not be the true in the case of a business process composed of composite web services, each of which may expose a variety of interfaces in each service.

3.4 What Next for Social Grid Agents?

The current SGA implementation and its internal matching and evaluation mechanism (Classad) are agnostic to the tasks, actions and values which they handle. In the prototype development [107], SGAs were deployed in a grid computing environment.

It is envisaged that a multiple agent system with large numbers of agents (100s or 1000s) populated by SGAs may exist. This will mean that there will be many different agent creators and owners. Each owner may own and control one or more agents, with each one having an interest or stake in the environment.

Candidates for representation by agents in such an MAS are entities such as individuals, institutions or enterprises which produce or consume computing services. In effect, the SGAs become producers and consumers of these services, co-existing and interacting with other agents as needed.

Towards this end, it would be very desirable to have SGAs exhibit the following characteristics:

1. Description functionality, allowing agents to use a description that reflects their social and economic attributes and outlooks, i.e. their *social profiles.*

2. 'Plug and Play' of agent abilities, allowing dynamic addition and removal of functionality and behaviour of the agents.

3. Scalable creation of SGAs which facilitates configuration and execution of 1 and 2.

These make up part of the research aims of this thesis. Item 1 is addressed in chapter 5, but before that, items 2 and 3 are the subject of the next chapter.

Chapter 4

A Proliferating Multiple Agent System

4.1 Introduction

This chapter describes research to further develop Social Grid Agents [107]. It explores developments to enable automated provisioning of a multiple-agent system (MAS) where the configuration of agents are specified in a central dataset (be that a data file or a database), and where the addition, removal and reconfiguration of agents is possible.

The following list summarises the attributes of such an MAS that are addressed in this chapter:

1. Scalability: Agents within the MAS need to be quickly and easily produced in large quantities (for example 100 or 1000).

2. It must be easy to add and remove agents to and from this MAS without impacting on other entities within it.

3. It must be easy to configure and reconfigure agents with the desired set of abilities.

4.1.1 Multiple Agent System Planning

In the context of addressing a distributed task and the ‘end to end’ requirements planning for a multiple agent system, the output from such a system design process will be an ‘overall plan’ which specifies the number of agents, their locations and the make-up of their internal configurations.

In this thesis, it is proposed that configuration of individual SGAs and MAS layout is defined in a combination of *two* different types of machine and human readable data sets. The outcome of this planning process feeds directly into the contents of these proposed data sets.

The use of these data sets to create the actual agents and subsequently configure them is merely a mechanism to ‘bring into being’ such an overall MAS plan.

4.1.2 Chapter Layout

The remainder of this chapter is structured as follows: In section 4.2, developments in individual SGAs which make them easier to reproduce are discussed. Section 4.3 describes a proof-of-concept which enables creation and configuration of large numbers basic SGAs with minimal abilities. Further to this, the initial concept of loading a binary class in a remote agent which has been serialising and transporting to it is explored. In sections 4.4 and 4.5, some tests, results and observations on this initial task are described.

Having proved the concept, a detailed examination of a typical (and already existing) system of SGAs from Pierantoni’s work [107] is carried out in section 4.6. This determines how the above proof-of-concept may be used to effectively configure such SGAs. It also examines at the different forms of binary classes and objects which need to be transported to remote agents, as well as what ancillary information must accompany them.

Section 4.7 describes an implementation which creates the components necessary to achieve this. Extensibility and reuse of some of these components are discussed in section 4.8. Finally, in section 4.9 some other developments in SGAs and their creation are briefly discussed.

4.2 A Proliferatable Multi-Agent System of SGAs

For SGAs to become more usable, some limitations outlined here and in Pierantoni's original work [107] need to be addressed. The migration of SGAs to newer technologies discussed in sub-section 3.2.5 was intended to partially facilitate this. This section further discusses exploiting SGA architecture in conjunction with exploiting the benefits of migrating to the CXF Framework and Tomcat web-services.

4.2.1 Basic Agent Strategy

One obstacle to quick and easy development of large numbers of SGAs is the requirement to develop and build each agent individually in an IDE such as Eclipse [35]. Working in an IDE, multiple agents can be built by duplicating the code base and making a small number of edits in the Java code and supporting XML files. The largely manual procedure does not scale well and effectively only changes the name of the SGA. Without further changes and development, it also does not alter any of the agent's abilities.

A methodology is needed which will allow several different SGAs to be created, with each one having a different set of Service-providers and Processors (i.e. abilities).

One heretofore unexploited aspect of SGA architecture is the capacity to use the Java hashmap API to add binaries (abilities) to an SGAs internal data-structure at any time (including after the SGA is initially started). Another enabling attribute of SGAs is how the combination of Classad messages and SOAP messaging can transport large strings. Using a combination of these attributes, this thesis proposes using a *Basic Agent* strategy. This approach strips all but the basic and necessary abilities from the agent and using this canonical or *basic agent* as a basis for reproducing all agents for an MAS. A basic agent is then made useful by adding the required abilities to it at a later time.

4.2.2 Reproduction of Basic SGAs

Essentially, an SGA starts out as a body of Java code which when built (using standard build tools, e.g. ANT or Maven) becomes a WAR file. However, only a small number of edits are actually required and these might be carried out using a simpler and automated procedure.

An example methodology for code editing

Different approaches were explored to achieve the required code changes, for example using the Linux command line tools `diff` and `patch` to automatically alter the canonical code base. The Linux tool `sed` (or *stream editor*) yielded a successful method. Based on `sed` capabilities, a script was developed which took the new SGA's *name* as a variable. The script, when applied to the body of Java code, targets the locations where the *name* occurs in the code for each SGA. The steps for reproducing an SGA are:

1. Duplicate the code base to a new directory.
2. Run the editing script on this to make the code changes.
3. Build the altered code.
4. Harvest the new (uniquely named) WAR file.
5. Delete the newly created code directory.

Multiple SGAs created this way will be identical in function, only differing in name. This name difference is not superficial, but is intrinsic to the Java code and internal build and CXF related XML files. However, these differences are adequately dealt with using the editing script described above.

4.2.3 Addition of Binary Abilities Using Serialisation

In keeping with the *basic agent* strategy, the reproduced SGAs will only have the minimum of binary abilities needed to start running and to receive a message. These are of limited use unless they can be enhanced by adding further abilities.

In the original implementation [107], abilities are loaded to the agent's internal data-structure when the SGA is initialised in the web service container. Objects of the binary abilities are instantiated in the Java code with the required `import <class name>` for the *ability* class present. However, if an object of a class can be instantiated from another source, it can similarly be added to the data-structure without the required `import <...>`. All that is required is to transport the object to the SGA.

Serialisation

Serialisation is the conversion of a binary object to a format which allows it to be saved to a file or transported over a network and is offered by many programming languages. The reverse is called *deflating* or *marshalling*, where the serialised format is re-converted to become a binary object again.

Base64 Encoding [81] is a serialization method which is independent of programming language. It is based on a set of 62 ASCII characters (a-z, A-Z, 0-9). Base64 strings can be handled and stored in text based file, offering sufficient versatility for use in this work.

Sending Abilities to SGAs

In the development environment, Java binary files of SGA abilities can be read from disk and converted to Base64 strings, then inserted in to the contents of a Classad message for sending. On receipt in the SGA, the contents are simply deflated to become a binary object again and become available for addition to the SGA data-structure. To exploit this, the *basic agent* has one important ability which has been developed specifically for this purpose. It has the following behaviour:

- It expects the contents of the message to be a Base64 representation of a binary ability.
- It can deflate the message content, instantiate a class of the ability and add it to the SGAs internal data-structure.

To recapitulate, we now have a methodology by which a *basic agent* can be easily created. It is deployed by simply placing its WAR file on a web service container machine. Once running, it can receive a message which will add to its abilities. In this thesis, incremental addition and removal of abilities will be called 'configuring' and 'reconfiguring'.

4.3 Making a Multiple Agent System of SGAs

The sequence for creating first a single SGA, and then multiple agents are discussed in this section, along with how the requirements for a multiple agent system outlined in section 4.1 are addressed. This stage assumes the a priori availability of suitable Tomcat containers in which to deploy the agents.

For now, only transport and loading of serialised Java classes in remote agents is examined. Later in sub-section 4.6.1, the more complex aspects of transporting both Java classes and their instances are discussed.

4.3.1 Details Needed for a single SGA Deployment

The information required for an SGA to be successfully deployed are:

1. The (unique) agent name.
2. The IP address (or hostname) where the agent is to be deployed.

Using this information tuple, the following steps can be carried out:

- Item 1 above can be passed to the script which copies, edits and builds the code to produce a WAR.
- Item 2 can be used to copy this WAR onto its hosting Tomcat container.

The details required to create multiple SGAs using these steps will simply consist of a list of pairs of SGA names and IP addresses. A *comma separated values* (CSV) file, or any data file or database is suitable for this.

By using an over-arching *'MAS Creation'* program or script which iterates over this list to carry out these steps for each pair, a multiple SGA system can be created.

Configuration of these SGAs

At this point, abilities can be added to these SGAs by sending abilities in individual messages to each one (for this, a simple CXF web service client can be used). But while the developments outlined so far attempt to address the first two requirements listed in section 4.1, configuring anything other than a small number of SGAs this way offers very limited scalability. A more scalable and repeatable configuration mechanism is needed.

4.3.2 Agent Configuration Files for SGAs

For scalability and ease of specification, it is desirable that the configuration of an SGA is a single atomic action. To achieve this, a single *vehicle* which carries the specified abilities for a given SGA is needed. For this, the following features are considered important:

- It must be a self-contained entity, storable and transferable between systems.
- It should be flexible, extensible and easy to edit.
- It should define the set of abilities for a given SGA so it can be recreated and reused repeatedly, e.g. for verification of experimental results, for disaster recovery, etc.

These requirements can be met by using the XML format. It offers human and machine readability, and extensibility to add further information which may be relevant to an agent. Using the XML structure, collections of individual abilities can be stored and later parsed for loading to the internal data-structure of an SGA. This file will be generically referred to as the `abilities.xml` file, the *abilities configuration file* or ACF for short. A sample of the structure of this file is shown in Fig. 4.1. In Fig. 4.1, the string of characters in the `<base64code>` element is a Base64 representation of a binary ability file. This binary 'ability' matches the name in the `<nameKey>` element.

Introducing an XML based configuration file requires a HTTP server to act as a repository which is accessible to the agents and controlled by the MAS creator.

```
<config>
  <abilities>
    <action>
      <nameKey>DoSomeThing
      </nameKey>
        <base64code>rO0ABXVyAzFACg ...
        </base64code>   <!-- abridged, usually 1000s of chars long -->
    </action>
    <action>
            ...
    </action>
  </abilities>
</config>
```

Figure 4.1: Structure of abilities configuration XML file.

To allow a single XML configuration file to be used, an ability (named `FetchProfile`) has been developed which has the following behaviour:

- It expects to receive a URL string which refers to the location of the abilities configuration file for a given SGA.
- It retrieves this XML file and adds the abilities therein to the SGA.

Where the abilities configuration file is used, this ability must be added to the SGA via a preceding message, typically sent just after deployment. Alternatively, this ability could be included in the *basic agent*, but using an 'initial message' to add it retains the flexibility to configure an SGA (albeit in a limited way) without the support of a HTTP repository.

4.4 Testing All Steps Together

This section describes an experiment to create multiple SGAs in a number of different web service containers using an automated script, a number of sets of arguments from a data set (a CSV file is used) and a set of pre-prepared ACFs.

If the tuple described in section 4.3.1 is extended to include the name of the abilities configuration file and its URL, an SGA can be created and fully configured based on this.

4.4.1 Test Parameters and Preparation

For testing, arbitrary quantities of 100 SGAs in 5 different hosting containers were chosen. A set of XML abilities files was also created (using an automated process not described here), each of them having containing a selection of binary abilities, all of which were serialised Java classes. These XML files were then placed in the HTTP repository.

A list of tuples, stored in a single CSV file defined how the 100 SGAs were to be distributed across the containers, and which ACF was to be designated to each SGA. Henceforth, this file will be referred to as the *MAS Plan File* or MPF for short. A sample line from such a file is shown in Fig. 4.2.

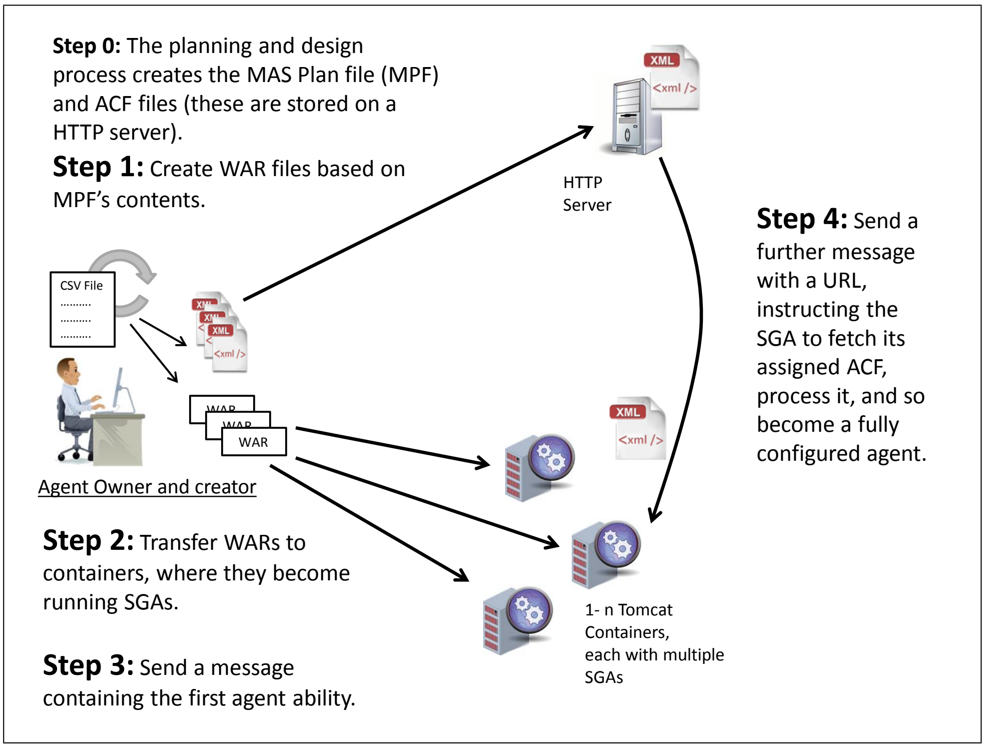
```
agent_0001,
10.53.37.27,
http://10.53.37.25/profile/,
abilities_0001.xml
```

Figure 4.2: Sample line from an MAS Plan File (MPF), columnised for clarity.

For supporting infrastructure, five StratusLab [36] [94] cloud VMs instances were created each having an instance of Apache Tomcat installed. A further VM was used for the HTTP repository to host the abilities configuration ACFs. This layout is illustrated in figure 4.3.

4.4.2 Execution of the 'MAS Creation' Script

Multiple WAR files were created and deployed as described in section 4.3. By extending the *MAS Creation* script described in section 4.3.1, items 3 and 4 in the tuple in Fig. 4.2 were brought into use.



Step 0: The planning and design process creates the MAS Plan file (MPF) and ACF files (these are stored on a HTTP server).

Step 1: Create WAR files based on MPF's contents.

Step 2: Transfer WARs to containers, where they become running SGAs.

Step 3: Send a message containing the first agent ability.

Step 4: Send a further message with a URL, instructing the SGA to fetch its assigned ACF, process it, and so become a fully configured agent.

Figure 4.3: Sequence for MAS Creation

Initial Message to new SGAs

Using items 1 and 2 from each line in the CSV file, the URI address to which to send a message to each SGA was composed. To each different URI address, the same message (containing the binary `FetchProfile`) was sent. At this point, every SGA now had one further ability over and above the *basic agent* configuration.

Full Configuration

Further configuration was triggered by sending a message to each SGA which contained the URL to its ACF (i.e. a combination of items 3 and 4 in the CSV file). On receiving this message, each SGA fetched its ACF and processed the contents as described in section 4.2.3. At the end of this cycle, each SGA was fully configured as dictated by the contents of the ACFs.

4.4.3 Removal of SGAs from an MAS

A further extension of the *MAS Creation* script was made to provide for deleting a WAR file from a web service container. This used items 1 and 2 from the MPF. When executed, the Tomcat automatically undeployed the service of the deleted file.

4.5 Results and Observations

4.5.1 MAS Creation

100 SGAs were created and configured using a specification laid down using a combination of a single CSV (the MPF) and multiple XML *agent configuration files.*

Each WAR file took approximately 12 seconds to build. A further manual delay of 10 seconds was introduced between each WAR file transfer to ameliorate network delays. On the containers, each WAR file took between 5 and 10 seconds to fully deploy. A manual delay was also used here to reduce potential for failures due to CPU and memory bound conditions on the container host machine.

The entire system of SGAs was created, deployed and configured in 58 minutes. This amount of time should be considered in the context of the vision outlined for SGAs in section 3.4 on page 53, where there are many different agent creators and owners. Each owner will only be responsible for creating a relatively small number of agents. This test indicates that, for an individual organisation this can be done in a reasonable amount of time. Initial indications are that the number of agents which can be created is limited only by (a) the available time and underlying resources and (b) the extent to which a complex system can be designed and then specified using the formats described here.

4.5.2 Memory Limitations

During initial test runs, no time delay between each WAR file transfer was used. Some of these deployments failed. These failures were attributed to memory bound behaviour on containers as 100% use of available memory was observed[1]. High memory usage was avoided by sequencing the WAR file transfer to VMs in a round-robin fashion and by introducing a delay between each file transfer. This allowed adequate time for deployment on any given VM. Subsequent to this change, SGA proliferation and configuration worked smoothly although a larger number of containers would allow the delay to be optimised or removed. During both tests, the HTTP server worked without errors and CPU workload appeared to be very low on the VMs which were observed.

While these experiments are not intended to test the capacity of the web service container used, running them gave an indication of their physical system limits, information which will be useful for further MAS planning and design.

4.6 Abilities as Classes and Instances

The MAS created in section 4.4 is simply a proof of concept. The ACFs used in this test contained only a small number of abilities, none of which had any particular function or purpose. It is time to take a closer look at the binary abilities which are found in SGAs. To do this, the following section will examine the available code from a typical multiple agent system from the 2008 experiments of Pierantoni. The system of agents examined is similar in layout and configuration to that described in section 4.3.5 on page 176 of 'Social Grid Agents' [107]. It consists of 8 agents, four of which are SGAs and four PGAs (see section 3.2.4, page 50).

The purpose of this examination is to look at the variety of abilities which are configured in this MAS, and what information needs to be transported to each agent if a correct configuration and functionality is to be achieved by using the approach introduced earlier in this chapter.

[1] The Linux tool top was used to observe this.

4.6.1 Important Concepts of Agent Configuration

The binaries added in section 4.4 were in the form of a full class type definition, i.e. an encoded `.class` file. In a more extensive systems of SGAs, transport of binaries to remote agents is more complicated. Therefore, two important extensions need to be made to the ACF. These are as follows:

1. As well as transporting full class definitions, it is also necessary to transport instances of classes.

2. For some abilities, along with the action-name, multiple other `String` values need to be transported along with classes and instances.

Class Definitions and Instances of Classes – Discussion

The term *abilities* is used here for the group of binary objects known as Service Providers and Processors. These are added to the SGA's data-structure during configuration. Depending on what functionality is required, these binaries are added in two different forms, either as the byte code of a full `.class` file (or class definition) or as the byte code of an *instances* of such a class.

Instances are used for some abilities where the agent needs to maintain 'state' or 'remember' a value from initialisation. Maintaining state also allows an ability to remember changes in state from one message processing cycle to the next. State is maintained by changing the value of a variable in such instances. A typical example of an ability which needs to do this is one which acts as a 'wallet'. The wallet holds a record of how much monetary credit or tokens an SGA has, but also has the functionality to increase and decrease this amount during message processing cycles.

Full class definitions (i.e. the class type) are used where there is no requirement for the binary to hold any state from one message processing cycle to the next. Using a full class definition also ensures that every instance of that object will be a different instance in system memory when the agent is running (in effect, this being the opposite of 'remembering state'). Because a new instance of the binary is created for each message, any variables present are initialised to defaults. At the end of the message processing cycle, the specific instance for that message is destroyed, and along with it, the memory of any values which may have changed.

On retrieval of an object from the SGA's internal data-structure (the hash-map), the implemented logic of the SGA determines if the object is an instance of a ServiceProvider or a class definition (i.e. an object of type `Class<?>`). If an object of class ServiceProvider is found (i.e. an *instance*), then this object is used while processing a given message. When the instance is said to be retrieved from the data-structure, only a reference to the instance is in fact retrieved. Changes made to this instance in memory are preserved, hence state changes are preserved. If the object retrieved is found to be an instance of a class (i.e. `(obj instanceof Class<?>)`, this indicates that a full class definition has been returned. In this case, a *new* instance of the class (a ServiceProvider) is instantiated in memory. Figure 4.4 outlines (in pseudocode) the logic used when a binary is requested from the agent's data-structure. It can be said that using an instance stores functionality and state in the data-structure, while storing a class definition only stores functionality.

```
Object = Get the object from the data-structure

if Object is an instance of ServiceProvider, then

         return the instance to the calling class


if Object is an instance of Class <?>, then

         instance = create a new instance of this ‘type’

         return this new instance of Object to the calling class
```

Figure 4.4: Logic for retrieval of a class definition or an instance from the SGA data-structure

Setting State in SGAs

In the original SGA [107], state could easily be hard-coded in the source code of the SGA. When the SGA was started as a web service, either the class definitions or instances of them were instantiated in the code and then stored in the agent's data-structure. Figure 4.5 illustrates the sequence for creation and loading of classes and instances at SGA start-up as executed in the then hard-coded logic of the SGA. Note that in this figure, when the application (the SGA) is instantiating a new class, it looks to a feature in the Java virtual machine known as the *class loader* for a definition of this class. A brief discussion on the role and relevance of the class loader, along with how classes used in programs are loaded into the Java Virtual Machine (JVM) is merited at this point.

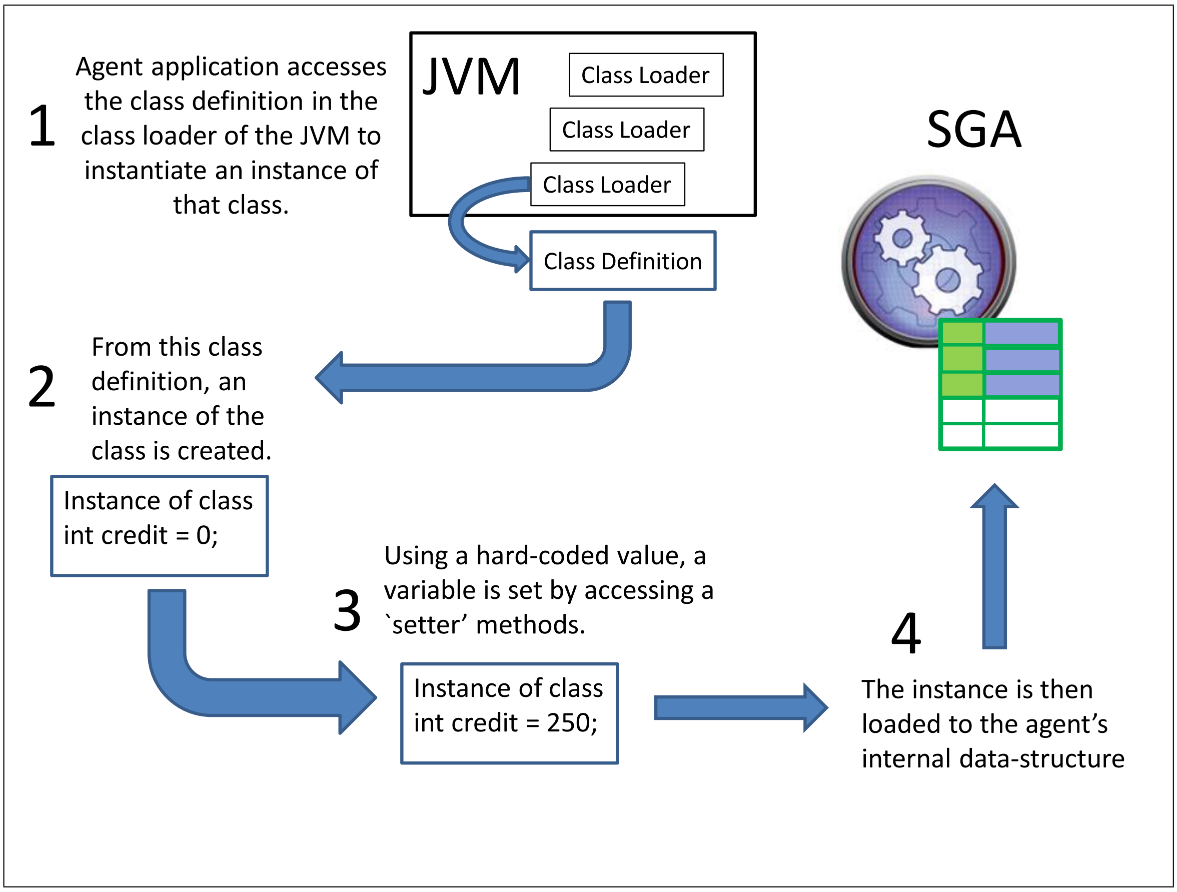

1
Agent application accesses the class definition in the class loader of the JVM to instantiate an instance of that class.
JVM
Class Loader
Class Loader
Class Loader
Class Definition
SGA
2
From this class definition, an instance of the class is created.
Instance of class
int credit = 0;
3
Using a hard-coded value, a variable is set by accessing a `setter' methods.
Instance of class
int credit = 250;
4
The instance is then loaded to the agent's internal data-structure

Figure 4.5: Sequence for creating a class instance in a hard-coded in the SGA of Pierantoni [107].

Class Loading and the Java Virtual Machine

The Java *Class Loader* is a mechanism which loads class definitions into the JVM for use by programs and applications running therein. The class loader has ‘knowledge’ of a number of predefined ‘class paths’ on which to look for such classes. In any Java program, each time an object of a class is instantiated, or even referenced, the bytes of that class must either be already loaded into the memory of the JVM or be available for loading on one of the available ‘paths’. Loading these classes is the role of the class loader.

Within the JVM, there are a number of class loaders. These are arranged in a hierarchy, each one having a parent-child relationship with the one above or below it. At the top of the hierarchy is the bootstrap loader (which is used when the JVM is started). When a class loader receives a request to load a class, it first delegates this request to its parent. This delegation continues upwards until the bootstrap loader is requested to load the class. If all of the class loaders to which the request was delegated fail to load the class, only then will the original class loader attempt to load the class. This procedure is used for a number of reasons, for example, to ensure that classes are not loaded more than once.

Java class loaders use a *lazy* or *on demand* approach to loading classes. This means that the classes required by a program are only loaded just before they are needed (as opposed to loading all the classes available in a system on booting the JVM). This approach allows classes to be loaded at a time after the program has started running. It is this behaviour that allows the classes of the SGAs abilities to be added after the agent has already started running.

If a class definition for a given type cannot be loaded in the class loader, creation of an instance of that class (by either instantiation or by deserialising an object) is not possible. This is the situation that presents itself when an instance of a class is to be deserialised within agent.

4.6.2 Loading Class Instances in SGAs

Figure 4.5 shows the sequence which takes place when creating a class instance at start-up. In this scenario, the configuration of the SGA is hard-coded and all required classes are on the 'paths' available to the class loader. In figure 4.5, from the time the class is loaded into the class loader, it can be said to be 'loaded' and 'visible' in that class loader. Its definition is available there should the application need to instantiate or reference an object of that class.

When a serialised class definition of a binary ability is transported to an agent, it can be deserialised to become a class object. This object is in fact an instance of the class `Class` which defines all classes and interfaces used in Java. As all Java classes are derived from this class, the class definition needed for it will be available on one of the JVM's class loaders (most likely one of those nearer the top of the hierarchy).

When a serialised instance of a binary ability arrives at the agent, its class definition will most likely *not* be already loaded in the class loader of the agent. Furthermore, given that the instance type was not required at build time, it will not necessarily be on any of the class paths available to the class loader either.

Transporting Instances – a Class and Instance Solution

Serialising and transporting an instance of a class to a remote SGA requires a two-strand approach. The solution to doing this requires that both the class definition and the instance (in which the required state is set) need to be transported to the agent. This was achieved by doing the following. At the development side, an object of the class is instantiated. Its state is set to the required values. Both the class definition and the instance are then serialised to become serialised objects (in the form of Base64 strings). These strings are concatenated (separated by a delimiter) and inserted into the abilities configuration (ACF) file for transport. The sequence of events which takes place within an agent is outlined in figure 4.6. The concatenated string is split into its two constituent parts. In step 1 of figure 4.6, the class definition part is first deserialised and loaded to the class loader of the application. In step 2, the instance is then deserialised by passing the bytes recreated from the serialised

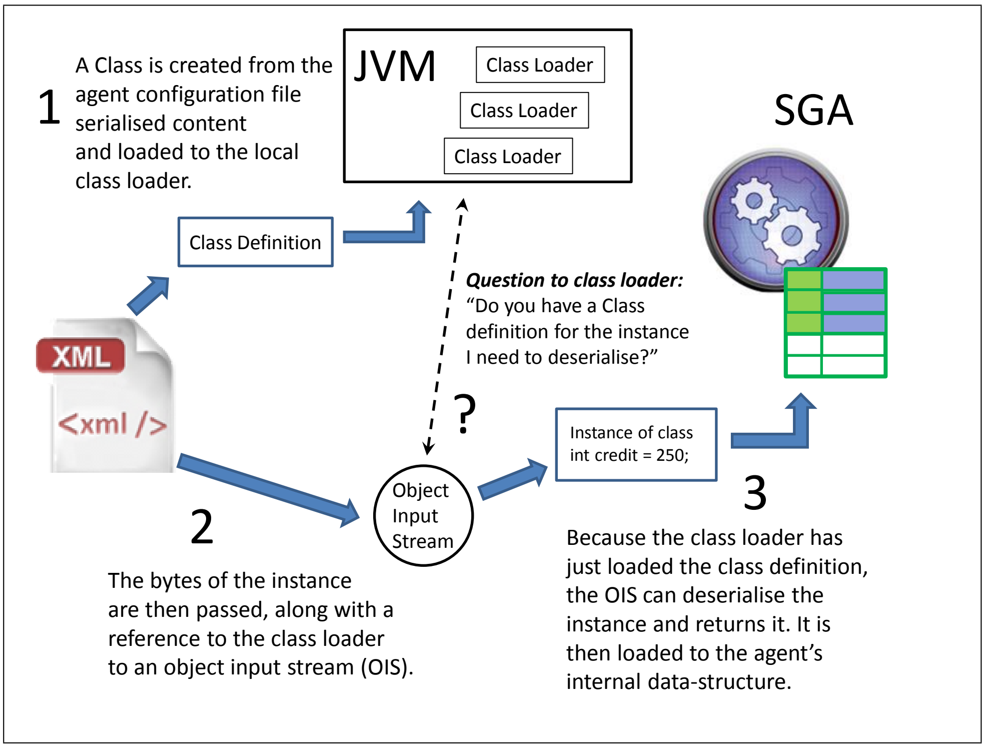

1
A Class is created from the agent configuration file serialised content and loaded to the local class loader.
JVM
Class Loader
Class Loader
Class Loader
SGA
Class Definition
XML
<xml />
Question to class loader:
"Do you have a Class definition for the instance I need to deserialise?"
?
Instance of class
int credit = 250;
Object Input Stream
2
The bytes of the instance are then passed, along with a reference to the class loader to an object input stream (OIS).
3
Because the class loader has just loaded the class definition, the OIS can deserialise the instance and returns it. It is then loaded to the agent's internal data-structure.

Figure 4.6: Sequence for deserialising a class instance from serialised content in an XML ACF file in the SGA, as proposed in this thesis.

object to a Java object input stream. The object input stream also takes as an argument, a class loader in which it will look for a class definition of the instance it is about to create and return. The same class loader into which the class definition was loaded (just an instant before) is used for this argument. The key point here is that the class definition of the instance to be loaded is loaded and 'visible' in that class loader when the instance is being deserialised. Finally (step 3), this deserialised instance can then be stored in the application's data-structure and can be used by the application.

Attempting to deserialise an instance without the presence of the class definition in the class loader causes the Java error for 'no class definition found' (namely `NoClassDefFoundError`). Without a mechanism to transport an instance of a class with its preset state included, and then subsequently load it to the agent's class loader and data-structure, the functionality of the SGAs under examination could not be replicated.

Class Loading Issues Specific to Tomcat

Other problems were encountered when attempting to create instances in the agent environment (as opposed to running SGAs in a software development environment). These were problems which were specific to the web service technology used in this thesis (Tomcat). Details of these issues and a number of solutions used are discussed in appendix B, page 198.

4.6.3 Configuration Information for Classes and Instances.

This section addresses what information needs to be included with the encoded binary abilities within the abilities configuration files used by the SGAs at configuration time. These requirements are driven mainly by what arguments need to be passed to the methods used in the SGA when adding the different types of abilities. These methods and their signatures are integral to the operation of the agents and cannot be altered. Therefore, the agent configuration approach created in this thesis must be able to fully accommodate their requirements.

Method for Adding Abilities Within SGAs

To recap, all of the abilities added to the SGA data-structure are binaries, and each of them has an action-name associated with it. The combination of this action-name and the binary forms the basis for what is needed to add each ability to an SGA.

Within each SGA, there are five different methods which may be called to add abilities to an SGA's data-structure. These, along with details of the signature of each method are listed in figure 4.7. In this figure, another argument, common to each method, refers to the agent's data-structure and is omitted here (for clarity).

At configuration time, for each ability to be added, the SGA needs to know three separate things:

1. Is the binary to be added a class definition or an instance of a class?

2. Which of the five methods of figure 4.7 is to be called?

3. How many and what arguments are to be passed to that method?

```
addSimpleAction(String, Class<?>,...)
addSimpleAction(String, ServiceProvider,...)

addPgaAction(String, String, String, Class<?>,...)
addPgaAction(String, String, String, ServiceProvider,...)

addPgaServiceProvider(String, String, String, String, ServiceProvider,...)
```

Figure 4.7: Methods used in SGAs to add binary abilities.

The combination of methods required for each SGA's configuration varies from agent to agent. This depends on its agent's role in the system. In the range of SGAs examined here, every method is called at least once. In order to achieve a successful and correct configuration, it is necessary that the structure and nomenclature used in the ACF convey this information to each SGA during configuration.

A further Variation on How Methods are Called

Up to now, in the context of adding abilities to the agent's data-structure, action-names and abilities have only been considered in 'matching pairs'. However, the configuration of the SGAs in the 2008 work [107] provides one further variation in how abilities are added to the data-structure. For an ability holding state, and where that state may be changed by a number of messages (each with different action-names), there is a requirement for more than one action-name to apply to that ability.

An example of this is the 'wallet' ability in the SGAs of the 2008 system [107]. This state in this ability represents the amount of money an SGA has at any given time. Unlike most others, three different actions need to be carried out by the *wallet* ability. These actions are *add credit*, *remove credit* and answer the query *how much credit is there*. Not surprisingly, these three actions must apply to the same instance.

Recall that when instances are added to the agent's data-structure, only references in memory to the instances are added. The same 'ability' can therefore be added three separate times, each time with a different action-name. Calling any one of these actions will act on the same instance. In the 2008 work, this was achieved in the 'initialisation' code of an SGA by invoking the method:

```
addSimpleAction(<action-name n>, walletInstance)
```

This method is called three times in succession, each time passing a different action-name argument (denoted by *n* here). However, for each call, the same instance of the ability is added to the agent's data-structure. This sequence needs to be replicated in the mechanism proposed in this thesis in order to achieve a successful configuration.

4.7 Automated Creation of Abilities Configuration Files

Section 4.6 detailed what information needs to be conveyed to an SGA at configuration time; this section looks at how this may be implemented. It further examines the system of SGAs described in section 4.3.5 on page 176 of 'Social Grid Agents' [107].

A customised program was created to prepare the required ACFs. The output from this program was the creation of an ACF for each of the SGAs. Each file contained all the Base64 string representations for the serialised classes and instances specified for each SGA. The ACFs also contained enough information to specify which method is to be used to add them to the SGA's internal data-structure, and also specified what arguments were to be passed to those methods. This section describes how these XML files are created, and in particular how all the requirements which have been discussed since the start of section 4.6.1 are accommodated by the extensibility of the XML format chosen.

4.7.1 Analysing and Specifying SGA Configuration

The first task is to identify what abilities are needed by analysing the available code of the target SGA system. This analysis yields two sets of data by doing the following two things:

1. Compilation of an inventory of all the required class definitions and instances of classes.
2. Determine for each ability if it is a class or an instance and what method and arguments are to be used (i.e. answers to the three questions of section 4.6.3 on page 72).

The list created in item 1 here is used for creation of the necessary serialised object files. The information obtained from item 2 needs to be notated in a data set (such as a machine readable file) for use when creating the required ACFs. For this, a set of comma separated (CSV) files are manually created (one for each SGA). For each SGA, the contents of this file is directly derived from the Java code which was used to configure the original SGAs in the 2008 work [107]. Because of this, this CSV file will be called the *abilities blueprint* file. Some sample lines from some of these files are shown in figure 4.8 and will be discussed further, following the introduction of the mechanism for creating the required serialised object files, next.

```
addSimpleActionINSTANCE,WriteStatus,statusSP

addSimpleActionCLASS,Purchase,JobPurchaseAsyncProcessor

addPgaServiceProviderINSTANCE,Execution,Submission,JDL,
            DirectControl,dcLcg2WmsPga

addPgaActionCLASS,Availability,Submission,JDL,JSAvSyncProcessor

addSimpleActionINSTANCE,HowMuchInWallet#######
            RemoveCreditInWallet#######AddToWallet,walletSP
```

Figure 4.8: Sample lines from the abilities blueprint CSV file used to specify details of what is to be added to the XML abilities configuration file (ACF).

4.7.2 Serialised Object File Creation

The next step is to create the required serialised object strings in a fashion that is somewhat automated and scalable (as opposed to the manual operations which have been carried out so far). The inventory of classes assembled for conversion to Base64 strings is listed in appendix C, page 201. From this list, a subset of these classes needed to be instantiated (to create the required instances). These must then be modified to set the initial state conditions. Next, these instances also have to be serialised in preparation for concatenation with their respective class definition strings for later insertion into the ACFs. To create these strings, a custom Java program was created for each agent which carried out the following two functions as required:

Serialising classes using Base64: Creation of a Base64 string for classes is relatively straightforward. The `.class` file is read from disk into a byte array and passed to Base64 encoding libraries. The string returned from this is written to a text file. At this point, the text file can be considered to be a serialised object.

Serialising instances using Base64: Creation of Base64 string files for instances requires the instantiation of an object of the class. On instantiation, the appropriate setter methods are then called to set the required state. As described in sub-section 4.6.2, the Base64 files for instances contain two strings, the class definition and the string of the instance, separated by a delimiting set of characters[2]. This file is, strictly speaking, not a serialised object as it contains two different strings, however, we will consider it to be one for this description.

The output of these two processes is a set of folders, one for each agent. Each folder contains a collection of text files, one for each of the required abilities for that agent.

[2] The characters used are not within the base64 subset or characters.

Combining the Serialised Object files and Abilities Blueprint files to make Abilities Configuration files

Using the details set out in the abilities blueprint files, an ACF is created for each agent. This is done by a further custom Java program which iterates across the list of abilities blueprint files made available to it (recall that there is one for each agent).

As this program iterates across the abilities blueprint CSV files, each line in any given abilities blueprint file contains details for one ability for that agent. Specifically, it contains how the element created in the ACF should be used later during agent configuration. For example, referring back to figure 4.8, the first item or line in that figure contains the following three items:

- `addSimpleActionINSTANCE`
- `WriteStatus`
- `statusSP`

The item `addSimpleActionINSTANCE` indicates that method `addSimpleAction(...)` should be called when adding this ability to the SGA data-structure. The word INSTANCE is appended to it, meaning that the serialised object string in that element represents an instance of a class. The next item `WriteStatus` is the *action-name* for that instance, also needed when adding the ability to the agent's data-structure. `statusSP` refers to the name of the serialised object text file in the folder for that agent which is to be added to the ACF.

Achieving Multiple Additions of the Same Instance

Note also the last item in figure 4.8 on page 75, listed below. The *action-name* for this line looks somewhat different as it contains three different action-names.

- `addSimpleActionINSTANCE`
- `HowMuchInWallet#######RemoveCreditInWallet#######AddToWallet`
- `walletSP`

Recall from section 4.6.3 on page 72 that certain instances need to be added to the agent's internal data-structure more than once for the configuration to be correct.

This *multiple addition* is achieved by using a single line for this particular type of instance in the abilities blueprint CSV file (as is done for others). The multiple action-names to be used during configuration are concatenated as seen above. In the agent, this string is split into its original constituent parts. For each part found, the same instance of 'wallet' is added to the data-structure, using that part as the 'action-name'. Only one instance of the wallet (with its initialised state) is used in the addition of the ability to the SGA. Specifying multiple additions of the same ability in this way achieves the same sequence that was hard-coded in the original SGA initialisation and configuration code.

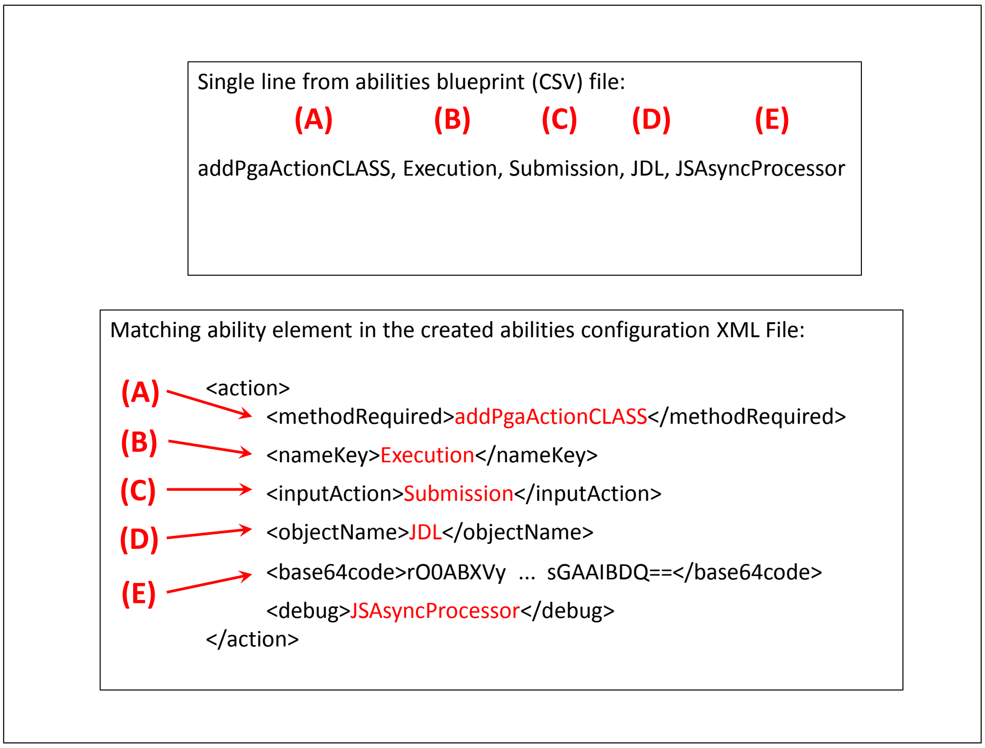
For generality, this 'splitting' approach of the action-name is applied to *all* action-names strings extracted from the ACF. This works correctly as splitting a string with only one action-name finds only *one* action-name with which to add the binary ability.

Extending the XML of the Abilities Configuration to Accommodate Different Method Signatures in SGAs

Figure 4.9 shows a different and more complicated abilities blueprint file line example where the line of the abilities blueprint files contains the action-name and some further details needed by the SGA. Recall from figure 4.7 on page 73, that some of the methods used during configuration require multiple `String` type arguments as well as a binary ability. Figure 4.9 shows how the schema of the abilities configuration is based on what is specified in the abilities blueprint file, and also how the ACF can accommodate the data needed for the various method signatures found in the original SGA implementation [107].

4.7.3 End-to-End Overview of an MAS Creation Framework

Figure 4.10 sums up the steps and mechanisms use to specify and creation a system of multiple SGAs. It presents an end-to-end framework for creating, deploying and configuring an MAS of SGAs. This figure draws together the creation of agent abilities



Single line from abilities blueprint (CSV) file:

```
addPgaActionCLASS, Execution, Submission, JDL, JSAsyncProcessor
```

Matching ability element in the created abilities configuration XML File:

```
<action>
     <methodRequired>addPgaActionCLASS</methodRequired>
     <nameKey>Execution</nameKey>
     <inputAction>Submission</inputAction>
     <objectName>JDL</objectName>
     <base64code>rO0ABXVy  ...  sGAAIBDQ==</base64code>
     <debug>JSAsyncProcessor</debug>
</action>
```

Figure 4.9: Mapping of abilities blueprint CSV file line to an ability element in the abilities configuration XML abilities file.

blueprint CSV files, the serialised object file creation process, the MAS creation mechanism described in section 4.3 and the infrastructure in which a multiple agent system may be created. The reader is also invited to refer back to figure 4.3 on page 63 for more details on the sequence carried out during actual configuration. The HTTP server in figure 4.3 is one and the same as the HTTP server in figure 4.10. Note that there are two different types of CSV files used. The abilities blueprint contains details of the abilities needed for each agent (as discussed in this chapter). The MPF, of which there is only one per multiple-agent system, contains details needed for initial creation and deployment of each agent (e.g. the agent name, its destination IP address and the name and URL of the ACF designated for its configuration).

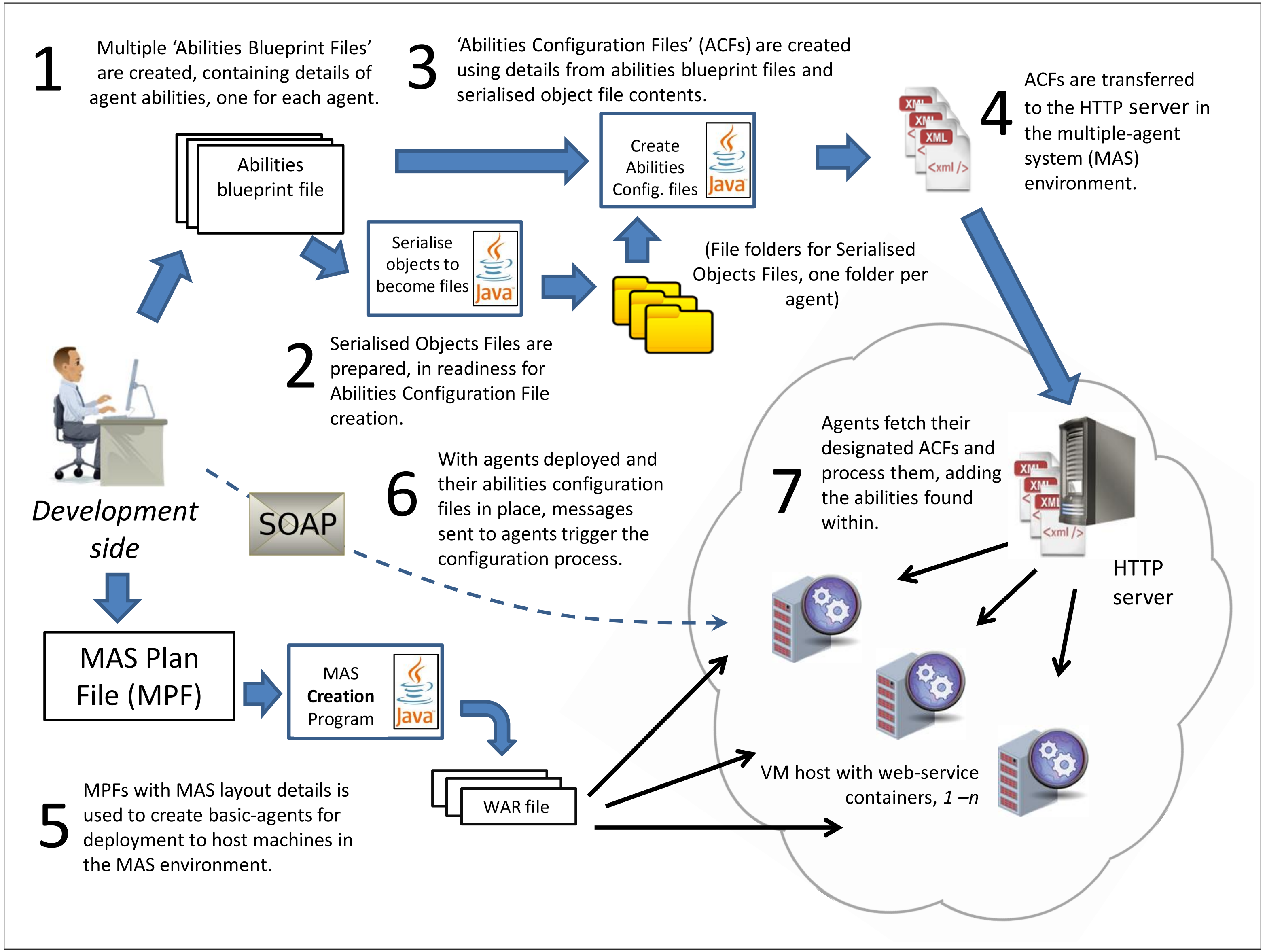

1
Multiple 'Abilities Blueprint Files' are created, containing details of agent abilities, one for each agent.
Abilities blueprint file
2
Serialised Objects Files are prepared, in readiness for Abilities Configuration File creation.
Serialise objects to become files
Java
(File folders for Serialised Objects Files, one folder per agent)
3
'Abilities Configuration Files' (ACFs) are created using details from abilities blueprint files and serialised object file contents.
Create Abilities Config. files
Java
XML
<xml />
4
ACFs are transferred to the HTTP server in the multiple-agent system (MAS) environment.
Development side
MAS Plan File (MPF)
MAS Creation Program
Java
WAR file
5
MPFs with MAS layout details is used to create basic-agents for deployment to host machines in the MAS environment.
SOAP
6
With agents deployed and their abilities configuration files in place, messages sent to agents trigger the configuration process.
7
Agents fetch their designated ACFs and process them, adding the abilities found within.
HTTP server
VM host with web-service containers, 1 –n

Figure 4.10: Summary of all parts which contribute to creation of a Multiple Agent System.

4.8 Including External Abilities Configuration Files

The ACFs described up to now have elements which only contain Base64 strings, their associated action *names* and other strings. This works well for a small number of encoded abilities. However, a typical Java binary (`.class` file) of size 4.0KB converts to 5.4KB in Base64 format (an overhead of 35%). Where there may be greater numbers of abilities to be configured in an SGA (for example, 20+), this will cause the required ACFs to become large and unwieldy.

A separate design iteration of the abilities configuration file structure proposes the concept of 'including' references to other external ACFs by introducing a further element which contains a list of URLs. In conjunction with this change, the internal code of the `FetchProfile` ability (see sub-section 4.3.2) was modified to work as follows:

1. The ability processes a message which has a single URL as its content. This is extracted and passed to a `processUrl(URL)` method which fetches the abilities configuration file.

2. Base64 strings and associated details found in the `<abilities>` element are added to the SGA's data-structure (as before).

3. If an `<includes>` element is found that contains one or more further URLs (for other ACFs), a recursive call is made on the `processUrl(URL)` method.

4. If the ACF fetched in this subsequent call yields further URLs, this recursion continues until all references are exhausted.

Taking care not to create circular references and to catch system stack errors associated with recursion, this approach allows the abilities configuration file to contain references to several separate ACF which may be common and shared with other SGAs.

The external behaviour as originally outlined for `FetchProfile` is preserved, but the ability will now process abilities configuration files which contain various com-

```
<config>
  <abilities>
    <action>
      <nameKey>DoSomeThing
      </nameKey>
        <base64code>rO0ABXVyAzFACg...
        </base64code>
    </action>
    <action>
               ...
    </action>
  </abilities>
  <include>
     <abilitiesURL>http:// ... .xml
     </abilitiesURL>
     <abilitiesURL>http:// ... .xml
     </abilitiesURL>
  </include>
</config>
```

Figure 4.11: Structure of XML abilities configuration file with included URLs.

binations of zero, one or multiple Base64 strings and URLs. A summary of the extended ACF structure is shown in Fig. 4.11.

4.9 Other Developments in SGAs

4.9.1 Making Agents Within an MAS – Discussion

In a real world application of an MAS, a natural and logical development would be to decentralise the creation of each agent's WAR file to somewhere other than the development environment of the agent owner. The most logical place to create WAR files is within the MAS and close to the machines which will host them.

This requires that the entire editing and build process described in section 4.2

is migrated to a machine within the MAS. From there it can be invoked with a command. As the creation of another agent will be in response to a request from an existing agent (or a human operator), it needs to be placed on a machine which hosts an SGA. In effect, that SGA then wraps the process. Outside of the (usually well resourced) development environment, the following considerations become important.

- Is there sufficient disk storage for the body of code required and the new WAR files which are created?
- What commands and file system permissions are needed?
- Will it be in response to a single command for a single agent, or be able to handle a list of details for multiple agents?
- What permissions and ssh key placements are needed for the SCP part of the deployment to be passwordless? This may be set up (manually), as the owner will, by necessity have full access to all the hosting machines it controls in the infrastructure.

4.9.2 General Purpose Message Sending Format for SGAs

The developments proposed in this thesis do not require complex message structures. Therefore, a simple, ‘general purpose’ message type is used.

In the 2008 work of Pierantoni [107], the Classad expression in each message was structured to match the requirements of the ability that processed it. This structure contained multiple numeric and string values. When developing a new agent ability (with different messaging requirements), it was also necessary to develop a number of supporting classes to parse and interpret the Classad content of the new message. These supporting classes were required to be available in both the sending and receiving SGAs. This presents a complexity and scalability issue.

The message type used mainly in this thesis has a simpler structure. It needs one type of parser, and this is capable of parsing all messages sent and received. It has two main parts, a *semantic value* and some *content* which can be considered to be its 'payload'. The semantic value is the 'action-name' and this maps to the 'action-name' of the ability in the receiving agent. It can be said that the action-name effectively describes what is to be *done* with the content. This *semantic-content* structure is in keeping with other agent language structures, the origins of which are discussed in 3.1.4, page 42.

From time to time, the content of a message needs to contain more than one item. This requirement is met by concatenating items which need to be sent as content. At the receiving end, this content is separated before use. Therefore, instead of the complexity being in the Classad structure, any complexity is transferred into the content of the message[3]. When the content of the message has to be separated or parsed, the logic to do this is placed within the ability which processes the message[4].

The number of different abilities needed by SGAs may grow. However, it is envisaged that whatever format the content needs to be in, it should be able to be inserted into the 'content' part of this message format as a single string. Because messages are sent to specific abilities in SGAs, the receiving ability can be programmed to separate and parse the assembled content.

Impact on Existing SGA Functionality

This new message structure does not alter or disable the functionality of the messages in the work of Pierantoni [107]. Instead, the message format developed here can be used in conjunction with those message types. It may also be possible to combine the semantic-content message format with serialisation to transport complex Classad messages in the simpler message structure, still providing the same functionality as seen in Pierantoni's work.

For the original messages of SGAs, it may even be possible to encode the com-

[3] An example of this can be seen in the experiments of this thesis in section 6.5.5, page 152 where a string and an integer are combined and sent from one agent to another.

[4] This approach is inspired by the way the myriad of different technology protocols all use the TCP/IP network. For this to happen, each technology develops its own custom format, meeting the needs of that technology. However, for transport over the IP network, all data must go into IP packets.

plex Classad part of these using the Base64 technique used for binary abilities. On receiving such a message, the SGA could then decode the complex Classad message and process it. This would require a further alteration to the SGA architecture and is not explored in this thesis.

Chapter 5

Social Profiles in SGAs

5.1 Introduction

As touched upon at the beginning of chapter 1, the social and economic differences found between different types of resources and work will influence the types of collaborations and relationships which are formed between them. This will in turn, influence any subsequent work allocation decisions. As an initial step towards forming such relationships, agents need a mechanism to consider social and economic backgrounds and origins.

Recall that SGAs were developed to address the diversity and complexity found in allocation of Grid computing resources. This chapter explores a solution for addressing social and economic differences using a combination of textual descriptions known as social profiles and search engine technology. It explores using free and unstructured language to describe entities, briefly introduced in section 1.3.

The remainder of this chapter is laid out as follows. In section 5.2, the concept of 'social and economic' differences is explored and relevant examples are given. Section 5.3 discusses the meanings of words and a model of that meaning. This yields a vector space model, from which similarities between these meanings can be quantified. Section 5.4 introduces Lucene, a software implementation of a vector space model and gives some details of its workings. Social Profiles, a solution for describing and differentiating between social and economic differences is described in section

5.5. This section also gives details of an extensible prototype implementation. The remaining sections discuss search in other areas of work and resource allocation as well as some other approaches available for evaluating text.

5.2 Social Evaluations — A Discussion

A simple example of a 'social evaluation' occurs when a person or entity is asked to make (even a relatively small) allocation of one's own resources to a third party (i.e. give something to a charity or to an underprivileged individual). Unless predisposed to either an overwhelmingly negative or positive response, a quick thought process often determines if the potential donor will actually make a donation in that particular case. Questions that may arise are:

1. Does the charity or individual really need this or could they do it without my donation?
2. Will the receiver use the donation wisely or squander it?
3. Do I need this 'donated resource' more that the receiver?
4. Does my 'view of the world' (i.e. my ideology) inform me that the receiver should be able to 'exist and prosper' without my support, or otherwise?

We saw in section 2.7.3, the case of WCG [20] and SNAC [27] that human committees are used to make similar decisions. The value of the resource to be allocated may be quite high and there may be a need for accountability and transparency. In such decisions, there may be discreet, subtle but important differences between the entities in question. Reproducing this complicated and nuanced thought process in a computer program is clearly difficult, as is any attempt to imitate any form of reasoning or intelligence. For this reason, the actual allocation is often left to human reasoning and decision making.

Whether carried out by humans or machine, resource allocation decisions are often made with incomplete information. In this case, an alternative thought process used for making an allocation decision may be to establish if the benefiting entity appears to be *similar* to other entities known to the decision maker. This leads to the question "is the background, behaviour and description similar to others I know of, or to my own?"

The motivation for compiling social profiles as a means of describing actors is to contribute towards a solution of an economic and social allocation problem, but in a way which respects the requirements of owners of work and resources. These requirements include social and economic considerations and outlooks.

In the context of the vision outlined for SGAs in section 3.4, page 53, where 100s or possibly 1000s of SGAs occupy an environment, and where an agent owner is an entity such as an institution or enterprise (producing or consuming computing services), it is envisioned that the social profile associated with (and embedded in) each agent will be its social and economic description. Each SGA will start out with its own, and in time, these will be exchanged and distributed throughout the entire system as agent interactions take place.

5.2.1 Social and Economic Differences

To illustrate examples of different social and economic outlooks, different types of resources which may be suitable for addressing large computing challenges are briefly discussed below.

1. *Commercial enterprises:* A commercial enterprise invests capital in computer resources. The resources may be used internally or rented to external users. It is likely that the owner's focus will be on maximising investment return or utility for their own benefit.

2. *Public investments:* Governments invest in large computational resources for scientific and academic institutions, doing so for the long-term benefit of society. The resources are owned and controlled by the institutions. Usage may be restricted to educational and research work but society in general may benefit from the results achieved.

3. *Volunteered resources:* In volunteer computing, a member of the public allows

their privately owned computer to be used by a third party project. However, they may exercise discretion when choosing how their computer will be used. They may require, for example, that the work aims to bring a humanitarian benefit or increased scientific knowledge and they may be concerned about who will benefit from the outcome and how it will be used.

4. *Environmental factors:* Institutes, groups and corporations who are sensitive to the impact of their carbon footprint may have a preference for using computer resources which are powered (to varying degrees) using renewable energy. This criterion may be critical for both providers and users of commercial and volunteered resources [61].

Considering the cost, impact and potential benefits of accessing such computing resources, owners are likely to have preferences, aspirations and rules which will determine how their resources should be used. Owners of work may also have similar preferences. The following questions may arise:

1. Why are the resources being provided *or* why is the work being done? Who benefits and do they need and deserve those benefits?

2. What are the remuneration and cost expectations? Is the work not-for-profit? What is the financial status of the owner of the work?

3. What policies and ideological orientations exist which may influence allocations? Is there a preference for commercial, educational, humanitarian or scientific work?

Matching resources and work based on technical requirements such as memory requirements, CPU architecture and bandwidth can be done using a Boolean matchmaking mechanism such as Classad [110] or similar (as discussed in sections 2.7 and 3.2.2). However, matching based on social and economic outlooks and attempting to capture answers to some of the above questions presents a different challenge. A new approach is needed to accommodate the larger number of considerations which need to be dealt with.

5.3 Modeling and Measuring Meaning

If the ‘meaning’ of free and unstructured language in text (i.e. what the author is attempting to say or describe) can be modelled and measured in some way, this may enable the establishment of similarities or differences between entities described by that text. One approach to *modelling meaning* of such text is introduced next.

5.3.1 A Model of Meaning – The Vector Space

In spoken language, the meaning of a word is often expressed by offering other words which are ‘close’ in meaning to it. Two words, therefore, can be said to be either close or far apart in meaning. This proximity metaphor suggests that meanings have a spatial property, that they occupy a location in a space occupied by other meanings and that nearness in that space can be considered a metaphor for similarity [111].

Among others, Osgood [102] and later Salton [112], in their work to measure meaning of words and documents argue that similar entities, when described will be located close to one and other in the semantic space of the language used. Meaning can also be linked to proximity in a mathematical sense in a way that can be measured and quantified. Research by linguist Harris [75] led to the theory that meaning and distribution of words were closely related, and that the meaning of words could be indicated by the *context* in which they frequently occurred (i.e. near certain other words). In other words, words which occur near certain other words frequently mean the same thing.

The context of a word may vary in scope from a sentence to an entire document. Research by Salton [112] created a high-dimensional model which used statistical information about frequency and contexts of occurrences of words in documents. A high-dimensional vector for each document can be produced which allows it to be located geometrically in a high-dimensional vector space. This is known as the *vector space model* (VSM). Each document vector is a statistical representation of word occurrences in that document, and the context for each word is the entire document.

A Example Vector in a VSM

For a simple example of how a vector is produced, let us examine the sentence in figure 5.1. By counting the unique terms in this sentence, and assigning a frequency of occurrence to each one, a 'term vector' for the sentence figure 5.1 is generated. This is shown in figure 5.2.

> *'It was not that that misspelling of receive annoyed me, but that the misspelling of receive further down the page made me wonder if that one was an error.'*

Figure 5.1: Sample Sentence.

```
vector d =
{it, 1: was, 2: not, 1: that, 4: misspelling, 2: of, 2: receive, 2:
 made, 1: me, 2: annoyed, 1: but, 1: the, 2: further, 1: down, 1:
 page, 1: wonder, 1: if, 1: one, 1: an, 1: error, 1}
```

Figure 5.2: Vector for Sample Sentence in figure 5.1.

Weighting of Dimensions in a VSM

In this short example, it can be said that the sentence is characterised by this vector as each dimension or *weighting* in the vector corresponds to the frequency of occurrence of the corresponding terms in that sentence. A simple VSM model can be created using vectors like this one. However, some words are more common than others and some documents contain words which do not occur in others. A word which occurs in a particular document, but seldom occurs in other documents distinguishes that document from the others. To take advantage of this, a further more sophisticated weighting scheme called *term frequency-inverse document frequency* weighting (tf-idf) is used [97, p.86].

This, in effect, reflects how important a term is in a document by considering how frequently that term appears in all other documents in the index. In other words, using *tf-idf* diminishes the weighting for terms which appear in most or all of the other documents.

Calculation of the tf-idf for a term in a vector can be summarised as follows. Two statistics are needed:

1. The term frequency (*tf*), the frequency of occurrence of the term in the document, divided by the total number of words in that document. This value is often normalised to allow for documents of varying lengths (see below).

2. The *inverse document frequency*, a measure of whether a term is rare or common across all the documents in an index.

The *inverse document frequency* is the log value of the following quotient: the number of documents in the index divided by the number of documents which contain that term. The equation for calculating the *idf* value for a term *t* is expressed in figure 5.3.

$$idf_{t} = Term\ Frequency.log\frac{Number\ of\ Documents\ in\ the\ Index}{Number\ of\ Documents\ which\ contain\ term\ t}$$

Figure 5.3: Term Frequency-Inverse Document Frequency Equation.

Thus, tf-idf for a term is the product of these two values.

Long Documents

Not all documents are the same length. If simple counts of the frequency of all the terms (or term frequency - tf) are used, vectors created from longer documents will have higher values than others. This distorts how documents are located in the vector space. To counteract this, values in vectors are normalised.

Removing Irrelevant Words

In practice, some words which appear so frequently in every document contribute nothing to the measurement of meaning. Words such as *an, and, are, be, by, no, not,* *the, that, will* are typical of what are known as stop-words. At the time of vector creation, these words are disregarded because they appear very frequently and in every document, but do not distinguish any one document from another.

The 'similarity' approach provided by the VSM is widely used in internet search engines where web pages and documents are retrieved based (among other things) on occurrences of the words they contain. A VSM maps text documents to individual entries within the space (often known as an *index*).

5.4 Lucene as a VSM Implementation

Lucene [37] is a Java implementation of a Vector Space Model search engine and provides extensive libraries and an API for interacting with its VSM index.

5.4.1 The Inverted Index

Central to Lucene's architecture is the concept of an *inverted index.* This index is populated with Lucene documents and is structured similar to the index found in many books, i.e. important terms contained in a book (or index in this case) are listed, followed by the page number (or document) in which they appear in that book. This data structure has enough information to model the VSM. *Indexing* and *searching* respectively, involves the *insertion* to and *retrieval* of these documents from this index.

5.4.2 Lucene Documents

A single Lucene document represents a single (text) entity, be that a book, a web page or anything which can be considered as an 'atomic' unit for indexing and retrieval by searching. To index such an entity, its text content must be extracted and inserted into a Lucene document in ASCII format. A Lucene document consists of one or more fields. Depending on the nature of the text that a field holds, it may contain one or several *terms.* As an example, for a (digital) book, content like 'author' and 'title' are usually added to their respective fields as single terms (i.e. not broken down into smaller terms). This allows these fields to be queried and matched (similar to the Boolean matching of values offered by Classad). Longer passages such as an abstract or a main body of text are usually assigned to other fields. Such text is typically broken down (or analysed) into the many individual terms which it contains. A Lucene document can also contain binary fields which can, for example, be used to store arrays of byte data. Such fields can store a binary object such as the source PDF or MS Word file, however these cannot be searched. Numeric fields are also possible, although all number values are converted to text for indexing.

Figure 5.4 is an abstract schema illustration of a simple Lucene document. Vectors have a flat structure consisting of one or more fields (i.e. there can be no field nested within another field), and fields are 'implemented' by adding metadata to each term in the vector.

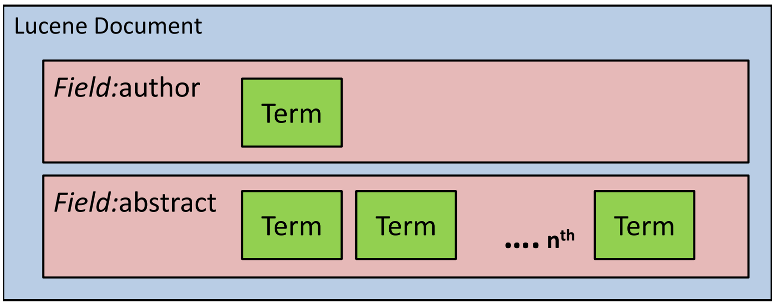


Figure 5.4: A Simple Lucene Document Schema.

For clarity, the term *Lucene document* will be used to refer to documents used in the (Lucene) VSM index. PDFs, XML and web page documents which are the source of text for indexing will be referred to by name and type.

5.4.3 Lucene Indexing Options

There are a number of options presented by Lucene when populating document fields and these impact on how documents may be searched for and retrieved. These options determine how the text of individual fields of a Lucene document are processed and can be summarised into four main groups, 'analysed', 'not analysed', 'stored' and 'Term Vectors'. The options discussed here are passed as arguments to the Lucene API when adding text to a Lucene Document for indexing.

Analysed When the `ANALYZED` option is used, the text is broken in to a stream of separate words. Each word is enriched with some metadata (e.g. which includes positional data of the word in the document and what field it is in). When a word is processed like this, it becomes a known as a token. This approach is usually suitable for large bodies of text as each token becomes searchable.

Not analysed Where `NOT_ANALYZED` is chosen, the contents of the field are indexed but are not broken into tokens. The entire content of the field is treated as a single token and is searchable in this form. This allows 'exact match' searching on strings such as personal names and multiple word strings such as a book title. Such a string is also known as a token.

Stored When the `Field.Store.YES` option is used, the entire string is recorded in the index and may be retrieved as this. When this option is used, the resulting index files are roughly the same size as the total size of the documents indexed. Where space is a consideration, use of the `Field.Store.YES` should be minimised.

Not Stored When the `Field.Store.NO` option is used, only the tokens extracted during analysing are recorded and the original string is not retrievable.

A combination of these options may be used, depending on the user's requirements. For example, for large bodies of text, it is common to analyse but not store content. Alternatively, for name, titles and other strings unique to a document, storing content in an unanalysed state may be more useful [97, p.43].

Using the `ANALYZED` option allows Term Vectors (see figure 5.2, page 91) to be created and stored in the inverted index. Using this option allows all the unique terms of a document to be retrieved at search time.

5.4.4 Querying the Vector Space for Documents

The vector space model or index is searched by creating a vector using a number of words or terms known as a *query* and then finding other vectors similar or 'near' to this. A query usually consists of a small number of terms, often two or three.

To retrieve documents in a search of the index, a *score* is calculated to reflect how similar these documents are to the query used. This calculation uses the values assigned to the terms (or tokens) in the vectors in the index, i.e. the *tf-idf* value for terms. As each document and query are represented as vectors, the score for each document is the sum of the term weights of all the terms in a document which match terms in the query. Therefore, a document which contains most or all of the query terms will have a high score. Documents which contain the query terms a number of times are returned with even higher scores. Matches on rare terms score higher due to the use of *tf-idf* values as dimensions in the index vectors.

Other factors can also influence scoring, for example using user defined setting to 'boost' scores for certain terms or documents. More complete details of how Lucene scores are calculated, including formulae and variables used, can be found in the Lucene documentation [97, p.86].

It should be noted that this approach of measuring similarity between documents only yields a similarity measurement; it does not offer any form of classification other than to differentiate it from another document relative to the query.

Lucene Query Searching Options

Lucene provides several options when creating a query for searching an index, all of which aim to improve the accuracy of document retrieval. Some of these are discussed here.

Term Query: This is the most fundamental type of query and consists of a field name and one term. For example, a book, this may be `("author", "lavin")`. If a field is indexed as `NOT_ANALYSED`, this type of query is useful for retrieving documents by a unique term from such a field.

Numeric Range Query: If data is indexed as a numeric field, documents can be retrieved based on numeric ranges within the data indexed.

Boolean Query: A Boolean clause can be used on its own or incorporated into another query and combined to filter results. An example of a Boolean clause for a book is that its `yearPublished` field 'must' be 2012. Note that this does not use a similarity score, but instead uses a Boolean type evaluation to determine if a result is returned.

More-Like-This Query: This query is created by extracting high frequency terms from an already existing document in the index. Parameters are used to determine how many terms are used from the subject document.

Other Queries: Other query types provided by Lucene include Prefix and Prefix Range which match documents with terms beginning with a specified string. Wildcard queries allow the characters `*` and `?` to be included in a query term. This can replace many or one unknown characters in the required query term. Lucene also supports a large number of 'contributed' query types generated by the user community.

5.4.5 Cosine Similarity

Vector space models, where there are more than *three* dimensions are difficult to visualise or represent graphically. However, visualising two vectors in a simple two dimensional plane intuitively tells us that the angle between them can be measured. For example, if the angle is small, the vectors can be said to be nearby each other.

The motivation for locating document vectors in a geometric space is to be able to find other vectors (such as queries) which are nearby in the space. The most common mathematical method for calculating 'nearness' of document vectors to query vectors is to evaluate the cosine angle or *cosine similarity* between them. In the context of measuring vectors, this is a measurement of angular orientation and not of magnitude. The calculation for the *cosine-similarity* between a query and a document is expressed in equation 5.1,

$$cosine\ similarity = \frac{Vector(q) \cdot Vector(d)}{|Vector(q)|\ |Vector(d)|} \tag{5.1}$$

where $Vector(q) \cdot Vector(d)$ is the dot product of the two vectors, and where $|Vector(q)|\ |Vector(d)|$ are the Euclidean norms of those two vectors.

Equation 5.1 can also be applied to the vectors of two documents, providing a similarity measurement between any two documents in an index. This is useful as it allows the similarity between a given document and a number of others to be compared. `Lucene-core` and `commons-math` libraries provide the necessary classes to calculate cosine-similarity between two documents in a Java application. The documents must be present in the index and only fields which are indexed using the `TermVectors.YES` options[1] can be used for this calculation.

A cosine-similarity requires two vectors. Recall from sub-section 5.3.1, page 91 that the dimensions of a vector in an index are based on *term frequency-inverse document frequency.* For a cosine-similarity calculation between two documents, the dimensions of the vectors need only be the frequency of occurrence of the terms in those documents. One or a number of specified fields can be used for calculating cosine-similarity. The procedure for creating these vectors and calculating their cosine similarity is as follows:

1. Using the Lucene API, a list of the terms, which are found in the specified fields in all documents across the entire index is compiled. In this example, two fields will be used as it is a more complex method when compared to calculating similarity for just a single field. Lucene provides a separate list of terms for each field found in the index. Therefore, in this case two lists are compiled.

[1] An indexing option which stores a simple term vector, without any tf-idf weighting.

2. Iterating over the elements of these lists, the terms from both lists are added to a single hash-map. As there is likely to be common terms in both lists, the size of the hash-map will be smaller than the sum of the two list sizes.

3. Next, using an `NOT_ANALYZED` 'name' field from the two documents to be measured against each other, the document ID number of the two documents are obtained from the index (Lucene API also provides methods to do this).

4. Using these ID numbers, Lucene provides a *Term Frequency Vector* for each field for each document. Each of these vectors is essentially a collection of term-frequency pairs for that individual document [97].

5. For each of the Term Frequency Vectors returned, Lucene provides two arrays, one an array of strings containing all the terms in that vector, the other a corresponding array of integers which contains the frequency of the occurrence of those terms.

6. Iterating across the elements of the array of terms, the terms and their frequencies are added to a new vector (an object of the `OpenMapRealVector` in the Java commons-math libraries is used for this). Where terms are found in more than one array (i.e. in more than one field in a document), their frequency is added to any previous frequency value present for that term. These vectors are then normalised to remove the effect of longer or shorter documents.

7. The output from this process, having processed both fields in both documents is *two* new normalised vectors which contain the terms and their occurrence frequencies for both documents. The angle between these can be calculated using the formula in figure 5.1 on page 98. In practice, only the cosine of that angle is used, hence the expression *cosine similarity.*

5.4.6 Lucene Scalability and Efficiency

Lucene is an efficient and scalable implementation of a vector space model, allowing it to be used on a global, local and personal scale. For example, tests involving Lucene, some indexing up to 100 million web pages [84] and 10 million books [56] have been carried out. These tests were carried out using ‘Nutch’ [38], an open-source search engine which uses Lucene for its index.

As an indication of the efficiency of the structure of the Lucene index, when text is indexed, and if `Field.Store.NO` is used, index files are about 30% of the size of the source text files [37, 84]. If the content of a document is stored, the index file will be roughly equivalent in size to the content indexed.

Although widely used in smaller web search engines, Lucene is also used in some commercial and internet-wide scenarios. A further example of Lucene’s scalability is its use in Twitter, the micro-blogging website [39]. Using a customised version of Lucene, up to 250 million ‘tweets’ and queries have been processed in a single day. Measurements taken on this work show an average latency of 50 milliseconds or less [120, 57].

Disk-storage of the index is partitioned into ‘segments’. Where these are large, there can be more than one, each one known as a ‘shard’. The maximum number of unique terms in a segment is 274 billion[2].

5.5 Social Profiles

To recap, the overall motivation for defining social profiles suitable for integration into agents is to aid resource allocation and to facilitate establishing relationships and organisations in a way which respects the requirements of owners of work and resources. This research proposes exploiting the measurement of similarity between documents which describe entities (in this case consumers and providers of resources) and to explore if it is a suitable and feasible way to find similarity between such entities.

For each entity, a collection of text which is compiled either by or about that entity is created. This description is called a *social profile* and aims to capture any aspect of an entity which can be described using text. This in effect allows the full

[2] Source: Lucene issue posting at https://issues.apache.org/jira/browse/LUCENE-2257. This figure was increased from 2.4 billion in response to an issue raised.

resources of language to be used as opposed to, for example, a restrictive set of key-pair values or a questionnaire. Each textual description document becomes an entry in the VSM index and is a statistical representation of a social profile there.

5.5.1 Content of a Social Profile

By way of discussion, consider the scenario where an e-Commerce company needs to create descriptions of all its sellers in order to present them to potential buyers. An example of such a company is eBay, the internet auction site [40]. A potential buyer is presented with a collection of feedback comments collected from past customers after previous transactions.

Taken in total, this is in effect a description of the seller. It consists of a short input where the seller describes itself. Appended to that are several short comments supplied by past customers (this can be called 'reciprocated' or third-party information).

In the case of eBay, reciprocated information has a structure imposed on it in the form of a list of questions and a short Likert scale[3] [92] dialogue about promptness of delivery and quality of packaging. What is interesting about this total collection of text is that it (automatically) contains words which describe the seller. If this were indexed in a VSM, statistical information about occurrences of 'superlatives' and other positive terms should differentiate a good seller from one who frequently has negative feedback written about them (however in the case of the eBay website, this is not done, instead they are simply presented to the potential buyer in summary form).

By contrast, another example of a 'product' offered to 'buyers' (by the film making industry in this case) is a feature film. The buyers here are people who wish to go to a cinema to view such a 'product' for their entertainment; they also need a description when deciding on which if any film to view. Film reviews also constitute a textual description but tend not to have a structure similar to that of the eBay feedback

[3] A psychometric scale often used in questionnaires, allowing users to select answers on a scale such as "strongly agree - agree - disagree - strongly disagree".

details discussed above. Reviewers can write whatever they feel is appropriate and there are several spectrums along which to describe a film (for example directing, acting, plot, etc.). However, the aggregation of even a number of film reviews cannot be said to have the same usefulness as an eBay feedback description as personal tastes (among other things) play such a large part in what is a satisfactory film viewing experience. Feedback to eBay sellers and film reviews are very different, one having some structure, while the latter is more likely to be an unstructured and amorphous passage of text.

The origin of the text is also important; and descriptive text from both the entity itself and third parties both have a place in an eBay sellers description, a film review and a social profile. In the eBay information discussed above, positive information from a third party is difficult to fake, and negative reviews are difficult to remove; this adds authenticity to any description. In a social profile however, information which comes directly from an entity is also important as they are often best placed to set out what their objectives are and to provide an overview or summary of their mission.

5.5.2 Structure of Social Profile

The question arises; what type of information is relevant when describing computing resources and work and what type of structure should it have? For this research, a broad view is taken, insofar as any text which applies to an entity contributes to a description of it, and therefore could aid a resource allocation decision.

As seen in the eBay example discussed above, judicious use of Likert scales and structured questions is useful. However, it should be remembered that in that instance, the reason for compiling a description of a seller is to allow a potential buyer to evaluate whether a successful transaction is likely to be completed with that seller or not. Using that definition, the eBay description may be quite narrow and does not need to describe a seller beyond that criterion.

The structure of a social profile needs to consider a broader set of criteria. It is not feasible to pose the many potential questions arising when trying to describe an entity under headings loosely described as ideological, political, economic and sociological. For this reason, at least to begin with, the structure of a social profile must rely mainly on collecting descriptive text. This text may then be indexed under a relatively small number of categories which are dictated by the origins of the text.

For these reasons, the initial prototype social profile contains three sections:

1. Fixed and incontrovertible information or 'logical data', such as the creator's or owner's name, address, type, URI, address or location.

2. The creator or owner's input, i.e. what it claims to be a true description of itself, its current and past activities and publications.

3. Third party (reciprocated) information and references, reviews, feedback, replies in blogs and forums.

The rationale for this structure is that its different parts can be mapped to separate 'fields' in a VSM entry (i.e. a Lucene Document). Data in the first section is indexed with each item as a separate field but is not statistically analysed for context and frequency. This allows each item to be queried and retrieved by exact matching of these terms. The second and third sections consist of large bodies of text and are each indexed in two separate fields. The text is statistically analysed, each becoming part of the vector for the social profile.

Collating Content for Social Profiles

In a real world application, it is envisaged that an owner or creator will have full control over the content of the owner's input. It would be reasonable for this owner to craft this part in a way which maximises themselves, all the while being careful not to exaggerate details or be untruthful. This expectation is similar to how an media advertiser uses the allocated time or space very carefully to maximise their offering, all the while staying within the laid down guidelines on advertising standards. For the reciprocated section of a social profile, the entity described is at the mercy of what others say about it. It is envisaged that the owner or creator would have no input into this section. In the actual collation of third party content, the questions of transparency and trust which would naturally arise in doing this are not considered at this stage. These concerns may be addressed at a later stage. For example, it may be necessary that all social profiles are published and digitally signed, as well as publishing references to the various sources used.

5.5.3 XML as a Social Profile Container

Chapter 4 outlines how separate developments in creation, deployment and configuration of SGAs use XML files and how this necessitates a HTTP server in the MAS environment. These factors, when combined with the human readability and extensibility makes XML a logical choice of *container* for creation of social profiles. Java implementations of XML parsers are ubiquitous and easily integrated in to the SGA development environment and to SGA abilities.

Transport and Indexing of the Social Profile XML file

Using the same infrastructure discussed in chapter 4, a completed social profile is transferred to the HTTP server within the MAS. A message containing the URL of this XML file is sent to the agent and used by the agent to fetch its social profile from the HTTP server. Lucene software can be incorporated into SGA at creation stage. Using this, the XML is parsed and each element is added to a Lucene document and subsequently added to a VSM index within that agent, using the underlying disk storage of the agent container for persistence. At this point, an SGA has a single document in its Lucene index which is in effect a high-dimensional vector representation of a textual description of its owner.

5.5.4 Extensibility of A Social Profile

For consistency throughout an MAS, it may be desirable that the overall structure of the social profile is not changed dramatically. However, exploiting more of the potential and flexibility of the Lucene document and search API may be useful. In the case of VSM entries or documents, at a minimum, it will be required that the number of fields be able to be increased without too much disruption to the agent environment or it configuration. Although the social profile structure outlined in sub-section 5.5.2 may suffice for a prototype, extensibility is also a required feature of any implementation.

In the social profile described in sub-section 5.5.2, the structure of the Lucene Document which is indexed in an agent is mapped directly from the structure of the XML. It is also the case that the logic coded into the agent ability used to index this information has to match this structure. For the agent and profile creator, this in effect means that adding further elements to an XML file requires a change in the logic available within the agent. In the current SGA architecture, this means that a new binary *ability* needs to be developed and added to the agent (i.e. a reconfiguration of the agent).

While this can be done using the mechanism described in section 4.2, it is restrictive and does not lend itself to flexibility and extensibility of a social profile.

5.5.5 An Extensibility Solution

The solution proposed to overcome this restriction involves being able to define a schema for a social profile XML file and including it within that same file. A typical XML schema is not suitable for this as such a schema would still have to be 'translated' to actual Java binary code to implement the logic needed for the additional fields.

This research proposes an *encapsulated binary schema* in the form of a Base64 [81] serialised Java class. This is added as content of an element in the XML file by the creator of the social profile when the social profile is being compiled. During the indexing operation within the agent, this class is deserialised and used to add the contents of each XML element to the Lucene Document.

This approach is possible for a number of reasons. A Lucene document has a non-hierarchical or *flat* structure. Therefore, an XML file which reflects this can have a similar flat structure. This means that the elements in the XML file which contain descriptive information can be parsed and used to create a list. Each item in that list can then be used to call a corresponding method in the Java class. Access to these methods is achieved using Java Reflection. As the methods of the class are invoked, the contents of the corresponding element in the social profile XML file is added to fields in the class, and subsequently added as a field to the Lucene Document. The overall sequence for social profile creation, and its subsequent indexing are outlined below.

Sequence During Profile Creation

In the development environment of the agent and social profile creator, the following takes place:

1. The social profile XML file is compiled.
2. The Java class is created, having a set of methods which meets the indexing requirements of the created profile.
3. This code is then compiled, whereupon the resulting `.class` file is converted to a Base64 string.
4. This string is added to the social profile XML file (under a separate element to the textual descriptions part).

The names of the methods in the Java binary must match the names of the elements in the social profile.

Sequence During Indexing

Within the agent, the sequence for processing the social profile XML file using the *encapsulated binary schema* is illustrated in figure 5.5.

This sequence can be described as follows:

1. The Base64 string representing the `ProfileStructure` class is parsed from the social profile, deserialised to become a Java object and is instantiated.
2. An empty Lucene Document is instantiated and passed to the class using a setter method.

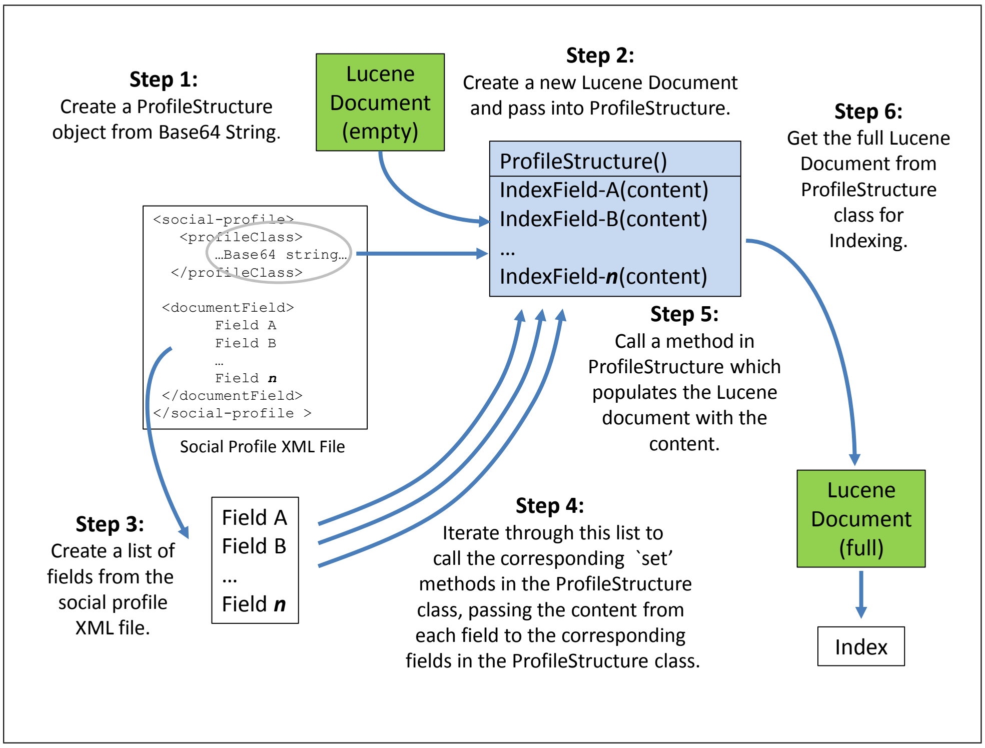

Step 1:
Create a ProfileStructure object from Base64 String.
Lucene Document (empty)
Step 2:
Create a new Lucene Document and pass into ProfileStructure.
ProfileStructure()
IndexField-A(content)
IndexField-B(content)
…
IndexField-n(content)
Step 6:
Get the full Lucene Document from ProfileStructure class for Indexing.
<social-profile>
<profileClass>
…Base64 string…
</profileClass>
<documentField>
Field A
Field B
…
Field n
</documentField>
</social-profile >
Social Profile XML File
Step 5:
Call a method in ProfileStructure which populates the Lucene document with the content.
Step 3:
Create a list of fields from the social profile XML file.
Field A
Field B
…
Field n
Step 4:
Iterate through this list to call the corresponding `set' methods in the ProfileStructure class, passing the content from each field to the corresponding fields in the ProfileStructure class.
Lucene Document (full)
Index

Figure 5.5: Sequence for social profile indexing using a class encapsulated in the XML file.

3. The XML file is parsed. The names of the elements in the social profile which contain the descriptive information are then extracted and compiled into a list.

4. Iterating over this list, corresponding setter methods in the `ProfileStructure` object are called, adding the contents of each element to the fields or variables within it. A further method in the class is then called to add fields to the Lucene document using the values of the class fields.

5. On completion of this process, the agent indexing *ability* calls a getter method on the ProfileStructure object which returns the (now populated) Lucene Document. At this point, it is then added to the agent's index.

The result of this development is that the number of fields in a social profile can now be increased or decreased without making corresponding changes to the indexing ability in the agent. The encoded Java binary is, in effect, a schema for the social profile and comes with the social profile. It should be pointed out that, as well as meeting the requirements of a changed XML schema, it can also contain code which can be tailored to match the exact arguments and options required for indexing the different type of fields contained in the XML elements.

5.6 Search in Other Areas of Work Allocation

Where this thesis examines using search and the vector space model to differentiate between social and economic differences, other researchers have also explored using various forms of search technologies. Most of these are in the area of resource allocation too and some are summarised here.

Research by Gorodetsky in [73] uses matching libraries embedded in P2P agents in a multiple-agent system which incorporates searching and matching of other agents. Services are defined using a key-pair descriptions and these are searched and queried for service discovery and matching. This work uses pattern matching to search the expressions used in the matching mechanism but does not incorporate any other search technology which uses statistical data about occurrences of words or terms.

Montella in [99] describes the integration of a statistical language model called Latent Semantic Indexing (LSI) [62] with the Classad matching tool. LSI is frequently used in information retrieval and internet search engines. In this work Classad and LSI are used in a two stage process, where initial matches returned by Classad are optimised using a customised LSI algorithm. LSI is a similar search technology to the solution used here in how it uses terms and vectors, although vectors are created differently. However, unlike Lucene, it is not freely available as an open-source project. Montella's approach only applies it to terms used in the matching tool's expressions but not to any additional text as proposed here.

Hao [74] uses vector space model technology (VSM) [112] to rank the relevance of web services where large numbers of responses are returned during service discovery. This work uses textual descriptions of web services. Similarity measurements between two or more of these are combined with measurements of the level of connectivity a web services has to others. By doing this, a measure of the relevance and importance of a given web service is produced.

5.7 Other Methods of Evaluating Text

5.7.1 Sentiment Analysis

Sentiment analysis is the extraction of information about subjective and changeable opinions, evaluations and attitudes with respect to a given topic [104, 105]. It can be described loosely as the 'state of current thinking' about a given topic, with the output of the analysis being positive, negative or neutral. Examples of topics which are frequently analysed are economic outlook and consumer sentiment, but the same techniques may be applied to any topic, for example how a new film release or restaurant is received by the public. The source of the content used may be any digitally available text such as new articles, blogs and comments from the public.

Sentiment analysis is carried out using a number of methods but these mostly centre around a technique known as *Natural Language Processing*[4] (NLP). Similar to some search technologies, the occurrence of words is important; however, NLP uses a deeper level of analysis than counting occurrences of words and recording their frequencies in an index. For example, relationships between words, sentence structure and grammatical constructs are considered.

5.7.2 Fuzzy Logic

Fuzzy Logic is a form of logical reasoning which deals truths which can be expressed true, false or somewhere along a continuum between 0 and 1 (i.e. partially true or false) [41]. It is useful where there are unclear (or fuzzy) boundaries between different classes of entities. Because of this, it has found some applications in linguistics and is used in conjunction with many techniques in that field. One such example described by Ordoobadi [101] uses Fuzzy Logic for selecting suppliers for an industrial process. This work exploits the idea that subjective human beliefs can best be expressed linguistically, without the limitations of numerical scales (similar to this thesis). In a somewhat manual process, a 'master' set of criteria is selected by the consumer,

[4] Natural language processing is concerned with the science of using a computer to understand and derive meaning from language which is written or spoken in its natural state.

for potential suppliers. Each criteria is described linguistically using terms which describe attributes like ‘service’, ‘delivery’ and ‘produce quality’. NLP tools are then used to evaluate each supplier, leading to a selection of the one deemed to be most suitable. However, unlike social profiles, the number of attributes which can be described is limited and the process is not internet based or distributed.

5.7.3 Relevance to Social Profiles and SGAs

Both sentiment analysis and fuzzy logic may, at a later time be applied to the contents of social profiles. For example, similarity measurements between documents, along with other measurements or values, may be used as part of fuzzy logic determinations.

Although it is (currently) intended that social profiles describe aspects and outlooks of entities which are largely fixed, in a increasingly changeable digital environment, sentiment analysis may have a role to play in determining ‘sentiment’ in a large system of ‘social agents’. The flexible and extensible nature of a social profile as described in section 5.5.4 facilitates the inclusion of a field in a social profile which may describes sentiment towards an entity. However, given that sentiment can rapidly change, and that change in sentiment is the main focus of analysis, consideration would need to be given to the ‘freshness’ of social profiles for this approach to become effective and useful.

5.8 Summary

In this chapter, the overall concept of describing an entity using a social profile was introduced along with a mechanism used for finding similarity between them, namely the *vector-space model.* The content of such a profile was also discussed.

Lucene, a Java implementation of a vector-space model was also introduced, along with a short investigation into its workings and capabilities. Finally, considering that a social profile will be indexed by a binary ability (which, recall, is to be loaded to the agent), a solution which allows for flexibility and extensibility of a social profile is proposed.

At this point, all the pieces are in place for testing of all the proposed mechanisms. The next chapter examines the concepts of chapters 4 and 5 by conducting several experiments which combine the concepts proposed thus far.

Chapter 6

Usage Experiments

6.1 Overview of Experiments

To evaluate the contributions of this thesis, seven experiments are described in this chapter. To recapitulate, this thesis makes contributions in two main areas:

- Mechanisms which allow SGA technology to be configurable and scalable without the need for them to repeatedly stopped, rebuilt and restarted each time they need to be reconfigured.

- Integrating search technology with SGA so that descriptions of agents can be used in resource allocation decisions.

The remainder of this section summarises these experiments and how they relate to the two areas examined in this thesis. This includes a table of outcome summaries. Section 6.2 details the environment in which these experiments and their development were carried out. Each subsequent section is dedicated to one of each of the seven experiments.

6.1.1 Experiment Summaries

Experiment 1: MAS Creation Functionality In Experiment 1, the usefulness of the multiple-agent system creation mechanism described in chapter 4 is examined. The task of recreating an existing system of SGAs from the 2008 work of Pierantoni

[107] using that mechanism is examined. Success in recreating the original system means that the mechanism proposed here is able to recreate a system which is at least as complex as that created using the 'hardcoding' approach used in the 2008 work.

Experiment 2: Indexing, Retrieval and Similarity This experiment examines using search technology for measuring similarity between entities. Textual descriptions of a number of entities from different categories are indexed and retrieved to determine if the *similarity* measurements obtained reflect known similarities between them. Being able to reliably measure similarity is an essential part of using such measurements in resource allocation decisions.

Experiment 3: Integrating Search into SGAs The techniques for indexing and obtaining similarity measurements examined in Experiment 2 must be integrated with SGAs if they are to be used in resource allocation decisions. This experiment examines what is required for this to take place. The techniques used to achieve this in a distributed agent environment are also examined and evaluated.

Experiment 4: Using Similarity for Resource Allocation This outcomes of Experiments 3 and 4 are brought together and 'put to work' in Experiment 4. Similarity measurements are combined with (monetary) price data and are used to make a simple resource allocation decision. Although only a proof of concept, this experiment demonstrates how this may be done. This is the principle motivation of integrating search technology into SGAs.

Experiment 5: External JARs Over HTTP In order to support the 'search' abilities configured in the SGAs in Experiments 3 and 4, additional JAR files had to be inserted into the *basic-agent*. It could be argued that having to add additional JAR files to the *basic-agent* each time 'new' functionality is required renders the mechanism examined in Experiment 1 somewhat limited, if not 'broken'. This experiment introduces a technique to allow additional JAR files to be made available to an SGA without having to alter the make-up of the original *basic-agent*. Success in doing this enhances the configurability and scalability of SGAs in a way that does not require the *basic-agent* to be altered (and hence does not require an agent to be stopped and rebuilt when adding 'new' functionality).

Experiment 6: Using Included URLs in Abilities XML Files Further to the configurability of SGAs, section 4.3.2 proposes an abilities configuration file (ACFs) which contains a number of serialised binaries (agent abilities) and which 'include' one or more URL to link to ACFs. In Experiment 6, the functionality of these additional 'included' abilities configuration files is evaluated as well as the possibilities which flow from their successful use.

Experiment 7: Profile Structure Class For the purpose of making indexing of textual descriptions scalable and flexible, section 5.5.4 proposes an extensibility mechanism which allows a social profile XML file to encapsulate the schema of the social profile file and the 'logic' needed to index that file's content. Experiment 3, among other things, tests its functionality. Experiment 7 examines to what extent the flexibility offered by this mechanism can be further exploited to enhance the indexed description of an SGA, using, for example, information from the underlying host system of the SGA.

Table 6.1 presents a summary of the outcome of these experiments.

Number	Experiment Name	Outcome	Comments
1	Examine SGA MAS creation functionality	Identical functionality and behaviour to the hard-coded system achieved.	Novel technique for loading class instance implemented.
2	Indexing, retrieval and similarity	Statistically significant measurements obtained for 3 of 4 categories.	Positive results, categories successfully distinguished from one another.
3	Integrating search into SGAs	Distributed search functionality and similarity measurements successfully tested.	Serialisation used for transport of Lucene objects.
4	Using similarity for resource allocation	Successful prototype use of similarity measurement for resource allocation in an SGA.	Brings outcomes of Experiments 1, 2 and 3 together.
5	External JARs over HTTP	Ability successfully tested which depends on a JAR not available in an SGA.	Uses URL class loader and Java Reflection, further enhances scalability and flexibility of agents.
6	Using ‘included’ URLs in abilities	MAS of Experiment 1 repeated using multiple chained abilities configuration files.	Enhances scalability and extensibility of the agent configuration mechanism.
7	Profile Structure Class	Scalability and flexibility of encapsulated binary schema successfully tested.	Descriptive and local system collected and added to social profile.

Table 6.1: Experiments and Outcomes

6.2 Environment for Experiments

The environment used for experiments in this chapter are outlined in table 6.2. This will remain the same for this chapter unless otherwise stated.

Resource	**Details**
Operating System	CentOS release 5.8 (Final) Linux version 2.6.18-348.3.1.el5 gcc version 4.1.2 20080704 (Red Hat 4.1.2-54) #1 SMP Mon Mar 11 19:39:25 EDT 2013
Java Programming Language	Java version "1.6.0_22" OpenJDK Runtime Environment (IcedTea6 1.10.10) (rhel-1.28.1.10.10.el5_8-x86_64) OpenJDK 64-Bit Server VM (build 20.0-b11, mixed mode)
Development Environment	Eclipse Java EE IDE for Web Developers version Indigo Release Build id: 20110615-0604
Maven Build System	Apache Maven 2.2.1 (r801777; 2009-08-06 20:16:01+0100)
Lucene Search Libraries	lucene-core, lucene-queries version 3.5.0
Math Libraries	commons-math version 2.2
Development Computer	Dell XPS, 4GB RAM, Intel(R) Core(TM)2 Duo CPU, 3.00GHz
Cloud Virtualisation	OpenNebula version 3.8.3
Web Service Container	Apache Tomcat version 7.0.14
Statistics Software	Minitab version 16

Table 6.2: Details of Development Environment Used

6.3 Experiment 1: MAS Creation Functionality

The aim of this experiment is to further test the mechanism described in chapter 4. To recap briefly, in the original work of Pierantoni in 2008 [107], an SGA was created and packaged in an IDE, and subsequently deployed to its working environment, i.e. a web services container. All binary abilities needed for its operation were loaded to the internal data structure (its hash-map) during the execution of the internal code as the agent started running. This meant that any further changes needed to those binaries required the source code of the SGA to be modified, thereby requiring re-compilation and re-deployment once again in the web service container.

In chapter 4, section 4.3 discusses developments for SGA deployment which allowed larger numbers of them to be deployed and configured, each independently and without impact on each other.

Chapter 4 also discusses a proof-of-concept of an approach which aims to get away from the need to hard-code all the required abilities in an agent at development time. Instead, it aims to be able to add abilities after the SGA has started running. The motivation for achieving this is to remove the need to stop, edit and recompile each agent's code whenever a change to that agent is necessary.

To test this approach, the same system as was examined in section 4.6, page 65 will be fully recreated. But, instead of hard-coding each agent with its required abilities, this experiment will attempt to reproduce the same system of agents by creating a number of 'basic' or 'minimally configured' agents. The required binary abilities will then be added to them after they are deployed and 'up and running'. This will be done using the framework described in chapter 4.

Further details of this system of SGAs are discussed later in sub-section 6.3.1.

Hypothesis and criteria for success or failure: The hypothesis being examined is whether the approach of serialising, transporting and loading of binaries in already running agents is flexible and extensible enough to allow the sophistication of the original SGAs to be reproduced. This will be tested without the classes of the binary abilities being actually hard-coded in the agent from the outset: instead they will be transported and loaded to the agent after it has started running. The criteria for success or failure of this experiment depends on positive outcomes to the following two questions:

1. "Is the configuration of the recreated SGAs (achieved using the mechanism of this thesis) the same as that of hard-coded original SGAs?"
2. "Can the same functionality be demonstrated in the recreated system of this thesis as can be seen from the original hardcoded system?"

It should be noted that the way the abilities are added is the only significant difference between the agents created using the code from the 2008 experiments and what is being created here.

6.3.1 Description of the System to be Created

In a real-world situation, the constituent parts of an agent system to be developed (or perhaps added to an existing system) would most likely be planned, prepared and tested in a development environment or IDE, especially in the areas of functionality and desired behaviour. For the system of agents recreated here, this planning and testing was effectively done in the work of Pierantoni [107] in 2008. No functional or layout changes will be made to the system of agents created then. This same system of agents, which will be recreated in this experiment is illustrated in figure 6.3.

Roles and Relationships of SGAs

As mentioned in section 3.2, page 46, the roles and relationships between SGAs are defined by binary abilities and adding these abilities correctly and successfully is the aim of this experiment.

SGAs which communicate with others, requesting services (and the price of those services) are termed social agents. SGAs which produce these services are called production agents. In the system to be recreated here, a "social agent" can control "production agents", the 'utility' of which can be exchanged and traded. There are

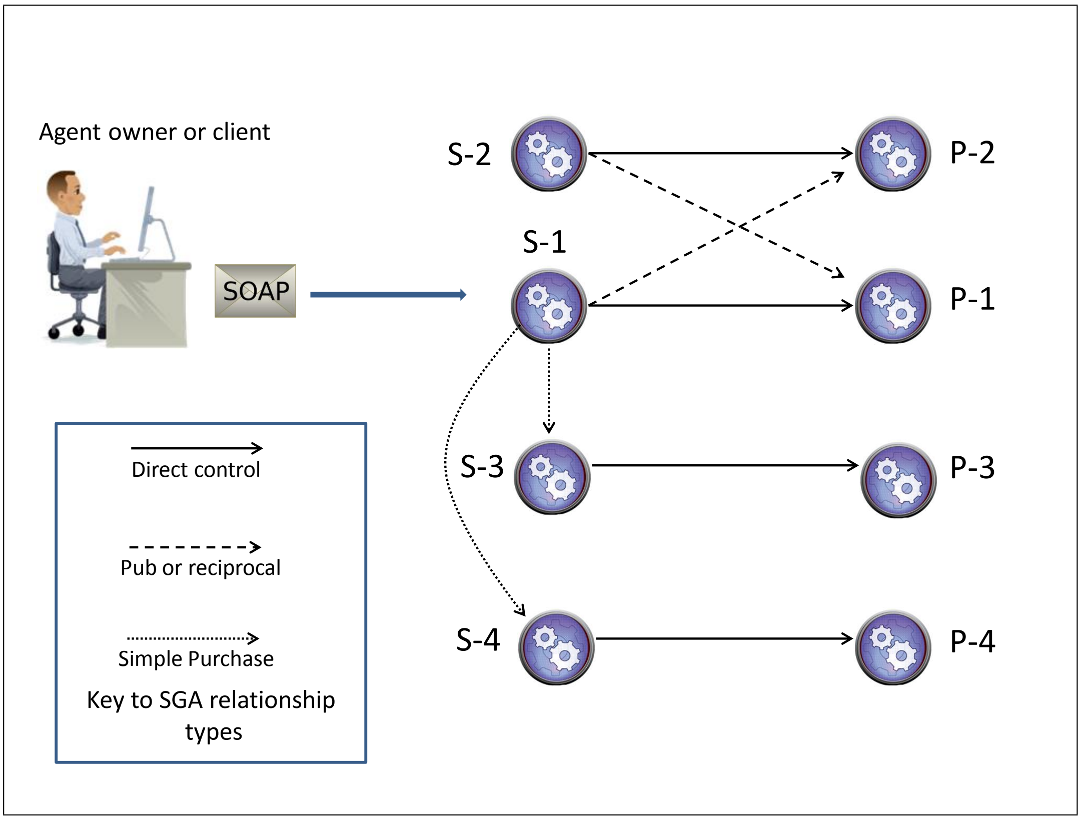


Figure 6.3: Layout of the eight SGAs to be recreated using the MAS creation mechanism.

three different types of relationships found between social and production agents in this system recreated here; these can be summarised as follows:

1. Simple Producer: One agent directly controls another, usually a social agent controlling a producing agent. A controlling agent simply instructs the controlled agent to carry out a task.

2. Simple Purchase: Two social agents, one buying, the other selling a service, exchange messages to determine if a purchase can and will take place. The seller sets the price and if the purchaser values the service at that price or higher, the transaction takes place.

3. Pub Producer: this is a sharing or 'reciprocal' arrangement between a social agent and a producing agent. A pub relationship is implemented by adding two 'pub-provider' abilities to the two participating social agents and their opposite

producing agents (see the key and details of figure 6.3).

The roles and relationships of the agents shown in figure 6.3 are summarised as follows:

1. The Client program is located outside the MAS. Its only function is to send messages to S-1 and it is envisaged that this program would be operated by an agent (human) owner or controller.

2. S-1 has direct control over P-1, simple purchase relationships with S-3 and S-4. Furthermore, it has an ability which represents one half of a pub relationship with P-2 (which is directly controlled by S-2). It also enforces a *policy* which dictates what order it selects the options available to it, e.g. it uses P-1 first, then 'pub' agents available to it, finally resorting to purchasing resources from S-3 and S-4.

3. S-2 has the other half of the above pub relationship with P-1, as well as a direct control relation with P-2.

4. S-3 and S-4 directly control the remaining two agents, P-3 and P-4 respectively. These agents accept jobs from S-1 in a simple purchase transaction, subject to their pricing policies being met.

5. Production agents P-1 to P-4 all accept jobs from their respective controlling agents subject to their policies. For agents P-1 and P-2, a further *authorisation* policy is enforced based on the status of the pub arrangement between them.

6.3.2 Experiment Environment Considerations

Host Container IP Addresses as State

In chapter 4, consideration was given to setting state in instances of classes prior to loading them to the SGA's data-structure. In this experiment, some of the state which has to be added consists of web service addresses which are used for communications between SGAs. These consist partly of the IP addresses of the hosting the containers in which the agents run. For this experiment, temporary VMs in Cloud infrastructure are used, the IP addresses of which are not known until after each VM is created. When available, these values are manually inserted into a Java *variables* file and used by a custom Java program when creating the serialised object files (recall section 4.7). Therefore, in this experiment environment, the IP addresses for each SGA must be known before any of the serialised object or abilities configuration files can be created.

As touched on briefly in section 4.1.1 on page 55, planning the layout of a multiple-agent system is not the focus of this thesis. The workings described are merely the mechanisms needed to create the necessary serialised object and ACFs necessary for recreating and configuring the required SGAs. Furthermore, in a real-world environment, IP addresses for the machines to be used would most likely be known and fixed. However, at the center of this mechanism are the *tools* which could be used in *any* mechanism used to create such abilities configuration files.

Development vs. Real Environment

In the original SGA work [107], this small system of agents tested the behaviour of SGAs in a simulated grid-job submission scenario. Depending on the availability of suitable infrastructure and available grid services, this experiment can be run either in a software development environment or with SGAs deployed in web-service containers in virtual machines (VMs).

For this thesis, I have used a development environment for testing and prototype work. After initial developments in an IDE, agents and abilities were sometimes tested in a web service container on the development machine (see section 6.2 for details of this environment). For this experiment (and all those of this chapter), each SGA is created as a web-service application. Each one runs in a separate web-service container, all of which are located on separate virtual machines. This was done to recreate a truly distributed environment where communications required sending ‘real’ message via HTTP over an IP network. Working in a ‘real’ distributed environment aims, as far as possible, to eliminate the possibility of classes or JAR files being inadvertently made available to a class loader in a scenario where, for example, 'class loading' is the actual functionality being examined. This proved to be a fruitful exercise as some of the issues addressed in this thesis did not surface until the mechanism under examination was tested in this distributed environment.

In either the development or real environments, 'dummy' job submission code which simulates randomly successful or failed grid job submissions is used in place of the Service Providers which would actually interact with real-world 'grid job submission' infrastructure (e.g. by executing the gLite commands described in section 2.2.1, page 12). Due to the current unavailability of grid services in my development environment, dummy execution Service Providers will be used in place of 'real' grid job submission. It is worth noting that examination of successful job submission is not the subject of this experiment. This was examined in the work of Pierantoni [107]. Furthermore, none of the job submission, pricing policies, etc. set out in that work have been altered.

WAR File Packaging

The code-base on which all the 'basic' agents for this experiment are based was stripped of all the classes which were to be added. This means that none of the classes which were added as abilities were already available in the package of the application (i.e. the WAR file in this case). A sample from the list of the 138 class files and the 37 JAR files which are necessary for the 'basic' agent are listed in appendix D, page 202. Although the full list is not shown in this appendix, it is important to state that there was no overlap with this inventory of classes created for the agent configuration (listed in appendix C, page 201).

Class Loading and Inheritance

All of the classes which were added as abilities were subclasses of a `ServiceProvider` super class. This super class definition is already included in the WAR file package of the SGA and also acts as a super class to the minimal set of abilities which are necessary for the SGA to run initially. Without the presence of this super class in the WAR, loading any class which extends it would not be possible.

Scope of Transient Fields in Serialized Classes

Transient fields or variables are a common feature of classes which are intended for serialisation. A transient field is one which remains null while the class is serialised; it will subsequently be null when the class is deserialised. In order to be able to use these deserialised classes agent-side, it was necessary to change the scope of such fields from being *class scope* to *method scope* (i.e. move each field within the methods in which they were required). Classes with transient fields which are *class scope* can be successfully serialised, transported and loaded to the SGA, but will throw a null pointer exception at runtime when that ability is used by the agent, specifically when the transient field is referenced.

6.3.3 Running the Experiment

Eight separate VMs were created using OpenNebula cloud infrastructure [30], their IP addresses noted and inserted into the program used to create the ACFs. When created, these files were moved to a location from which they could be accessed over HTTP by the VMs.

The MAS Creation tool described in section 4.3 on page 59 was used to create and deploy eight new *basic* agents, one in each of the cloud VMs. In turn, each agent was sent a message which added an initial ability, providing each agent with the capacity to then fetch and process its designated ACF from the HTTP server.

6.3.4 Evaluation of Outcome

As outlined in section 1.4.1 and at the start of this section (6.3), the criteria for success or failure is based on whether the proposed mechanism is flexible and extensible enough to allow the sophistication of the original hardcoded SGAs to be reproduced, but without the configuration being hard-coded in the agent from the outset.

This is evaluated in a number of ways, each of which is listed and answered in succession here. Comparisons are made between the system of this experiment and the remaining available code from the original SGA work. While the agents for this experiment run in web-service containers in separate VMs, Pierantoni's code of 2008 runs in the development environment. However, the behaviour and functionality is still suitable for comparison.

The two questions outlined at the start of this experiment description (on page 118) are repeated here and are answered in turn. The overall experiment result is then summarised.

Evaluation Question 1: Is the configuration of the new SGAs identical to those of the original SGAs?

Evaluation Method 1: After an SGA is configured, the listing of its internal data-structure is printed to screen. In this experiment, this output was captured in the log files of the web-service containers. In the case of the original 2008 agents, this output can be seen in the console-output of the IDE in which they were run. The logged output from an SGA, from its initialisation to being fully configured is quite verbose, totaling approximately 370 lines. Part of this output is the Classad expressions which map the action-names used for each ability to the binary stored in the agent's data-structure. Examination of these parts of this output shows how an ability has been added in a given agent.

Answer 1: Comparisons between these two outputs show that the abilities configured in the agents in this experiment are identical to their equivalent abilities in the 2008 work. Appendix E on page 203 shows a short sample of this logged output. It lists the configuration details of the 'Wallet' ability is S-1 and the same data for the equivalent agent in the 2008 work. This listing shows that the 'wallet' ability has been added in an identical way to both the agents being compared. It can be seen to have been added three times, each time with a different action-name, but each one referring to the same instance, shown here by its memory location.

A further comparison is demonstrated in appendix F, for an ability added to agent P-4 in this experiment. This ability, as well as having an action-name associated with it, also required three other strings to be transported with for configuration. It can be seen that the configuration details available in both SGAs are identical, showing that the configuration was correct. Note that this is the same ability used to illustrate the mapping of an abilities blueprint file line to an ACF in figure 4.9 on page 79. By comparing the content of the abilities blueprint file line and the ACF in figure 4.9 to the output in appendix F, it can be seen how the information needed to configure this ability maps from the abilities blueprint file line to the said ACF. From there it is passed to the appropriate method in the SGA during configuration, and eventually to the data-structure of the SGA. This example was chosen intentionally, aiming to demonstrate the 'lineage' from the *abilities blueprint* file to the ACF, and eventually on to the actual configuration of the SGA within the system.

Evaluation Question 2: Is the same functionality and behaviour observed in the system created in this experiment as that observed in the original hardcoded?

Evaluation Method 2: To test this, as was done in the 2008 work, a series of messages (10 in total) were sent to S-1. Each message triggered a potential transaction between the various agents in the system and the behaviour of the system was observable in the logged outputs of the agents.

Answer 2: Repeated trials showed that the system had the same behaviour as that of the original system of 2008. The outcome of each message showed either the success or failure (as randomly determined in the simulation code configured in the SGAs) and the changes in the 'credit' values assigned to the wallet after each message was processed. This is identical to the behaviour observed when running the original SGAs from 2008. An abridged sample of the output from these trials are included in appendix G on page 205.

Furthermore, no errors, agent communications failures or Java exceptions occurred while processing the ten messages sent.

6.3.5 Summary of Experiment Result

Positive answers to these two questions demonstrate that SGAs can be created and deployed using the approach described in this thesis and that doing this provides the same functionality and behaviour as was provided using a hard-coded approach in the original work [107]. This significantly improves the configurability and creation process of SGAs and is a contribution of this thesis.

6.4 Experiment 2: Indexing, Retrieval and Similarity

This section describes experiments which use Lucene search libraries for indexing and retrieval of textual descriptions. Search technology is used because it provides similarity measurements between these descriptions. The descriptions used in this experiment apply to a number of different entities, each of which fall into a small number of different categories. If reliable similarity measurements can be obtained, and if these measurements can be used to group entities, or distinguish them from each other, this information may be useful as part of a resource allocation decision making process.

Showing that these entities can be grouped or distinguished from each other supports the rationale for incorporating search technology with SGA for use in resource allocation, one of the contributions of this thesis, as outlined at the start of this chapter.

6.4.1 Experiment Description and Layout

Sixteen different textual descriptions, each belonging to one of four different categories are created and indexed in a Lucene index. This index is then used for a number of different types of searches. For background information, sub-section 6.4.2 describes how and why these entities are selected, and how their descriptions are created and subsequently added to the index as documents.

Following the creation of the index, the remainder of this section examines the use of *two* different types of queries for searching that index. The use of the two query types is described in two different sub-sections and are also named part (A) and part (B).

Part (A) is described in sub-section 6.4.3. Two sets of queries are used to search the index. One set consists of 4 queries, each one containing a group of 12 'designed' terms. Each designed query is used to target documents from a given category. A further set of queries uses the same terms but instead, they are randomly selected from the pool of 48 terms used to create the 4 designed queries. By doing this, a large number of ‘designless’ queries are created. Statistical analysis is used to compare the outcomes from using the ‘designed’ and ‘designless’ queries, and examine if using ‘designed queries’ is an effective way to return or distinguish the documents that are from the different categories.

Part (B) is described in sub-section 6.4.4. A type of query known as a ‘More Like This’ (or MLT) is used here. This type of query is based on the contents of a chosen document from the index, with the aim of finding other documents similar to that one (discussed briefly in section 5.4.4, page 96). This part of the experiment shows the usefulness of this type of query, and how they are comparable, to some degree, to the outcomes of part (A) of the experiment.

Criteria for Evaluation of Experiment

As will be seen in part (A) in particular, the queries used are ‘designed’ to return documents from a specific category. Two questions arise which define the criteria for success or failure of this experiment:

1. “Can designed queries be used to return documents from a specified category?” Or alternatively, “Can designed queries be used to distinguish documents that are from one category or another?”

2. “To what degree of reliability or confidence can this be done?”

In sub-section 5.4.4, we saw that similarity in search results is expressed as a *score* for each of the documents returned. Therefore, for the query associated with each category, the scores for the documents of that category should be higher than those of the other categories. If found to be the case, this will indicate that the ‘designed queries’ were successful in retrieving their respective documents.

For part (B), the question arises:

“Can a query based on a document from a particular category be used to retrieve other documents from that category?”

If the outcomes of the MLT queries are similar to those of the designed queries, this will show the two approaches are comparable and can perhaps be used interchangeably.

6.4.2 Index Preparation

This section sets out the steps carried out to create the necessary index used in both part (A) and (B) of this experiment. It describes the methodology for selecting the entities and how text files were compiled for indexing.

Category Selection

There are many different ways to categorise entities and the method chosen here is kept as simple as possible. Four categories of entities were arbitrarily selected, each of which are different from each other because they contain entities which have different aims. The entities for each category were chosen because they are *similar to each other in what they aim to do* and also because they are typical of entities which may consume, produce or broker either work or resources in the environment envisaged for SGAs in section 3.4, page 53.

There are four (arbitrarily chosen) entities in each category, each one having similarity with their cohorts (i.e. each category can be said to have internal similarity). Table 6.4 lists the four categories along with the four entities then selected for each one.

Text Collection and Creation of Text Files

For this experiment, the vision of how social profiles are to be created, as outlined in sub-section 5.5.2, page 102 will not be adhered to. Compiling each text file manually would be tedious and time consuming, so for practical reasons[1], the approach outlined below will be used instead.

[1] The main practical reason is the lack of available industrial scale data for actual social profile creation.

No.	Category	Entity Name
1	Commercial Companies and Corporations	Vodafone Ryanair General Electric, USA Virgin Atlantic
2	Universities, Academic Institutions	Trinity College Dublin, Ireland Imperial College London, UK University of California Berkeley, USA Princeton University, USA
3	Charities and Benevolent Foundations	Bill And Melinda Gates Foundation Doctors Without Borders The Carnegie Foundation The Global Fund
4	Government or Similar Depts and Institutions	US Govt, Department of Commerce Irish Govt, Dept. of Agriculture, Food and Marine Irish Govt, Dept. of Education Skills European Commissioner for the Environment

Table 6.4: Categories of Entities Used

For each of these 16 entities listed in the center column of table 6.4, a text file containing information relating to each one was compiled. A mixture of automated text harvesting techniques and manual editing and cleaning were employed for this. The sequence for creating the 16 text files can be summarised as follows:

1. An initial list of URLs was created, containing one URL for the website of each of the 16 entities.

2. Using this initial list and a custom made web-crawling program, the internal URLs or links on each of the 16 websites were harvested, creating 16 new URL lists, one for each entity.

3. In turn, each of these lists was used to get the web content available at the

internal URLs of each entity's website. The text from each webpage retrieved was then extracted using the Tika extraction toolkit[2] [42] and inserted into a text file for that entity. These files formed the basis for the text descriptions which were stored in the Lucene index later.

Manual editing and cleaning; Not all harvested URLs yielded useful descriptive text. For example, some did not have many descriptive text passages and others had content such as large tables of figures. It was also noticed that web-pages like 'privacy' and 'security' policies did not contain text which was specific to the entity. To adjust for this, the harvested lists of URLs was manually 'curated' to remove such URLs.

In addition to entities' own websites, and where available, URLs of descriptions on the free collaborative online encyclopedia *www.wikipedia.org* were also added. Text copied from publicly available articles in national and international online news media web sites was also used.

The only commonality between all the sources used was that the content used was related to or pertained to the relevant entities. These files were compiled in this way, using the assumption that text which relates to an entity either directly or indirectly describes that entity. Collecting random text was not used as such text would not be likely to distinguish one entity from another. While this process is somewhat crude, it aims only to create text files which contains words and sentences which relate to the respective entities.

Indexing of Text Files

The resulting files were converted into Lucene documents and indexed. Section 5.4 (on page 93) discusses the structure of the Lucene document and the Lucene API used to create them. The sequence for indexing these files was as follows:

1. An instance of an index was created.

[2] The Tika toolkit from Apache™ takes, as an input, a number of different document formats (HTML, XML, PDF, etc.). It removes all metadata, markup or other binary structure and outputs a stream of plain text.

2. In turn, each text file was read into memory.

3. For each file, an instance of an empty Lucene document was instantiated.

4. The entity name and its category were added as fields called *name* and *category*. The `NOT_ANALYSED` option was used for both.

5. The entire content of the text file was then added as a field called *text*. For this, the `ANALYSED,TermVector.WITH_POSITIONS_OFFSETS` and `Field.Store.NO` options were used. No logical data (as described in section 5.5.2) such as location, etc. was indexed.

6. A populated Lucene document for each entity was, in turn, added to the index. Each document is, at this point a *vector space model* entry.

The index created and populated was stored on persistent storage and contained only 16 documents. The storage space used by the populated index was roughly equal to that used by the 16 text files, approximately 5.5MB. The populated index was now ready for searching and retrieval.

6.4.3 Part (A) — Using Designed and Designless Queries

Searches are carried out using one or more words (or terms) grouped together to become a 'query'. Recalling section 5.4.4, just as a document becomes a vector in the index, a query too is converted to become a vector. The similarity score is a measurement of how alike a query and a document are. This, part (A) of this experiment uses two different sets of queries for searching and retrieval of the documents from the Lucene index.

Selection of Query Terms

Query terms are selected based on what the entities of each category 'aim' to do. This is in keeping with how the categories (of table 6.4) were originally selected. For the purpose of illustration, the following is a short textual description of the aims of each category:

> ‘‘*Companies and corporations aim to provide services and make a profit. Universities aim to educate people and do research, they have lecturers and students. Charities and benevolent foundations aim to gather and distribute resources to address poverty, humanitarian problems or other ills in society. On the other hand, a government body administers public affairs and enforces legislation and policy. They are usually organised into departments and have ministers or a secretary.*”

Table 6.5 lists the four different sets of terms, 12 for each category, used to search the index for documents. It can be seen that there is some overlap between the query terms shown in table 6.5 and those that appear in the above brief category descriptions. The creation of the above textual description and the selection of query terms was no more sophisticated that this arbitrary process of picking words which can be used to describe what the entities of each category do on a daily basis.

For this experiment, we will call the queries made from these four sets of terms the ‘designed queries’. Later, we will see that this will help to distinguish them from the other set of queries called the ‘designless queries’.

Running Designed Queries and Collecting results

Four individual searches on the index were run using the ‘designed queries’ from table 6.5. In each search, all 16 documents are retrieved. Table 6.6 shows, as an example, the results obtained from using Query 1 from table 6.5. It can be seen that there is at least *some* similarity between this query and every document (as is the case for all queries in this experiment). For a given individual search result, such as the one shown in table 6.6, the similarity score for each document is recorded. The scores for the four documents in each category are summed. For one query, this yields four totals, one for each category. Each total consists of four individual document similarity scores. This is repeated for all four queries.

No.	Category	Query Terms Used (12 for each)
1	Commercial Companies and Corporations	corporate profit expansion sales assets investment produce output revenue grow service global
2	Universities, Academic Institutions	degree masters phd research science arts graduate campus student lecture scholar education
3	Charities and Benevolent Foundations	poverty humanitarian voluntary training malaria aids development vaccination aid donate philanthropic medicine
4	Government or Similar Departments and Institutions	administration legal secretary ministers policy taxation political legislation trade parliament federal regulation

Table 6.5: Designed sets of query terms used for retrieving text descriptions.

Running Designless Queries

A series of 'designless queries' were run on the same index. Twelve terms were used in each query, the same as the number used in the 'designed queries'. The terms for the 'designless queries' were selected randomly[3] from the pool of all of the 48 terms listed in table 6.5, i.e. the combined set of the 48 terms used for the four 'designed queries'.

When generating each query, no single term was allowed to be used twice in the same query. 1 million queries were run from this pool, and for each query, the sums of scores for documents from each category was recorded. Incidentally, for groups of twelve terms, there are in excess of 69 billion different combinations possible from this pool.

Results

The totals recorded from the 'designed queries' are shown in table 6.7 with the highest *totals* in each column in bold type. It can be seen that the highest value for each

[3] Java util.Random was used for random number generation.

Query 1, Commercial Companies and Corporations Terms		
Rank	Similarity Score	Entity Name (Category)
1	**0.075778**	Vodafone (1)
2	**0.074451**	General Electric, USA (1)
3	**0.052064**	Ryanair (1)
4	0.029512	Doctors Without Borders (3)
5	0.026211	US Govt Dept of Commerce (4)
6	0.020018	The Global Fund (3)
7	**0.019127**	Virgin Atlantic (1)
8	0.014487	Bill & Melinda Gates Foundation (3)
9	0.013734	Irish Govt Dept of Agric Food Marine (4)
10	0.008825	The Carnegie Foundation (3)
11	0.007491	Irish Govt Dept of Education & Skills (4)
12	0.007334	University of California, Berkeley, USA (2)
13	0.006828	Imperial College London UK (2)
14	0.004292	European Commissioner for Environment (4)
15	0.003873	Trinity College Dublin (2)
16	0.002944	PrincetonUniversityUSA (2)

Table 6.6: Extract of Results for Term Query Search

query is the total for the documents of its corresponding category.

Category	Qry 1	Qry 2	Qry 3	Qry 4
1	**0.22142**	0.006869	0.010501	0.100621
2	0.020978	**0.250755**	0.048549	0.021691
3	0.072842	0.140663	**0.207188**	0.065589
4	0.051729	0.034665	0.023554	**0.216086**

Table 6.7: Sum of similarity scores for each category against its corresponding 'designed query'.

For the series of 1 million 'designless queries' run, the mean, standard deviation, maximum and minimum values for the totals for each category are detailed in table 6.8.

Category	Mean	StDev	Max	Min
1	0.10012	0.04204	0.37009	0.00551
2	0.12402	0.06165	0.66166	0.00376
3	0.18075	0.05228	0.45794	0.03117
4	0.09724	0.04131	0.40996	0.00518

Table 6.8: Statistical data for 1 million 'designless queries'.

Comparison and Analysis Results

This analysis calculates the probability that a designless query would yield a similarity score equal to or higher than the observed value returned for each of the 'designed queries' seen in table 6.7. In other words: "Are the results from designed queries significantly different from what was obtained using the randomly generated designless queries?"

As both sets of data are based on the same index and the same set of terms, direct comparison between the two results is possible. For the remainder of this analysis, the bold type values from table 6.7 for each category will be called the 'observed' values.

Examination of designless query results Further analysis of the 1 million designless query data using statistical software[4] shows that data for all 4 categories did not (visually) appear to be normally distributed. As an example, the distribution and normality test plots for category 1 are shown in figures 11.1 and 11.2 in appendix K. All four of the distribution plots are skewed to varying degrees towards the smaller values, indicating that the data is not normally distributed. Further to this, a normality test on this data using the Anderson-Darling test yields a P-value

[4] Minitab Version 16 Statistical Software is used in this thesis.

<0.005. This shows that the data collected is not normally distributed and therefore not suitable for analysis using Gaussian normal distribution methods. This low P-value may be explained by the visible skew in the distribution plot, but also possibly by the fact that each data point is the sum of four actual data similarity scores. No further analysis was carried out to determine the reason for the skew distributions or low P-values.

Calculating Probability

Because of the ease with which large sample sizes can be generated for this test[5], determining the probability that a 'designless query' score will be higher than that of the observed values is done by directly determining the proportion of the population which falls *above* the observed values as it lies within the data from the 'designless query' test. This was done for each category using the following methodology:

1. The observed value is appended to the designless query data set for its corresponding category.

2. This column is then ranked and the position of the observed value noted.

3. The percentage of entries above the position of the observed value in the ranked data is taken as the probability that any designless query value would exceed that observed value.

The probability $\hat{p}$ for each category is given in table 6.9 with probabilities given in percentages (%). The right-most column of table 6.9 is the 95% Confidence Interval (CI) for this probability, expressed as $\hat{p} \pm \delta$, where δ is calculated using equation 6.1 and N = 1 million.

$$\delta = 1.96\sqrt{\frac{p(1-p)}{N}} \tag{6.1}$$

An Alternative View of Search Results

An alternative view of the 'designed queries' results is given in table 6.10. This does not aim to further analyse these results other than to shed further light on

[5] 1 million queries take about 12 minutes on the development computer detailed in table 6.2.

Category	**Obs. Value**	**Position**	**Probability $\hat{p}$ (%)**	**95% CI(%)**
1	0.22142	8676	0.868	0.868 ± 0.019
2	0.020978	39434	3.943	3.943 ± 0.039
3	0.072842	288969	28.897	28.897 ± 0.091
4	0.051729	10594	1.059	1.059 ± 0.020

Table 6.9: Probabilities of 'designless query' values exceeding observed values.

the usefulness of using 'designed queries'. Recall that the object of this part of the experiment is to record the similarity scores for documents from a given category when that category is targeted with a 'designed query'. Table 6.7 indicate that these scores are higher than when 'designless queries' are used. However, another way to look at these outcomes is to report which documents were actually returned in each search. Table 6.10 summarises the outcomes and comments briefly on exceptions found in each one. Although all 16 results were returned in each search, for brevity, only the top 4 will be discussed here.

Query No.	**Category**	**Outcome**	**Comments on Exceptions**
1	Companies, corporations	3 of 4 returned in top 4.	Doctors Without Borders in category 3 ranked at 4.
2	Universities and Academic Inst.	3 of 4 returned in top 4.	The Carnegie Foundation, an educational charity foundation, ranked at 1.
3	Charities and benevolent foundations	3 of 4 returned in top 4.	Imperial College London UK ranked at 4.
4	Government like Institutions	2 of 4 returned in top 4.	Vodafone and Ryanair at position 3 and 4.

Table 6.10: Summary of top 4 rankings from 'designed query' terms.

Outcome of Part (A)

It is time to return to the original questions to be answered from Part (A) of this experiment:

Evaluation Question 1: Can designed queries be used to return documents from a specified category? Or alternatively, can designed queries be used to distinguish documents that are from one category or another?

Answer 1: Addressing Question 1, table 6.10 shows that for 3 of 4 of the categories, 3 of 4 of the documents were returned in the top 4 results. In the remaining category, 2 of 4 documents from that category were returned. This outcome supports a 'yes' answer to this question . However, as clear and as positive as it may be is, nonetheless, based on a very small sample of 16 documents and four queries.

Question 2: To what degree of reliability or confidence can this be done?

Answer 2: Addressing question 2, a more robust demonstration of the capability of a 'designed query' to return documents from a specified category is given in table 6.9. For three out of the four queries, the probability that a 'designless query' will return a higher similarity score than a 'designed query' is between 1% and 4%. This is a low probability. Therefore, there is a statistically significant difference between the outcomes from the two different types of query. For category 3, the probability is much higher at almost 29%.

These results are for similarity scores. These scores are used directly to return and rank documents. Therefore, the 'designed queries' as applied to the index, are an effective approach to distinguishing between entities which are described using textual descriptions.

In summary, and notwithstanding the category 3 result, these results show that entities from a targeted category can be retrieved from an index and thereby can be distinguished from other entities using 'designed queries' in search technology.

Discussion

Bias in the query terms used Examining table 6.8, the mean of the 'designless query' values recorded for category 3 is between 1.3 to 1.5 times that of the other categories. Although the maximum values for all categories are largely similar (within 0.029 or each other), the minimum value of category 3 is between 6 and 8 times larger than the other categories' minimums. These two phenomena suggest that there is an inherent bias towards the documents of category 3 in the overall pool of terms listed in table 6.5.

The category 3 result The category 3 result could be interpreted as suggesting that this category may be more difficult to distinguish from others but this is unlikely. Given that the number of documents in each category is only four, this less definite result may also be attributed to the crude method used for creation of the text file descriptions. Further examination of the category 3 result was not carried out.

Diversity and overlap between categories Entities in all categories have a lot of diversity in their aims, and each one shares some of the concerns and aims of entities in the other categories. Therefore, some terms which may be used to describe one category may also apply to others. As an example, in the search results using the category 2 query of table 6.5, The Carnegie Foundation (it itself an educational research foundation) scored higher than any of the universities or academic institutions from category 3.

Expectations from similarity scores The outcome of these four searches is an 'expected' result. But this expectation in itself, merits some discussion.

It was expected that, to at least some degree, a given query should score higher against at least *some* of the documents from its corresponding category. As can be seen in table 6.10, this is in fact how each query result returned the documents. However, there was no expectation of how large the sums of the scores would be. For example, it is theoretically possible that, by using a query which had a high similarity to four different documents, the sum for one category could be close 4.0 (recall that similarity scores are between 0 and 1). Furthermore, there was also no expectation of how different any of the categories total scores would be, i.e. whether they would be close to each other or whether there would be one total for one category which was clearly much larger than the others.

Query and description construction Description content and query selection are important. If search technology is to be useful for either grouping or distinguishing entities, care is needed in how descriptions (and by extension, social profiles) are composed, perhaps to the extent that some terms should be intentionally left out for improvement of search scores. In the case of an individual creating its social profile, rather than simply 'harvesting' text of related content, a more selective and refined approach should be used.

Recall that the 'designed queries' used in this experiment were constructed based on only fundamental knowledge of each of the categories. In general use of search technology, e.g. when searching on the world wide web, results depend upon the effectiveness of the queries used. For distinguishing entities from one another, results also depend on the effectiveness of the descriptions used. Refining and improving both descriptions and queries to make them more accurate and effective is not within the scope of this thesis. Work such as that lies in the domain of 'Information Retrieval', where several advanced techniques can be employed to improve search results. The Lucene technology used here facilitates a great number of these techniques. Therefore, there is ample opportunity for these techniques to be incorporated when search is used in SGAs.

6.4.4 Part (B) — More-like-this Queries

In part (A) of this experiment, the queries used are made from arbitrarily chosen terms. Lucene provides another approach for creating terms called *More-Like-This* or MLT. An MLT query uses an existing document in the index as the basis for a query for searching for other documents similar to it.

Criteria to be Examined

This, part (B) of the experiment aims to determine if using an MLT query is comparable to using a 'designed query' when targeting documents from a particular category of document in an index. The specific question to be addressed by this part of the experiment is:

"Can a query based on a document from a particular category be used to retrieve other documents from that category?"

It will not be feasible to use statistical analysis to compare the outcomes of the MLT queries to those of either of the 'designed' or 'designless' queries of part (A) as there is no readily available equivalent to a 'designless' MLT query. However, some comparison can be made to the results of part (A) as presented in table 6.10.

Experiment Procedure

This experiment uses the same index used in part (A), the creation of which is described in sub-section 6.4.2. The index is queried 16 times, using an MLT query based on each of the individual documents contained therein. The 16 separate results are summarised here and the capability of each MLT query to retrieve the remaining three documents from its own category is examined.

To create an MLT query, the subject document is first retrieved from the index. A number of terms from this document are used to construct a query. The terms selected are influenced by a number of user defined parameters. Apart from the parameters detailed in table 6.11, Lucene default values are used. These parameters govern (among other things) how many terms are used from the original document. As an example, the queries used in part (A) (sub-section 6.4.3) of this experiment had twelve terms. For manual query creation, large numbers of terms can become unwieldy. As can be seen in table 6.11 for the '*set*MaxQueryTerms' value, the number of terms that can be used is much larger; 100 is used in this case. Term selection is also governed by how the frequencies of terms in the subject document compare to

Parameter	Value Used	Details
*set*MinDocFreq	1	For a given term, the Document Frequency is the count of documents which contain that term. Terms will be ignored if they do not occur in a number of documents at least at or above the 'MinDocFreq' set.
*set*minTermFreq	2	Sets the frequency of occurrence below which terms will be ignored in the MLT source document.
*set*MaxQueryTerms	100	Sets the maximum number of query terms that will be included in any generated query.

Table 6.11: Lucene More-Like-This Query Parameters

frequencies of those terms in other documents in the index (recall section 5.3.1, page 91).

6.4.5 Analysis

Table 6.12 summarises the results for each document, expressing the outcome as a count of the number of documents from the query documents' category that were returned in the top 4.

The result ranked at number 1 for any MLT query is the document on which the query is based (as it is intuitively most similar to the query). Therefore, each result in table 6.12 includes the documents on which the query is based. Where only one document is counted, (as is the case for the query based on the 'US Dept. of Commerce') this was the only one returned.

6.4.6 Outcome and Evaluation

Returning to the question to be answered by this, part (B) of the experiment:

Cat No.	Subject Document used for MLT Query	Matches in top 4
1	Vodafone	4 of 4
	Ryanair	4 of 4
	General Electric	4 of 4
	Virgin Atlantic	2 of 4
2	Trinity College Dublin	3 of 4
	Imperial College London UK	3 of 4
	University of California Berkeley, USA	3 of 4
	Princeton University United States	4 of 4
3	Bill And Melinda Gates Found	3 of 4
	DoctorsWithoutBorders	3 of 4
	The Carnegie Foundation	2 of 4
	The Global Fund	3 of 4
4	US Govt Dept of Commerce	1 of 4
	Irish Govt Dept of Agriculture Food Marine	2 of 4
	Irish Govt Dept of Education Skills	2 of 4
	European Commissioner for the Environment	3 of 4
All	Mean across all categories	2.85 of 4

Table 6.12: Summary of MLT Query Results

Evaluation Question: Can a query based on a document from a particular category be used to retrieve other documents from that category?

Answer: In table 6.10, for the 'designed queries', no more that 3 of 4 document for any category were retrieved by any designed query. Table 6.12, results for the MLT queries, shows that in some instances, 3 out of the 3 remaining possible documents were returned (not counting the document returned because it is the one upon which the MLT query is based). This indicates that MLT queries provide results which are similar to those of the designed queries.

6.4.7 Discussion

In a large multiple-agent system of SGA, where social profiles are ubiquitous, distributed across every agent, a *more-like-this* query may be a suitable and interchangeable alternative to using a 'designed query'. A practical example of an MLT query being used between two different SGAs is demonstrated later in Experiment 3 in sub-section 6.5.5.

The results set out in tables 6.10 and 6.12 are based on small numbers of tests. Further tests, using larger numbers of indexed descriptions, categories and queries may give a more definitive and reliable result. However, this is not addressed in this thesis as this experiment only sets out to support the rationale for using search technology in SGAs.

6.5 Experiment 3: Integrating Search into SGAs

In section 6.4 of this chapter, creating and searching indexes were examined. Earlier, in section 5.5.3 (page 104), the text content of a social profile was transported to an SGA in the XML file format and subsequently indexed in that SGA.

This experiment examines integrating further Lucene capabilities into Social Grid Agents. The motivation for doing this is to examine if (and how) search functionality can be integrated and used in the distributed environment of a multiple-agent system. A successful integration of search technology into SGAs is necessary if social profiles, in the form of textual descriptions, are to be used in resource allocation decisions. Section 5.4.5 introduces cosine-similarity and describes how this measurement is calculated. For comparisons of social profiles within agents, this functionality also needs to be integrated to SGAs.

6.5.1 Criteria for Evaluation

Experiment 2 (section 6.4) used Lucene libraries to create an index and subsequently search that index using two different types of queries. In that experiment, the indexing of the documents and generation of queries takes place within the confines of a single system. The principal question to be addressed by this experiment is:

"Can the search functionality provided by Lucene in a 'single' system be migrated to a system of distributed agents ?"

The answer to these questions will be determined by carrying out functionality tests for the following actions:

1. Sending and receiving Lucene document objects from one agent to another.

2. Sending and receiving Lucene query objects from one agent to another.

3. Using a cosine-similarity calculation to compare social profiles within SGAs.

These three pieces of functionality will be tested and demonstrated in turn in this experiment. Each one will be addressed in the outcomes of each part of the overall experiment.

Being able to send and receive Lucene documents means that social profiles can be sent and received between agents. Sending of Lucene query objects between agents will allow one agent to search the index of another agent. Such a request can be answered by the return of one or more social profiles, sent back to the requesting agent.

6.5.2 Experiment Description and Layout

This experiment examines and evaluates the techniques required for the distribution of social profiles and query objects throughout a multiple-agent system. The challenge in doing this is essentially one of moving Java objects. Recall that in the initial configuration of an SGA, the initial binary ability is sent to the agent as a serialised object (i.e. a Base64 string) which was inserted in to a Classad message for sending in a SOAP message over the network. This technique is adapted and further exploited in this experiment.

When indexing a social profile, the descriptive text is transported to the agent in an XML file. Once it is added to an agent's index, it becomes a Lucene document and the XML file is no longer stored or used. Therefore, from the point of indexing onwards, any processes involving a social profile are carried out on a Lucene document object.

In keeping with SGA architecture, one ability is added for each specific process. Depending on the complexity of the overall task which needs to be carried out, multiple binary abilities are used. As in other experiments in this thesis which use SGAs, all the functionality tests are carried out in 'real' web-service containers, hosted in virtual machines. All SGAs used are created and configured using the MAS creation and agent configuration mechanisms used and examined in section 6.3.

The layout of the remainder of this section is as follows. Initially, some technical requirements and background supporting work is described in sub-section 6.5.3. In sub-section 6.5.4, sending and receiving of social profiles is examined. Sub-section 6.5.5 examines sending a query object to another SGA. Finally, the integration of a cosine-similarity calculation into an SGA is examined in sub-section 6.5.6. The outcome and evaluation of these experiments is then discussed.

6.5.3 Technical Requirements — Libraries

Adding abilities to SGAs, as described in chapter 4, requires that each ability be contained in a single class. Any other classes required by that ability must be either already available in the application package, or alternatively must be made available to the application by another means. In this experiment, in order to incorporate search and the cosine similarity calculation in to an SGA, two sets of additional (external) libraries (3 JARs in total) are needed. These libraries have not been used in SGAs prior to this. For expediency, these JARs are added to the application package (the WAR file of the SGA web-service application in this case). An alternative approach to providing such 'external classes' to an SGA will be examined in section 6.7 of this chapter).

Storage Location for Lucene Index

Lucene allows an index to be stored in system memory. However, for long-term continuous operation, each agent needs access to persistent storage on its host machine. The persistent storage of the SGA host machine is accessed when social profiles are written and read from the index. In the environment of the Apache Tomcat container used in this thesis, the SGA application is owned and run by a user 'tomcat'. To allow file system access on the host container, a user account is created for the 'tomcat' user and its home directory is used as a storage location for the index. Below this location, further directories are created and the following naming scheme is used: `/home/tomcat/.<agent-name>/.indexFiles/<agent-name>/`. Using the name of the agent ensures that, where there are multiple SGAs hosted in the one VM, their respective indexes are stored in different locations.

Simulating a Populated Index

In a real-world application, it is envisaged that an agent's index would contain its own social profile and that of several others. These would typically have been accumulated during interactions with other agents over time or by other distribution processes. To simulate this, one SGA in this experiment has approximately 20 social profiles in its index.

For this simulation, 20 XML social profile files are created. A similar approach is used for harvesting content for these as is used for the text file descriptions used in the experiment in section 6.4. For this experiment, each file has the structure discussed in section 5.5.2 (page 102, each one containing descriptive text under two headings, the entity's own description and a third-party description. Some logical information (name, location, address, etc.) is also included. The contents of these files describe a number of entities which are commonly involved in computer resource and work allocation decisions. Examples are computing resource providers like commercial Cloud providers and Grid computing providers, and also bodies which often provide the work which uses these resources.

Figure 6.13 illustrates the layout of the agents used in the experiments in this section. In this simple system, there are two SGAs, each having its own social profile in its index. This was added when the agents were initially created and configured. Agent S-2 has a further 18 social profiles describing the other 'fictional' agents.

6.5.4 Requesting and Returning a Social Profile

This experiment examines the transfer of a social profile between two SGAs in a scenario where one agent requests, and the other agent returns a single social profile. This takes place where an SGA needs the social profile of another in its own index. It may do this as a single task (for example, at the request of its 'owner') or as part of an overall bigger task, consisting of a number of sub-task.

The ability used in this task has an action-name `RequestIndexSocialProfile`. This ability processes a message which contains the address of the target SGA; in this application this is a web-service URI. From within the ability, a new message

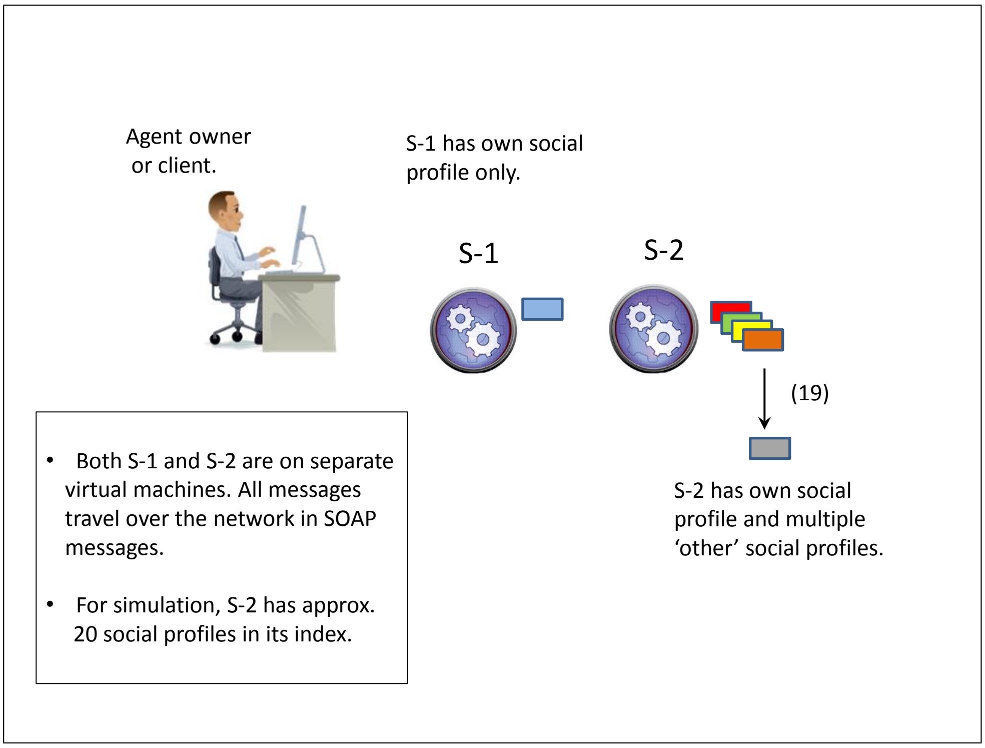

Agent owner
or client.
S-1 has own social profile only.
S-1
S-2
(19)
S-2 has own social profile and multiple 'other' social profiles.
Both S-1 and S-2 are on separate virtual machines. All messages travel over the network in SOAP messages.
For simulation, S-2 has approx. 20 social profiles in its index.

Figure 6.13: General layout for SGAs used in Lucene integration experiments.

is generated which is sent to the remote agent. This sent message has an action-name `ReturnSocialProfile` and does not need to have any content. The expected response to this message is a serialised Lucene document.

Within the agent receiving this request, the sequence of events is programmed in the ability which responds to the message. Each agent 'knows' its own name (this is hardcoded when it is first created). The ability creates the necessary Lucene objects to query its own index and searches for the document which matches its own name. On retrieving its own social profile from the index, the Lucene document is serialised (i.e. converted to a Base64 string) and inserted (as content) for returning to the requesting SGA.

On receipt of this returned message, the 'requesting' agent deserialises the document and adds it to its own index. The logic used here checks if there is already a document in the index for the remote SGA. If one is found, it is deleted and the 'latest' version received from the remote agent is added instead. This sequence of events is illustrated in figure 6.14.

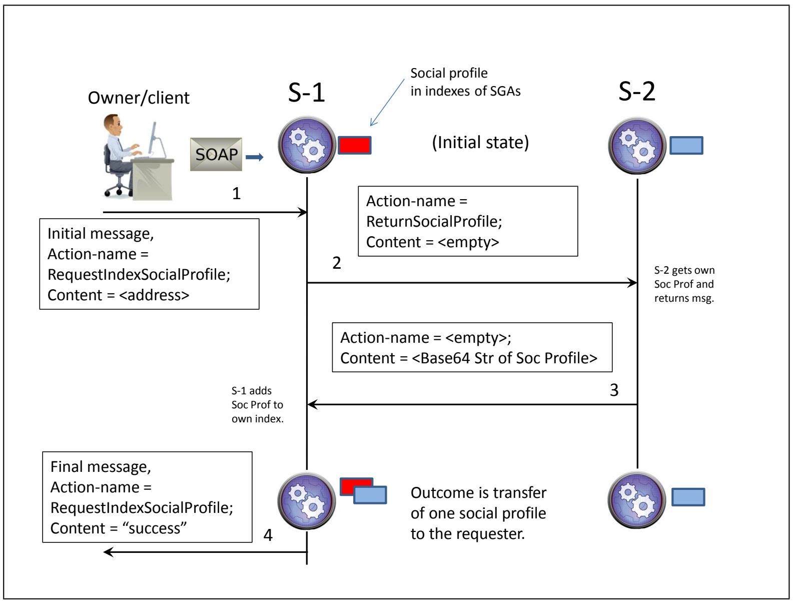

Owner/client
S-1
Social profile in indexes of SGAs
S-2
SOAP
(Initial state)
1
Action-name = ReturnSocialProfile; Content = <empty>
Initial message, Action-name = RequestIndexSocialProfile; Content = <address>
2
S-2 gets own Soc Prof and returns msg.
Action-name = <empty>; Content = <Base64 Str of Soc Profile>
S-1 adds Soc Prof to own index.
3
Final message, Action-name = RequestIndexSocialProfile; Content = "success"
4
Outcome is transfer of one social profile to the requester.

Figure 6.14: Sequence of messages for simple request and return of social profiles between SGAs.

Outcome

Returning to the first of the functionality tests set out at the start of this experiment, this was described as follows:

"Sending and receiving Lucene document objects from one agent to another"

The outcome of this part of the experiment shows that this functionality has been achieved. This interaction between the two SGAs was successful. The number of documents in the requesting agent increased by one. Although this sequence was executed without error, an invalid URI being sent, or the unavailability of a remote agent due to unforeseen network issues present opportunities for possible errors.

Sending Own Social Profile to Remote SGA

A variation on the pair of abilities used to request and return a social profile was also developed and tested. In this instance, an SGA sends its social profile to another unsolicited. The approach used in both the sending and receiving abilities are the same as used for requesting and returning social profiles. The receiving agent must have the matching ability to receive and index the sent social profile. This is illustrated in figure 6.15.

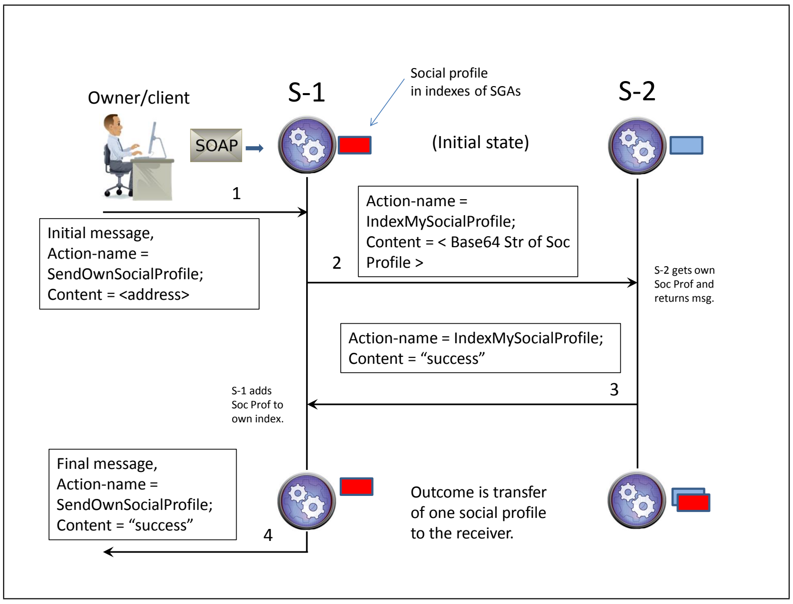

Owner/client
S-1
Social profile in indexes of SGAs
S-2
SOAP
(Initial state)
1
Action-name = IndexMySocialProfile; Content = < Base64 Str of Soc Profile >
Initial message, Action-name = SendOwnSocialProfile; Content = <address>
2
S-2 gets own Soc Prof and returns msg.
Action-name = IndexMySocialProfile; Content = "success"
3
S-1 adds Soc Prof to own index.
Final message, Action-name = SendOwnSocialProfile; Content = "success"
4
Outcome is transfer of one social profile to the receiver.

Figure 6.15: Sequence of messages for sending a social profiles from one SGA to another.

6.5.5 Querying Another SGA's Index

In the previous experiment, a social profile was requested from a remote agent by asking it to simply return its own profile from its index. The returned social profile was retrieved from the index of the responding agent based on the agent-name of that agent. To further exploit search technology in distributed agents, it is necessary to be able to request a remote agent to return one or more social profiles in response to a *query*.

This experiment is the culmination of a number of prototypes where queries of various types were sent between SGAs (recall the various query types discussed in section 5.4.4, page 96). All Lucene queries are derived from the same super class `Query`. Therefore, if one type of query can be serialised, the same technique can be used for any query. In this experiment, a More-Like-This (or MLT) query is used as an example. Recall that this type of query is created by selecting terms from a specified document which is already in an index. For serialisation and transport of the query object, a similar approach to that used to send the Lucene document objects in the previous part of this experiment is used.

In section 6.4.4, an MLT query was used to search for documents. In that experiment, the MLT query was created using a document from the same index as was used in the subsequent search. The document used to create the MLT query must be indexed in the same location as the MLT query is created. In a distributed scenario, let us assume that the requester has a social profile (i.e. the Lucene document) on which it wants to base its MLT query in its index. This presents two choices: the query can either be created using this 'local' index, or the social profile can be sent to the remote agent for indexing there. When that is done, the MLT query can be created in the remote agent. Using the capacity to send a social profile makes creating the query in either location possible.

In this experiment, the query is created using a document from the index of the requester. The query is serialised and sent to the remote agent, where it is used to search there. The execution of the overall task uses two abilities, one in either agent, with action-names `MltSocProfileRequest` and `MltSocProfileReturn`.

The sequence is as follows:

1. The requesting agent receives a message with `MltSocProfileRequest` as its action-name and a string as content. The string contains a URI or address from which to make the request and an optional integer which is the number of social profiles that are to be returned. If no value is specified by the owner for this, a default (arbitrary) value of 3 is used.

2. This ability in the requesting agent gets its own social profile from its index.

3. Using this and default parameters, a Lucene Query object is created and this is serialised to become a Base64 string.

4. A message is then created and sent to the remote agent with `MltSocProfileReturn` as the action-name and with the string as its content. The expected response

from this request is a serialised array of Lucene document objects.

5. On receipt of this message, the `MltSocProfileReturn` ability in the remote agent extracts the content and deserialises the query object (the presence of the integer value is also detected at this point).

6. The index in the remote agent is then searched using this query. A number of Lucene documents (either the default or the specified value) are then retrieved from the search results and placed in an array. The array is then serialised and becomes the content of the message returned to the requester.

7. When returned to the requester, the array is deserialised. Its elements are then added to the index of this agent.

Note that a feature of MLT queries is that the document on which they are based will be returned with the highest score. To prevent overwriting the requester's own social profile with a version from the remote agent, social profiles containing the name of the requesting agent are filtered out. This is a common practice when using MLT queries [97, p.283].

Outcome

Revisiting the second of the functionality tests set out at the start of this experiment, this was described as follows:

"Sending and receiving Lucene query objects from one agent to another."

This experiment shows that Lucene Query objects can be transported between agents and used to query a remote index. The technique of serialising a Lucene document was extended to serialise an array of such objects. This experiment demonstrates that SGA messages can also accommodate the transportation of these arrays.

Discussion

An MLT query object only differs from other Lucene queries in the way that the query is made. In the general case, once any type of query is made by the sender, it can be serialised and sent to a remote agent. The response to any query sent is an array of Lucene documents. Some key points from this experiment are:

- Sending a query from the requester leaves control of the query with the sender.
- The query created is, to some extent, governed by the content of the index in which the source document is stored, so the alternative approach of sending the 'subject' social profile to the remote agent for MLT query creation may be necessary in some circumstances.
- When considered along with the findings of Part B of Experiment 2 (page 141), the capacity to send both social profiles and MLT query objects between SGAs renders MLT queries a suitable way to search other agents' indexes. For example, an SGA may query another SGA for social profiles most similar to its own, or most similar to another SGA with which it had successful transactions in the past.

6.5.6 Calculating Cosine Similarity in an SGA

In section 5.4.5, on page 97, *cosine-similarity* was introduced. In that section, a procedure was outlined for calculating cosine-similarity between two documents in an index.

The cosine-similarity measurement is vital to this thesis as it provides a quantitative measurement for the similarity between two social profiles. This is important as it allows the similarity measurement between an agent's social profile and others to be directly compared. Pairs of social profiles which have high similarity measurements indicate that the entities that are described by the social profiles are also similar.

In this experiment, the logic used to do that calculation is integrated into a single ability called `GetSimilarity`. This ability is designed to respond to a message from either the agent's (human) owner, or to a message from another ability within an agent.

The ability receives a message which contains a URI of another remote SGA. The social profile of the agent and that of the remote agent are used in the cosine-similarity calculation. Both 'descriptive' fields of the social profile are used here, i.e. the 'own-description' and the 'third-party description'. The sequence of operations is as follows:

1. The URI from the received message is extracted. A new message with action-name `RequestIndexSocialProfile` (previously used in sub-section 6.5.4 of this experiment) is created and sent to the SGA at the URI. The social profile returned is then indexed.

2. The fields used in this comparison are 'own description' and 'third-party description'. The algorithm described in section 5.4.5, page 97 is then executed within the ability.

3. The result is then returned to the sender of the message (be that the owner, another SGA or an ability in that SGA).

Figure 6.16 illustrates the sequence of events during the process. In the simple scenario of this illustration, the message originates from an owner or client. Note that the social profile of the remote agent is transferred to the index of the requesting agent during the process.

Outcome

Finally, the third of the functionality tests set out at the start of this experiment can be recapitulated as follows:

"Using a cosine-similarity calculation to compare social profiles within SGAs."

The `GetSimilarity` sequence described in figure 6.16 was successfully tested and a cosine-similarity value obtained for the social profiles of the two SGAs used.

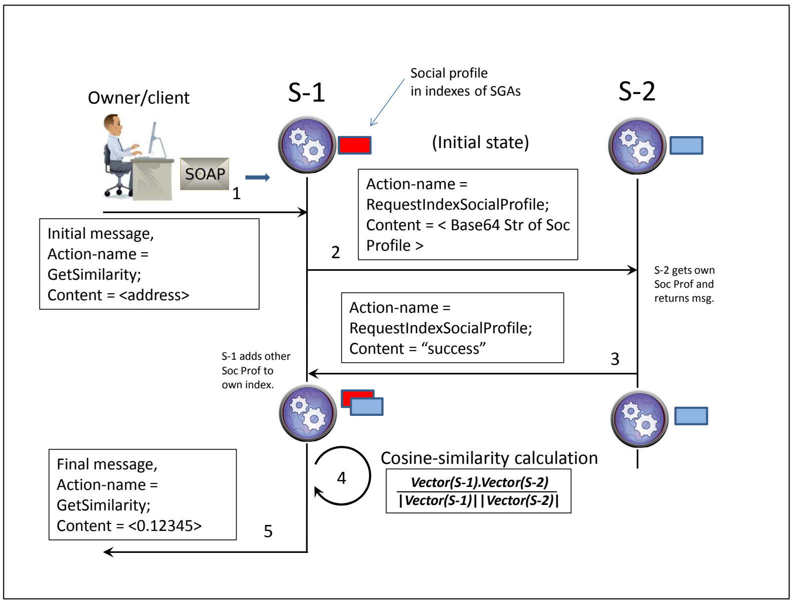

Owner/client
S-1
Social profile in indexes of SGAs
S-2
(Initial state)
SOAP
1
Initial message,
Action-name =
GetSimilarity;
Content = <address>
Action-name =
RequestIndexSocialProfile;
Content = < Base64 Str of Soc Profile >
2
S-2 gets own Soc Prof and returns msg.
Action-name =
RequestIndexSocialProfile;
Content = "success"
3
S-1 adds other Soc Prof to own index.
Cosine-similarity calculation
Vector(S-1).Vector(S-2)
|Vector(S-1)||Vector(S-2)|
4
Final message,
Action-name =
GetSimilarity;
Content = <0.12345>
5

Figure 6.16: Cosine-similarity calculation for an SGA's profile and that of a remote SGA.

6.5.7 Overall Evaluation

The question set out a the start of this experiment was: **Can the search functionality provided by Lucene in a 'single' system be migrated to a system of distributed agents?**

Three different areas of search functionality were tested in this experiment and positive outcomes were achieved for each one.

Lucene documents and query objects were sent from one SGA to another using serialisation. Furthermore, a search initiated in one agent was executed in another agent, using a completely separate instance of a Lucene index. The results for this search were returned to the requesting agent for use there.

Cosine-similarity functionality was also integrated into SGA successfully. Having transferred a previously remote social profile to an agent, a cosine-similarity measurement was carried out. This experiment demonstrates 'distributed search' in Social Grid Agents.

6.5.8 Discussion

The previously used method for serialising agent abilities for transport (i.e. Base64 encoding) has been shown to be useful for transporting software objects related to Lucene search technology. Because the tests in this experiment are carried out in a 'real' environment, error catching is desirable to check for invalid URIs and to check if the remote SGA exists and is available to process requests. During this experiment, error catching to mitigate some of these failures was incorporated to the abilities used.

6.6 Experiment 4: Using Similarity for Resource Allocation

This experiment brings together several pieces of functionality which have been examined in previous experiments. It uses information which was previously used for resource allocation in SGAs [107], namely *price* and *available credit.* The output from a calculated similarity measurement ability, developed in this thesis, is combined with this. All three pieces of information are used together to make a resource allocation decision.

6.6.1 Criteria for Success

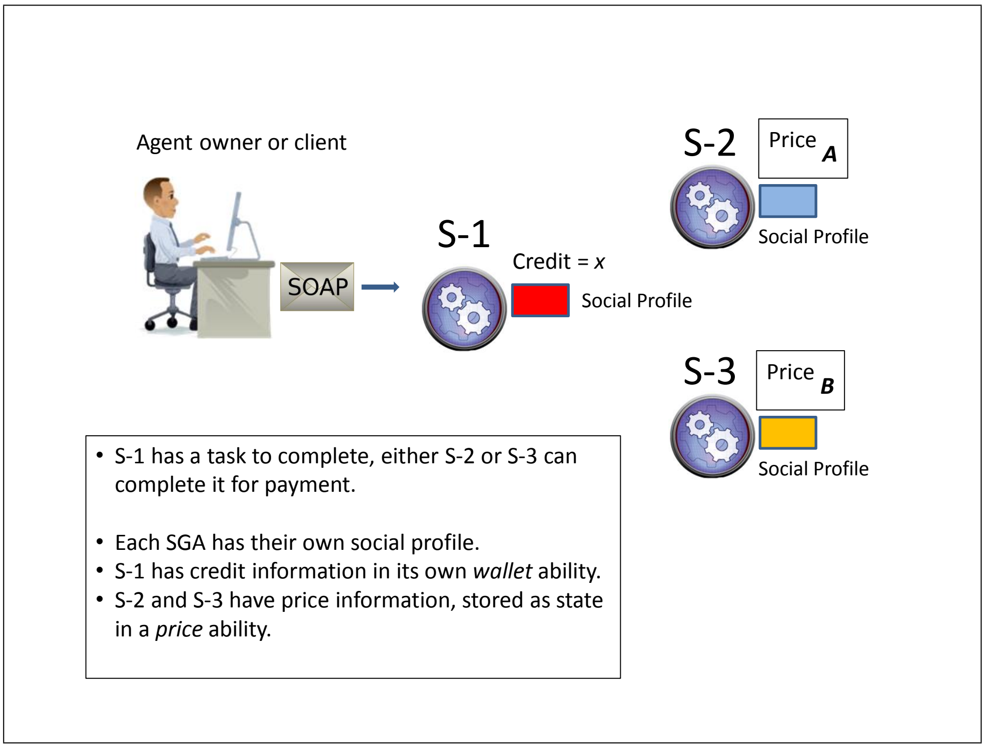
Success or failure of this experiment will be determined by the answer to the following question:

“Can the abilities developed in this thesis be brought together and used, alongside some already existing SGA abilities, to address a simple resource allocation decision?”

6.6.2 Experiment Description

Consider a small SGA system of three SGAs like the one in figure 6.17. Agent S-1 requires the services of another agent to complete its task. It has an allocation of credit in its wallet. Two other agents in the system S-2 and S-3 are able to provide this service. Each of these contains two pieces of information which are relevant to agent S-1’s task allocation decision making process. These are their respective prices that they need to be paid for their services and a description of themselves in the form of their social profiles. In figure 6.17, relationships such as ‘simple purchase’ or ‘direct control’ as seen in figure 6.3 are not shown.

For this experiment, three minimal SGAs were created and deployed in separate VMs. The MAS creation mechanism described in section 4.3, page 59, and examined in section 6.3 in this chapter was used for this.



- S-1 has a task to complete, either S-2 or S-3 can complete it for payment.
- Each SGA has their own social profile.
- S-1 has credit information in its own *wallet* ability.
- S-2 and S-3 have price information, stored as state in a *price* ability.

Figure 6.17: SGA multiple-agent system for using similarity for resource allocation.

6.6.3 Abilities Required

Two new SGA abilities which were developed for this experiment are outlined here.

6.6.4 Evaluation of Price and Similarity

The *Price and Similarity Evaluation* ability (with action-name `PriceSimilarityEval`) is added to agent S-1. It receives addresses of candidate agents which can, for example, provide a service for it. It uses these addresses and a number of other abilities to compile a matrix of information about the remote agents and uses this matrix to make a work allocation decision. This ability is configured as an *instance* as it is required to maintain some state which is used in the decision making process.

Request of Price Details

A *Price* ability is configured in S-2 and S-3, the 'candidate' agents which will be considered in the evaluation process. This ability was developed for this experiment. It allows agents S-2 and S-3 to hold state about what price they need for their services, and also allows this price to be adjusted without reconfiguring the agent. Similar to how the wallet ability is configured in Experiment 1 in section 6.3, the price ability is added to the agents' data-structures three times, one for each of the following action-names:

1. QueryPrice
2. DecreasePrice
3. IncreasePrice

In a real world scenario, these agents would only respond to messages with `DecreasePrice` or `IncreasePrice` as their action-names when such messages were sent by their owners. The original implementation of SGAs provides for this type of restriction [107].

Previously Used Abilities

Many other abilities which were used in previous experiments are employed here, these include:

- `RequestIndexSocialProfile` and `ReturnSocialProfile` (examined in section 6.4).
- The `Wallet` ability (already in SGAs [107]).
- The `GetSimilarity` ability (examined in section 6.5.6).

6.6.5 Information Matrix and Decision Logic

In this prototype, two remote SGAs are considered as candidates for the work allocation and a matrix of information is compiled about them. An example of the information matrix is shown in table 6.18.

As an example of how this matrix may be used, a simple evaluation logic is implemented. A preset price difference threshold is set in the agent ability. The preset value is in effect a 'policy' of the agent S-1. This can be adjusted by the owner as required using the same approach as used to adjust the price in the `Price` ability.

	Price	**Similarity with agent S-1**
S-2	PriceA	0.12345
S-3	PriceB	0.54321

Table 6.18: Matrix of information gathered by the PriceSimilarityEval ability for price and similarity evaluation.

The implemented evaluation logic is as follows: If the price difference between S-2 and S-3 is greater than the preset 'price threshold' value, then choose the lower price, regardless of similarity measurements. If the difference between the prices is less than the preset value, then choose the agent which is closer in similarity to S-1. A pseudocode representation of this (in a procedural language) is illustrated in figure 6.19.

```
// sample information
Price Threshold = 2
Price A = 7
Price B = 8
Similarity with A = 0.12345
Similarity with B = 0.54321

if (Difference in Price A and Price B is greater than Price Threshold) then

     Chose the provider (A or B) with the lowest price

else

     Chose the provider with the highest similarity measurement
```

Figure 6.19: Logic for resource selection based on price and similarity measurements.

6.6.6 Sequence for Price and Similarity Evaluation

The ability `PriceSimilarityEval` executes the following sequence:

1. It receives a message which contains an array of two URIs for remote SGAs.

2. Agent S-1 contacts S-2 and S-3 to request their price information.

3. The address of each agent is then passed to the `GetSimilarity` ability (described in section 6.5.6 of this chapter). This ability, having retrieved the social profile for the given agent, calculates and returns a cosine-similarity measurement for the agent referenced by the address. This is done for both addresses available.

4. Having received price and similarity information for both candidate agents, the information gathered is used in a Classad expression. This expression contains logic which determines which of the two remote SGA is selected (this pre-programmed logic is discussed in the next section).

Figure 6.20 on page 165 illustrates this sequence.

6.6.7 Using Price and Similarity Values in a Classad Function

In keeping with the SGA original design, the logic of figure 6.19 is implemented using the Classad functional language. Figure 6.21 shows this. This expression evaluates to a reference to either agent S-2 or S-3 and uses a combination of ternary conditional operators and Boolean selections. It is shown here to demonstrate how the values from the matrix in table 6.18 can be used in SGAs.

6.6.8 Outcome

A message was sent to agent S-1 which invoked the ability `PriceSimilarityEval`. The sequence of events described in figure 6.20 executed without any errors and the experiment was a success.

6.6.9 Evaluation

Let us return to the question set out at the start of this experiment: **Can the abilities developed in this thesis be brought together and used, alongside some already existing SGA abilities, to address a simple resource allocation decision?**

The abilities described in this experiment were combined and used together successfully. These abilities incorporated similarity measurements between two different social profiles into a resource allocation decision which also took consideration of available credit and price information.

6.6.10 Discussion

The mechanism demonstrated here is a 'proof of concept'. It gathers two different types of information which are relevant to resource allocation decisions, price and descriptive similarity. The outcome of the decision made using this data can then be used in other existing SGA abilities, and by doing this, similarity measurements between SGAs can be used to influence resource allocation. A short Classad function is used here to show how these values can be used to determining which SGA will be chosen for the next stage of the allocation process. Further work on resource allocation is not explored here as this was examined in detail in the 2008 work of Pierantoni [107].

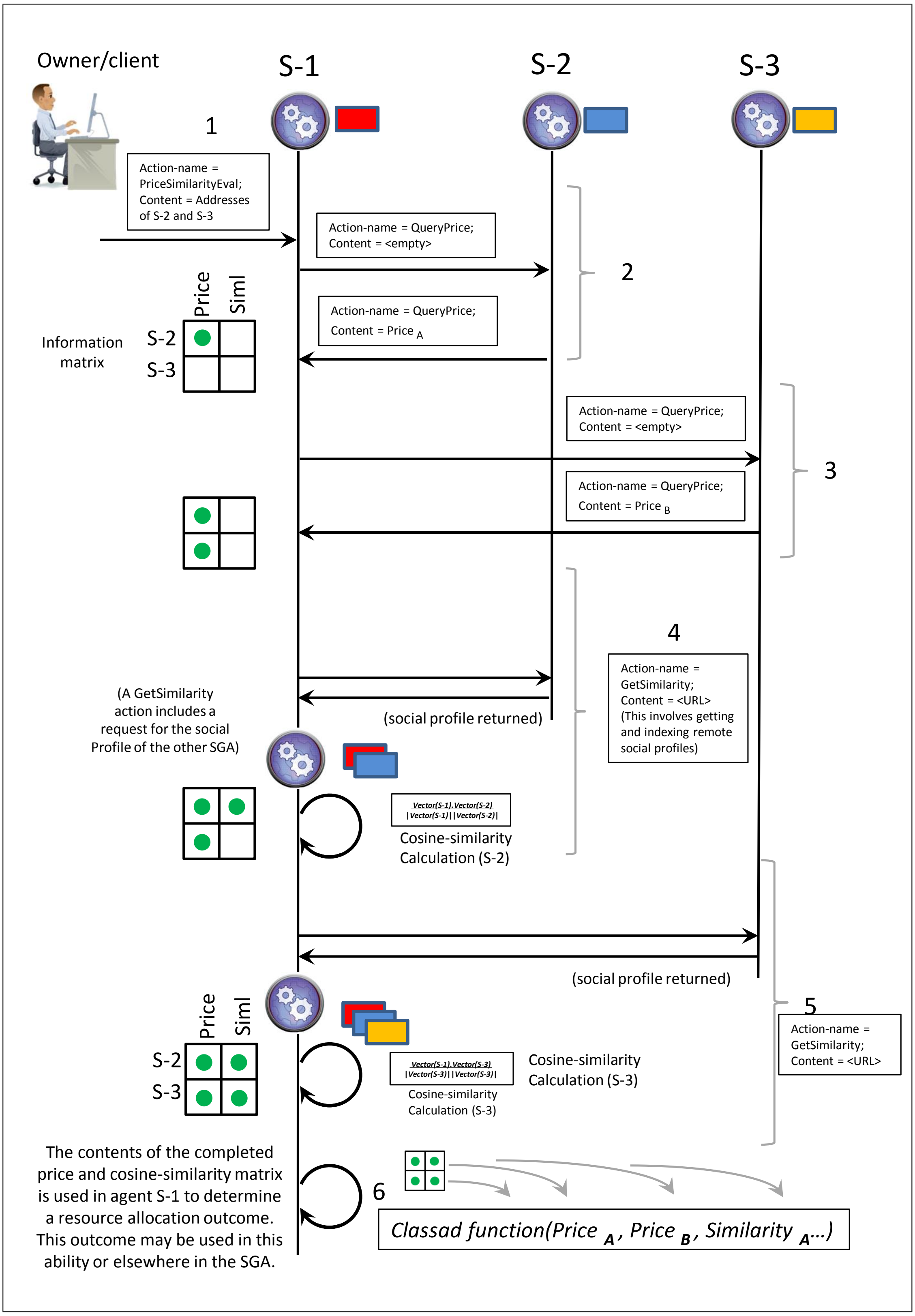
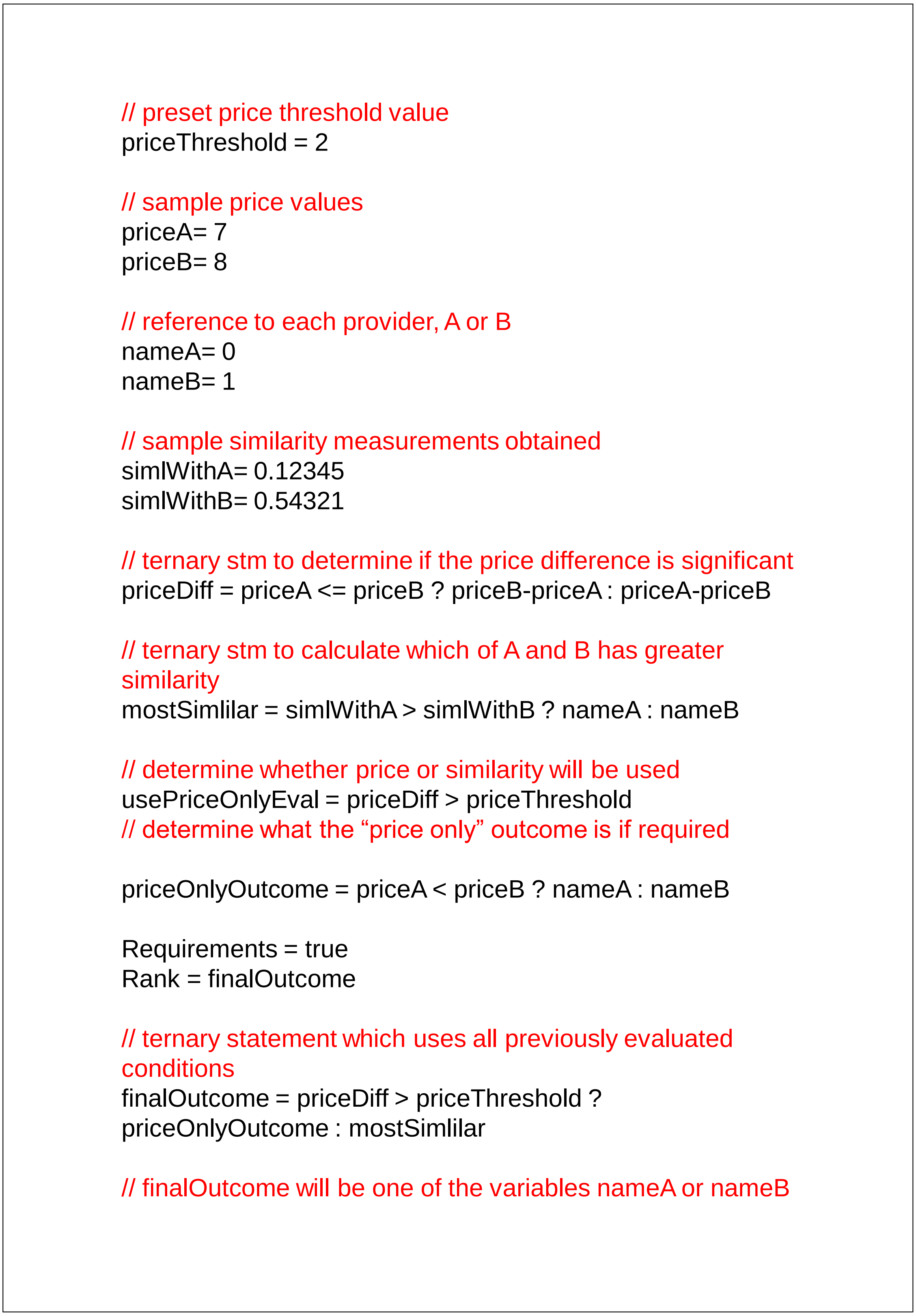

Owner/client
S-1
S-2
S-3
1
Action-name = PriceSimilarityEval; Content = Addresses of S-2 and S-3
Action-name = QueryPrice; Content = <empty>
2
Price
Siml
Information matrix
S-2
S-3
Action-name = QueryPrice; Content = Price A
Action-name = QueryPrice; Content = <empty>
3
Action-name = QueryPrice; Content = Price B
4
Action-name = GetSimilarity; Content = <URL> (This involves getting and indexing remote social profiles)
(A GetSimilarity action includes a request for the social Profile of the other SGA)
(social profile returned)
Vector(S-1).Vector(S-2) / |Vector(S-1)||Vector(S-2)|
Cosine-similarity Calculation (S-2)
(social profile returned)
5
Action-name = GetSimilarity; Content = <URL>
Price
Siml
S-2
S-3
Vector(S-1).Vector(S-3) / |Vector(S-3)||Vector(S-3)|
Cosine-similarity Calculation (S-3)
Cosine-similarity Calculation (S-3)
The contents of the completed price and cosine-similarity matrix is used in agent S-1 to determine a resource allocation outcome. This outcome may be used in this ability or elsewhere in the SGA.
6
Classad function(Price A, Price B, Similarity A...)

Figure 6.20: Sequence for assembling information needed for price and similarity evaluation.



```
// preset price threshold value
priceThreshold = 2

// sample price values
priceA= 7
priceB= 8

// reference to each provider, A or B
nameA= 0
nameB= 1

// sample similarity measurements obtained
simlWithA= 0.12345
simlWithB= 0.54321

// ternary stm to determine if the price difference is significant
priceDiff = priceA <= priceB ? priceB-priceA : priceA-priceB

// ternary stm to calculate which of A and B has greater
similarity
mostSimlilar = simlWithA > simlWithB ? nameA : nameB

// determine whether price or similarity will be used
usePriceOnlyEval = priceDiff > priceThreshold
// determine what the “price only” outcome is if required

priceOnlyOutcome = priceA < priceB ? nameA : nameB

Requirements = true
Rank = finalOutcome

// ternary statement which uses all previously evaluated
conditions
finalOutcome = priceDiff > priceThreshold ?
priceOnlyOutcome : mostSimlilar

// finalOutcome will be one of the variables nameA or nameB
```

Figure 6.21: Classad function statements to implement the logic of figure 6.19.

6.7 Experiment 5: External JARs Over HTTP

Throughout chapter 4, and again and in Experiment 1 (section 6.3), significant emphasis is placed on being able to create and deploy an SGA, and subsequently add further abilities to that SGA as required. Experiment 1 demonstrates and examines this concept in action. Developing the capacity to do this was the original motivation for using a *basic-agent* in conjunction with a mechanism to configure that agent at a later time (as described in section 4.2, page 56). However, in Experiments 3 and 4 (sections 6.5 and 6.6), when integrating search technology to an SGA, additional libraries (JAR files) were required in this basic-agent. These JARs were added to the basic-agent code base (see section 6.5.3), the one used to build each of the SGA application packages (i.e. the WAR files used when creating the MAS used in Experiment 3). Having to add these additional JARs 'breaks' the concept of being able to add abilities *without* stopping and rebuilding the agent. This is a limitation of the mechanism examined in Experiment 1.

6.7.1 Experiment Description

The limitation of needing to add additional JARs was not exposed in Experiment 1 as all the required classes and JARs were already available in the basic-agent. Adding a new technology, as done in Experiment 3 and 4, exposed this limitation, and for expediency, the JARs were added to the basic-agent package for that work. This experiment introduces and demonstrates a solution to this limitation. It examines using a JAR located on the HTTP server which can be accessed by the SGA over the IP network.

Just as an ability can be added to an agent after it has started running, a JAR file can be placed on an accessible HTTP server which is completely independent of the SGA. If the JAR is needed by methods in such an ability, it must only be in place before the methods that it supports is invoked.

Java provides a URL class loader to access class binaries over the network. This can load one or a number of classes stored in a directory or all the classes from one or more JAR files by 'fetching' them over a network. Once a remote class is loaded, and assuming prior knowledge of its name, methods and their signatures, Java Reflection can be used to invoke its methods. All this happens without the binary of the class being stored on the local system (as is the usual case).

6.7.2 Criteria for Success

Success of this experiment depends on a positive answer to the following question:

"Can the same functionality seen in Experiment 4 be recreated in an SGA which relies on JARs accessed over the IP network?"

The following work will determine this answer. Success in using an 'external' library means that a wider range of abilities can be added to SGAs without requiring that the agent be stopped for insertion of additional JARs or classes.

6.7.3 Procedure for Experiment

The experiment aims to reproduce the same functionality observed in Experiment 4. As before, all functionality is tested in separate virtual machines, including that of the HTTP server. Two steps are needed, the removal of the `commons-math` JAR from the package, and the development of the modified ability which uses a URL class loader.

Removal of Math JAR from WAR File

For this experiment, the changes made to the code base for Experiment 4 were reversed. The `commons-math-2.2.jar` was once again removed from the list of JARs to be included in the basic-agent package. On building a WAR file from this package, the absence of this JAR was verified by listing the contents of the WARs created.

Development of a 'No JAR' Ability

The ability `GetSimilarity` in agent S-1 of figures 6.16 and 6.17 in Experiment 4 was renamed and modified to become `GetSimilarityNoJar`.

While the logic and outward functionality of the ability remained the same, the internal code was modified to use a URL class loader to access some classes. The changes to that code are described here.

The cosine-similarity calculation within that ability uses three classes and methods from the `commons-math` JAR. One of these classes, `OpenMapRealVector` is used here as an example to illustrate the different approach needed when loading a class from a remote JAR. This is then compared to that of the usual and 'conventional' approach, i.e. when the class is available locally.

In terms of Java syntax, for the conventional scenario, classes are 'imported' at the top of a class and instantiated by calling the `Constructor` method for each class as needed. Following this, methods can then be invoked on each instance. The segments of code which do this are shown in figure 6.22. By contrast, to implement the same logic where the `OpenMapRealVector` class is loaded from a URL class loader, the (longer) syntax required is illustrated in figure 6.23. Notice in figure 6.23, a `Method` object has to be created for each method which is invoked. This object is then used to invoke a method of the instance.

```
// import at top of class, Class is visible one of the available class loaders.
import org.apache.commons.math.linear.OpenMapRealVector;

// instantiate an object
OpenMapRealVector vectOne = new OpenMapRealVector(termsHmapAll.size());

// invoke a method on the object, arguments are ‘int’ and ‘double’
vectOne.setEntry(position, frequency);
```

Figure 6.22: Conventional Java Code with Import of Classes.

Programming for using classes which are only available via a network class loader is inherently more verbose and complicated when compared to code which uses 'locally available' classes. Methods are called on classes where the method signatures

```
 // Note: OpenMapRealVector is NOT imported at the top of this class.

 // Get the current class loader of the application
 ClassLoader contextClassLoader = Thread.currentThread()
                                                .getContextClassLoader();

 // use this class loader to load all classes in the remote JAR
urlLoader = new URLClassLoader(new URL[] { new URL(
    "http://123.123.123.123/classes/commons-math-2.2.jar") },
                                           contextClassLoader);

 // the class for OpenMapRealVector is defined
 omrvCL = Class.forName(
         "org.apache.commons.math.linear.OpenMapRealVector",
                                             true, urlLoader);

  // an instance of one of its constructors is instantiated
 Constructor<?> ctor  = omrvCL.getConstructor(int.class);

  // the constructor instance is used to create an instance
  // of the class to which the constructor belongs, i.e. omrvCL
  vectOne = ctor.newInstance(termsHashmap.size());

 // this gets a method of this class of vectOne, i.e. omrvCL
 Method setEntry = omrvCL.getMethod("setEntry",
                           new Class[] { int.class, double.class });

 // invoke the method, arguments are an ‘int’ and ‘double’
 setEntry.invoke(vectOne, new Object[] { position, frequency });
```

Figure 6.23: Java Code for Remote Classes.

are specified ‘blindly’ by the developer. The method names and signatures specified must match those of the actual classes used. For this reason, and because of unreliable networks and other opportunities for errors, Java URL class loading and Reflection code is surrounded by several exception and error catching braces.

Figure 6.24 shows an SGA with the modified `GetSimilarityNoJar` ability. The

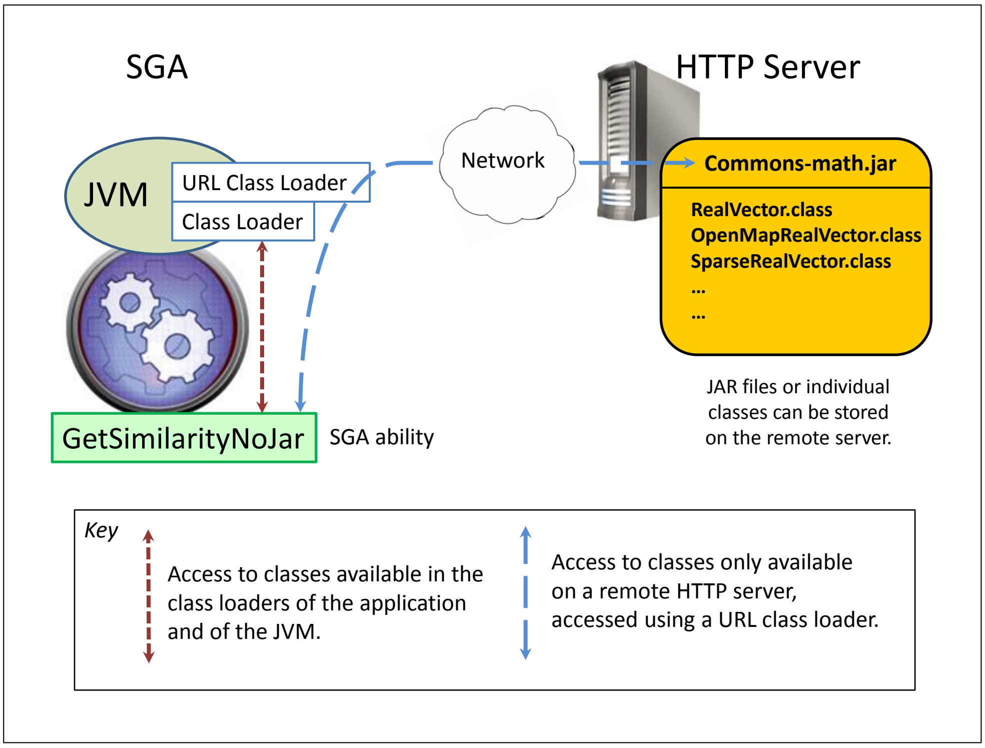

SGA
HTTP Server
JVM
URL Class Loader
Class Loader
Network
Commons-math.jar
RealVector.class
OpenMapRealVector.class
SparseRealVector.class
...
...
GetSimilarityNoJar
SGA ability
JAR files or individual classes can be stored on the remote server.
Key
Access to classes available in the class loaders of the application and of the JVM.
Access to classes only available on a remote HTTP server, accessed using a URL class loader.

Figure 6.24: SGA access to a remote JAR file using URL class loader.

SGA in figure 6.24 corresponds to agent S-1 in figure 6.17, the system used in Experiment 4. This figure shows the 'access routes' the ability uses to load the classes it needs to execute. The majority of classes will be available in the class loaders of the JVM. For the classes required from the JAR `commons-math`, the code of the ability calls for classes to be loaded using the URL class loader.

6.7.4 Running the Experiment

A small multiple-agent system of agents S-1, S-2 and S-3, the same as that in Experiment 4 was created (see figure 6.17), using the MAS creation mechanism as before. The newly developed ability `GetSimilarityNoJar` was used in agent S-1 and replaced the `GetSimilarity` used in Experiment 4.

Before the required JAR was put in place on the HTTP server, the first cycle of the experiment attempted to invoke the agent S-1. This failed as expected. This was done to ensure that the required classes were not inadvertently being made available due to caching or a programming error.

Having made the required `commons-math` JAR available on the HTTP server, the test was repeated, this time with a successful outcome. That is to say, the same functionality and behaviour was observed as was seen in Experiment 4 and the test ran without any errors or failures.

Access to the HTTP server was observed in the logs for that system, a sample of which is available in figure 10.1 in appendix J, page 210. Examination of that figure shows the initial failed attempt to fetch the `commons-math-2.2.jar` file (causing a 404 error) and two further successful responses for the required JAR file. Also in figure 10.1, prior to these entries, the request for the abilities XML file and social profile XML file can be seen.

6.7.5 Evaluation and Contributions

Returning to the question posed at the start of this experiment: **Can the same functionality seen in Experiment 4 be recreated in an SGA which relies on JARs accessed over the IP network?** The outcome of this experiment has shown that it can.

Successfully using classes from a JAR file located outside of an SGA is a viable alternative to adding JARs to basic-agent package (as is done in Experiments 3 and 4). The URL class loading technique used in this experiment enhances the scalability, maintainability and flexibility of SGAs. This is most significant where it may be required to add 'new' abilities to an existing and already configured agent, but where it is undesirable to stop and rebuild the underlying basic-agent.

Although the `commons-math` JAR was chosen for this experiment, the Java objects used from the Lucene JARs could also have been examined in this experiment. In this thesis, however, many more Lucene classes are used than the three required from the `commons-math` JAR in this experiment. For that reason, the Lucene JARs have been permanently incorporated into the basic-agent package.

6.8 Experiment 6: Using Includes URLs in ACFs

In Experiment 1 (section 6.3), the abilities configuration file (ACF) played a central role in configuring a small multiple-agent system. In section 4.8 on page 81, the idea of including URLs within the ACF was introduced. The function of these 'included' URLs (or links) is to direct an agent to further ACFs. The reader is invited to view again figure 4.11 on page 82 which illustrates the XML schema used for this.

This experiment is a variation on Experiment 1 in section 6.3 where configuring each SGA using just one ACF per agent was tested. From a single URL sent to an agent in a message, an ACF is retrieved and then processed, adding varying numbers of binary abilities to each agent. Although, the prototype of the 'includes' feature was tested and reported in chapter 4, this experiment tests this concept in the same multiple-agent system as was used in the experiment of section 6.3.

6.8.1 Criteria for Success or Failure

Success or failure of this experiment can be determined by answering the following two questions:

1. "Can the multiple abilities configuration files specified in the 'includes' section of the initial abilities configuration file specified for an agent be used and processed without errors or failures?"

2. "Can the same configuration and behaviour achieved in Experiment 1 be achieved using abilities added using the 'includes' approach?"

Because the system recreated here ultimately must be the same as that created in Experiment 1, the same criteria for success set out in Experiment 1 is used here to answer the above questions in this experiment.

6.8.2 Experiment Description and Methodology

Two aspects of using 'included' abilities configuration files are examined, these are set out as parts (A) and (B) next. Two of the eight SGA's abilities configuration files are selected and manually altered. For parts (A) and (B), these alterations and new files created are as follows:

Part (A), distributing abilities over several files: Agent S-1 in figure 6.3 is configured with 11 abilities. This set of abilities is broken up and configured using 5 separate abilities configuration files; figure 6.25 illustrates this. The abilities are distributed so that at least one file has a mix of abilities and included URLs, i.e. file (A) in figure 6.25. Another file has zero abilities and a number of 'included' URLs, this is file (B) in figure 6.25. The remaining files contain abilities only, ranging in number from 1 to 3. It can be said that the various files are linked together to form a chain, all parts of which contain the complete configuration for the agent.

Part (B), reusing an abilities configuration file for multiple agents: Agents P-3 and P-4 of figure 6.3 share a number of common abilities. In this short and simple experiment, the common abilities are moved to an abilities configuration file which is then 'included' for use by both agents in their respective abilities configuration files. This is illustrated in figure 6.26.

6.8.3 Outcome

This test was successful. All three agents were configured correctly and functioned with the same behaviour as seen in Experiment 1. Part of the S-1 agent's output can be inspected in appendix I on page 208. This was retrieved from the output log in the web-service container. In this (abridged) output, it can be seen that the agent S-1's configuration is interspersed with access to various remote abilities configuration files. For demonstration purposes the agent printed to log, details of each recursive call made. The actual file names used in figure 6.25 (e.g. `lcg2wmssga1_abilities_b.xml`) can be seen in this output.

Inspection of the print-out from the final configuration showed that this S-1 agent was configured to be the same as that of the S-1 agent in the system created in section 6.3, albeit with abilities added in different orders (which is unimportant). Further

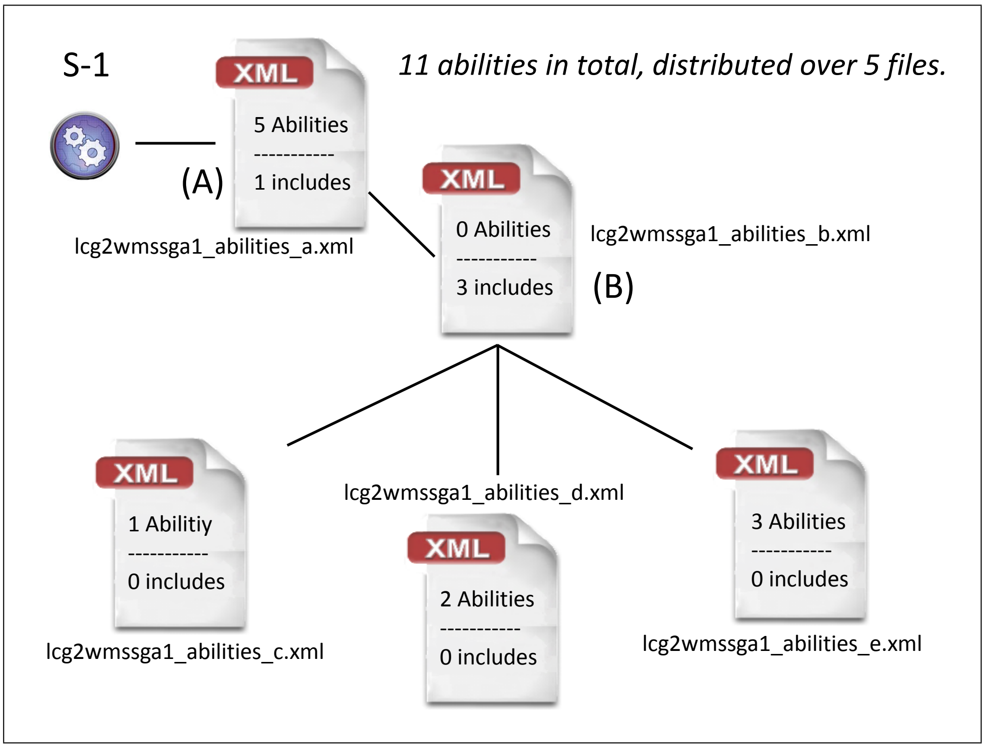

S-1
XML
5 Abilities
----
1 includes
11 abilities in total, distributed over 5 files.
(A)
lcg2wmssga1_abilities_a.xml
XML
0 Abilities
----
3 includes
lcg2wmssga1_abilities_b.xml
(B)
XML
1 Abilitiy
----
0 includes
lcg2wmssga1_abilities_c.xml
lcg2wmssga1_abilities_d.xml
XML
2 Abilities
----
0 includes
XML
3 Abilities
----
0 includes
lcg2wmssga1_abilities_e.xml

Figure 6.25: Scenario for distributing all 11 required abilities over several ACFs.

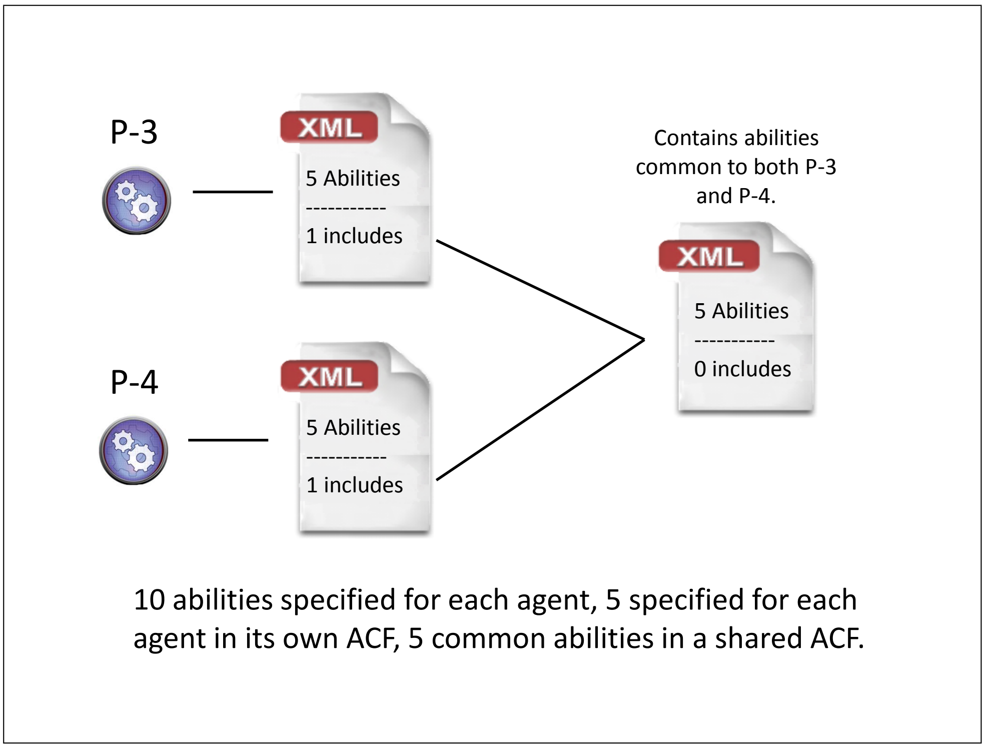

P-3
XML
5 Abilities
----
1 includes
Contains abilities common to both P-3 and P-4.
XML
5 Abilities
----
0 includes
P-4
XML
5 Abilities
----
1 includes
10 abilities specified for each agent, 5 specified for each agent in its own ACF, 5 common abilities in a shared ACF.

Figure 6.26: Scenario for sharing ACF abilities files between agents.

functionality tests were carried out by sending messages to agent S-1 as done in Experiment 1. The behaviour observed was the same as that seen in Experiment 1, detailed on page 123. This means that the agents with the altered abilities configuration files were correctly configured and that the ‘includes’ mechanism worked as it was intended to.

6.8.4 Evaluation

In light of this outcome, the two questions set out at the start of this experiment can now be revisited:

1. “Can the multiple abilities configuration files specified in the ‘includes’ section of the initial abilities configuration file specified for an agent be used and processed without errors or failures?”

2. “Can the same configuration and behaviour achieved in Experiment 1 be achieved using abilities added using the ‘includes’ approach?”

The answer to both these questions is ‘yes’. The outcome of this experiment shows that the two agents tested were configured as expected. These agents used a number of ACFs which were only specified as ‘includes’ (or links) in their initial ACFs. The same configuration, functionality and behaviour was achieved as was seen in Experiment 1, section 6.3.4, where only one ACF was used in each agent.

6.8.5 Other Prototype Tests for Included URL in ACFs

Some further tests from the prototype development of this ‘include’ feature are briefly discussed here. These look at the implications of accidentally creating a circular recursive loop and how this can be mitigated. Two approaches were tested which limit the impact of this happening during an SGA configuration as described in section 4.8.

Catching Stack Overflow Errors

If the number of recursive calls is allowed to increase indefinitely, JVM memory resources will eventually become exhausted, causing a `StackOverflow` error. However, where anticipated, this exception can be caught and dealt with in a graceful manner. Such an event should also alert the agent owner to this failure.

Limiting the Number of Recursive Calls

Alternatively, the recursion can be programmed to attempt no more than a defined number of recursive calls. To examine this, a test was carried out by deliberately creating a circular reference. This resulted in a stack overflow error after 5024 recursive calls[6]. By keeping a count of recursive calls made, recursion can be programmatically stopped before such a high figure is reached. However, it is not envisaged that such a large number of referenced URLs would be necessary, even for a very large and complex MAS. Nonetheless, this result is a suitable guideline for the limit which can be predefined in the `FetchProfile` ability used for this.

6.8.6 Discussion

Proof of the 'include' concept was important as it has positive implications for planning and creation of multiple-agent systems. The chains of abilities configuration files tested in part (A) allows a large and unwieldy ACF to be broken down into smaller parts. This is a contribution to scalability of SGAs, or any multiple-agent system which would find this mechanism suitable. Part (B) facilitates the reuse of what may become large blocks of 'tried and tested' abilities in the future. This is a similar concept to importing 'tried and tested' libraries when programming in Java or C++, i.e. there is no need recreate things that are commonly and frequently used. This too makes a contribution to the scalability of SGAs. The combination of these two contributions should also simplify planning, programmability and maintenance of a large agent system.

[6] The value 5024 is related to the available resources of the underlying hardware and will vary.

6.9 Experiment 7: Profile Structure Class

In section 5.5.4, page 104, a solution was proposed to allow a social profile file to be extensible without having to constantly alter the binary ability used to index it. An *encapsulated binary schema* in the form of a serialised Java class was introduced. The class used for this is called the `ProfileStructure` class and its workings are illustrated in figure 5.5 on page 107.

Recall that the ProfileStructure class is a class which is deserialised by the logic of an agent's ability. Prior to this, that ability itself is retrieved from the agent's internal data-structure. This binary ability has been previously put in the agent's data-structure after being deserialised from a serialised object (in the form of a Base64 string) contained in the ACF (recall section 4.3.2 on page 60). While in theory, the ProfileStructure class is just like any other class loaded into the web-service container class loader, this long chain of events which lead up to its instantiation and use make it is necessary to test its operation in a real web-service container in a virtual machine. This experiment is carried out in that environment.

For clarity, it may be helpful to review figure 5.5, page 107 for details about the interactions between the ability `FetchProfSocial` and the ProfileStructure class.

6.9.1 Objectives of Experiments

This experiment examines the use of the ProfileStructure class in areas of gathering local information in the underlying host system of the SGA and the scalability of the ProfileStructure class. These are both attributes which will be objectively assessed. Furthermore, known limitations, requirements, failure scenarios and failure behaviour when using the ProfileStructure as an *encapsulated binary schema* are discussed. The experiment is concluded by highlighting the overall contributions of the encapsulated binary schema, including those related to the outcomes of this experiment.

Criteria for Success

Success or failure for the objective parts of this experiment depends on whether or not the ProfileStructure can be tested without any errors. Such errors would prevent it being used in the ways described. More specifically, the above two objectives can be evaluated by asking the following questions:

1. Is it feasible to task the ProfileStructure class with gathering information only known 'locally' to the social grid agent and its host system with the intention of adding that information to the social profile at indexing time ?

2. What are the scalability implications for increasing the number of elements which a social profile XML file contains ?

These two questions are addressed in turn in the following two sub-sections (6.9.2 and 6.9.3).

6.9.2 Adding Local Information to the Indexed Social Profile

The following tests examine whether the `ProfileStructure` class can be used to 'sense' an agent's environment and gather 'local' information to further add to what has been transported to the agent in the social profile XML file. The benefit of being able to do this is that several pieces of information which can be 'sourced' within the agent can then be added to the agent's Lucene document as it is prepared for indexing in that agent.

Using locally sourced system information is preferable to attempting to predict what this information may be, say for example, when compiling the descriptive part of a social profile XML file in the domain of the agent-owner. Information gathered in real-time and directly from its definitive source is much more reliable and therefore is desirable.

By extension of this, the agent's social profile then becomes a combination of information which the creator placed in the social profile XML file, and other information which can be obtained from the host system. A positive outcome to these tests will will answer question 1 in the criteria set out at the start of this experiment.

Methodology

Recall from section 5.5.5 that data from the social profile XML file is transferred to fields in the ProfileStructure class. When this transfer is complete, another ProfileStructure class method is called which transfers the data in these fields to the fields of the Lucene document. This point in the procedure presents a suitable opportunity to add further logic which can source local information for addition to the Lucene document. This method in the existing ProfileStructure was modified to include code to get system information and read a number of system files. The data and files tested were as follows:

1. Code was added to determine the IP address of the system which hosts the agent using the Java `InetAddress` class. The combination of the IP address and the agent's name can be used to provide the URI of the agent.

2. Code was added to read system files which contain and possibly describe the hardware or operating-system (OS), the files to be read are as follows:

   (a) `/etc/redhat-release` - this contains the name of operating-system on the particular type of system used for this test.

   (b) `/proc/cpuinfo` - this file describes the CPU(s) available in the system.

   (c) `/proc/version` - this file contains kernel and more comprehensive OS description.

   (d) `/proc/scsi/scsi` - a file with details of storage devices available on the system.

Files under the `/proc` directory are of particular interest as they are rich with details about the underlying system. These files are in fact a pseudo-file system and are created in memory only. That is to say that they merely 'appear' as files but are never written to disk and therefore are, for example, not present after the system has shutdown.

The logic used in the ProfileStructure class added all the gathered information to the Lucene document using the `NOT_ANALYSED` and `Field.Store.YES` options, but this can easily be changed in the ProfileStructure class as required.

Testing

To examine the outcome of this experiment, 'print-out' code (which results in log entries) was added to the `FetchProfSocial` ability used. This code read data from the completed Lucene document after it was returned to this ability by the ProfileStructure class figure (i.e. after step 6 in figure 5.5, page 107). Apart from this added code, no other functional change was made to the `FetchProfSocial` ability used here (the same ability used to index social profiles in Chapter 5 and in Experiment 2, section 6.5 of this chapter).

This modified ProfileStructure class was compiled, its class file was then serialised to become a Base64 string and was added to an otherwise typical social profile. This social profile was then indexed in a 'real' SGA, deployed in a web-service container in a VM.

Outcome and Evaluation

Question 1 set out at the start of this experiment can be answered by evaluating the outcome of this part of the experiment. The question is as follows:

"Is it feasible to task the ProfileStructure class with gathering information only known 'locally' to the social grid agent and its host system with the intention of adding that information to the social profile at indexing time?"

The presence of both the agent's URI and the contents of the system files in the Lucene Document was verified by using the 'print-out' code described above. The IP address of the machine was successfully obtained and combined with the already available agent-name from the SGA. This allows the 'actual' URI address of the SGA to be added to the Lucene document. The presence of content from the host's file system files obtained from both from `/etc` and `/proc` directories in the Lucene Document was also verified.

Local information, read from the host system of the SGA (its definitive source) was added to a Lucene document and subsequently added to the social profile of the SGA as part of its description. Therefore, using the ProfileStructure class to do this was successful and is feasible.

6.9.3 ProfileStructure Class Size

This short experiment examines the impact of adding more elements to the social profile XML file, thereby requiring more logic to be added to the ProfileStructure class. The increase in size of the ProfileStructure class is assessed here.

A series of ProfileStructure classes are compiled with the capability to process XML files having elements ranging from 10 to 40 in number. For each additional element in the social profile XML file, the ProfileStructure class requires an additional variable (or field member), a setter method for that variable, and some lines of code which add the field's content to the Lucene document. The sizes of serialised objects produced (in the form of Base64 string text files) as a result of the additional code are recorded.

ProfileStructure Class Size Measurements

The sizes (on disk) of the class required as the number of XML file elements was increased are presented in table 6.27. With this and other files tested in this thesis, serialisation by conversion from the Java binary format to Base64 carried a linear overhead of 35% in file size.

No. of XML Elements being processed	Class file size (KB)	Base64 string size (KB)
10	3.87	5.2
20	5.40	7.3
30	6.96	9.3
40	8.54	11.5

Table 6.27: ProfileStructure File Sizes Compared to Number of Elements in XML

6.9.4 Evaluation

Using the information of table 6.27, the second of the two questions set out in this experiment can be addressed:

“What are the scalability implications for increasing the number of elements which a social profile XML file contains?”

From table 6.27, the sizes of the Base64 strings produced by serialising ProfileStructure classes are between 1% and 10% of the sizes of social profiles which used Experiment 3 (section 6.5). Based on the file sizes used and successfully indexed in Experiment 3, the increase in size of the serialised class is not substantial when compared to the size of the social profile XML files which typically contain them. For completeness, an XML file with 40 elements was also created. Using its corresponding ProfileStructure class, it was successfully indexed. This took place in an SGA in the ‘real’ VM and web-service application environment and worked without error. No scalability issues were observed when extending the social profile XML file and ProfileStructure class up to the sizes shown in table 6.27.

6.9.5 Discussion

The following discusses the outcome of both tests as well as some of the known limitations, requirements, failure scenarios and failure behaviour when using the `ProfileStructure` as an encapsulated binary schema.

Generality and Limitations of the ProfileStructure

The name of the `ProfileStructure` class is not hard-coded in the ability class which uses it. A class of any name could be encapsulated in the social profile as long as its methods which accept and return the Lucene document match those of the ability which uses it.

The ‘setter method’ names used in the `ProfileStructure` class are composed by prepending the word ‘set’ to the element names found in the social profile XML file.

These methods are accessed using Java Reflection. This approach allows a String object to 'name' a method, and then invoke that method, provided it is present in the ProfileStructure class. The XML element names may vary, but as long as the ProfileStructure class is coded correctly, the SGA ability `FetchProfSocial` is designed to handle this variability.

It is intended that both the social profile XML file and its `ProfileStructure` class (its encapsulated binary schema) are prepared together. It should be noted that so long as the creation of both the XML and ProfileStructure class files is a manual procedure, there will be opportunities for errors. Simple errors like incorrect case sense and mis-matches between method and element names are typical. Automation of this process may be possible and may reduce or eliminate such errors.

Consequences of Error and Failure behaviour:

Superfluous fields and methods in the `ProfileStructure` class which do not have matching elements do not pose a problem as they will simply never be used (recall that methods are called based on the list of elements present in the XML file).

Elements present in the XML file which do not have a corresponding method in the `ProfileStructure` class (i.e. a missing or incorrectly named method) throw a Java `NoSuchMethodException` exception. In the implementation for this thesis, this exception is 'caught gracefully', i.e. it does not stop the further processing of the remaining elements. In a production quality implementation, occasions of this type of exception should be logged and flagged as the social profile XML file is not indexed as intended.

Differences in Operating Systems

In this experiment, a variant of the Linux-Redhat operating system (OS) is used as an example. Structure, availability and access permission of system files and other sources of system information will vary from system to system. However, for the Linux environment used here at least, the ProfileStructure class can be used to obtain information from the underlying OS. It follows that if system file reading permissions allow, the ProfileStructure class can be modified to match the different file names and structures found in other OSs.

6.9.6 Contribution

Notwithstanding the above limitations, the *encapsulated binary schema* (in the form of the `ProfileStructure` class) makes two important contributions to this theses, these are recapitulated here:

1. It allows the schema of the social profile XML file to be extended without the need to reconfigure any of the agent's abilities. This enhances the flexibility with which social profiles can be used in SGAs or any agent technology.

2. The *encapsulated binary schema* can capture real-time and definitive system information for use in social profiles; it can do this without reconfiguration of any agent abilities. As a result of this, the social profile can then become a combination of 'socially' descriptive information and technical system information, both of which may compliment each other when social profile content is used in resource allocation decisions.

Chapter 7

Conclusions

At the start of this thesis, we discussed the goal of assembling heterogeneous computing resources in order to address the many large computing challenges which exist today. In doing this, we saw that both the computing resources and potential work for them were very different. Differences in technologies, ownership, accessibility, and social and economic motivations all presented difficulties in achieving this goal. Interoperability between these different systems presents one challenge, and resource allocation another. By virtue of their distributed nature and their ability to interact with each other, software agents are suited to both of these challenges and can address some of these difficulties.

In the context of using agents in resource allocation, this thesis extends and improves an existing agent technology, Social Grid Agents (SGAs), one which was originally designed to allocate grid computing resources.

The motivation for doing this was to enable fast and easy creation of large numbers of these agents. Because the environment for which they are designed is continually changing, these agents need to be able to be configured in a flexible and dynamic way.

Two particular areas were examined. First, the ease of creation and configurability of these agents. And secondly, integration of search technology into the agents for measuring similarity between resources and work. Both of these developments aim to enhance the agent's usability and potential for use in computing resources allocation.

It is time to return to the original research questions of this thesis as posed in sub-section 1.4.1, page 7 and to consider how the work of this thesis contributes to the overall challenge of allocating resources in a disparate and heterogeneous computing landscape. Thus, this final chapter briefly recaps the main contributions of the thesis, referring back to these questions where appropriate. It also provides a discussion and conclusions, and outlines some areas of possible future work.

7.1 Scalability and Configurability of SGAs

Chapter 4, discussed how Social Grid Agents (SGAs) were originally created [107]. Later in that chapter, a mechanism was introduced which built, deployed and configured SGAs in a number of stages (see figure 4.10).

The configuration approach used in the original work [107] could be called a 'single-stage' approach, where the entire agent was coded, built and deployed with its final configuration. Using an entirely new approach to creating and configuring SGAs, this thesis showed that a system of multiple SGAs from Pierantoni's work could be recreated, achieving identical configuration and functionality to that of the original 'hardcoded' system. This result directly addresses question 1 of section 1.4.1. Two contributions flow from this positive result. Firstly, multiple SGAs can now be easily created. Second, SGAs can now be configured in a dynamic and flexible way.

Multiple agent name and location details can be specified in the *MAS Plan File* (MPF) data-files. These agents can then be quickly and automatically created and deployed. In a similar way, the abilities to be added to these agents can be set out in 'abilities blueprint files'. On creation of the resulting abilities configuration files (ACFs), these abilities are then added to each agent after they are already running. Further abilities can be added at any time. This significant enhancement to SGA technology addresses questions 2 and 3 of section 1.4.1 and ultimately enhances their usability and makes further development of SGAs feasible.

7.1.1 Creating Multiple SGAs

The mechanism presented facilities creation of large numbers of agents, creating a multiple-agent system or MAS. Each SGA created has a unique name and is automatically deployed to its 'target' host system. The MPFs, on which the creation sequence is based contain these details. They are both machine and human readable and, in the future could be created either manually (on a small scale) or as the output of a large-scale MAS planning program. The *comma separated variables* files used for this were just one available technology choice. In the future, a relational database could provide this function, bringing with it, better scalability and potential for a user interface for easier management of agent details.

The agent applications themselves, are reproduced from a relatively small fixed body of code. Applications are 'cloned', differing only in name. This alone would be of limited use without the mechanism to dynamically configure each agent in a different way after it has been created. The use of the Apache Tomcat application server as a host container for SGAs provides the mechanism by which they can be easily deployed and removed (see section 4.2).

7.1.2 Flexible Agent Configuration

In the development of the configuration stage of agent creation, several challenges around the transporting of both binary classes and instances of them were overcome. The most interesting of these is the technique used for adding an instance of an agent ability class as an ability.

Loading state in instances to SGA: In SGAs, both logic and 'remembering state' can be provided by using an instance of an object (see sub-section 4.6.1). In a 'hardcoded' configuration, achieving this is trivial (see figure 4.5) as the required class definition is already available in the JVM of the agent.

In the context of configuring a remote agent by transporting serialised binaries to it, this thesis presents a technique which can transport and load to an agent, an instance of a software object which contains pre-defined state.

We have evaluated this technique against JADE [51], a similar and established agent technology. For configuration, JADE agents provide a similar mechanism, but one which only loads a full class definitions. Examining the source code and literature of the JADE framework showed that it did not have a facility to load class instances. Further to that, email contact with the JADE development group confirmed that JADE cannot provide this functionality. They have also expressed an interest in how this was achieved and would like to see it published. In the context of agent configuration, we believe this technique, which loads an instance of a class to an agent to be a novel contribution.

Configuration Extensibility: The ACF used for transporting serialised abilities to SGAs (section 4.3.2) is both flexible and extensible. The 'includes' feature means that several files may be 'chained' together. Large and unwieldy files can be broken down for easier management and reuse of frequently used abilities is facilitated. ACFs can also be shared and reused, allowing the process of creating ACFs to become more efficient. In being able to do this, question 4 of section 1.4.1 is directly answered. Where multiple-agent systems are large, blocks of abilities common to a number of SGAs can be easily specified for addition to them. This further enhances planning and execution of an MAS creation.

URL class loading: Experiment 5 (section 6.7) showed that a remote JAR file can be used to support an ability within an SGA. Exploiting this, abilities which need JARs or libraries which are not already available within an SGA can now be created and used. The required external JARs may be accessed over the network and any number of JARs can be provided this way. JARs can be put in place and made available without any impact on agents of their host machines. The main import of this is that an already running SGA does not have to be stopped to insert the required JARs into its package. In a real-world resource allocation environment, this means that continuity of service and other interactions an agent may have undertaken need not be disturbed. This contribution addresses question 5 as set out in section 1.4.1.

Considering that an SGA starts out being a *basic* agent with only a minimal configuration, this capability further enhances the scalability, extensibility and programmability of any system created using this mechanism.

7.1.3 Discussion

The reader should bear in mind that, in the original SGA development work, consideration was not given to SGAs being created and configured using the steps described here. The mechanism proposed here is entirely a creation of this thesis and is the fruit of an examination of the architecture and available code from the original work.

The motivation for this examination was to improve agent creation and configuration, giving a better alternative to the 'single-stage' hardcoded approach. This was achieved, all the while preserving the original SGA architecture and functionality.

Developing Abilities for SGAs

Future SGA developers should be mindful of the one-to-one relationship between the *action-name* and the *ability* when these are added to the internal data-structure of the agent. This means that an ability needs to be a stand-alone binary class. If an external supporting class is unavoidable, it can be provided as an external class via a URL class loader. However, in some instances, this can be avoided by using certain coding practices.

Java Exceptions: Exception classes which are specific to an ability require an additional classes, external to the ability they support. These are common in the original SGA work [107] and are perfectly applicable to the 'single-stage' agent creation approach. To avoid external exception classes, a generic Java Exception (the super-class of all Exceptions) can be used. This, combined with informative error logging will often suffice.

Refactoring of Abilities: Writing stand-alone classes will sometimes mean that good object-oriented programming practices have to be sacrificed. As an example, refactoring code to make another 'software object' creates an 'external' class. However, avoiding refactoring can lead to less code reuse and longer sequences of code. This is a design judgment to be made by future developers.

Serialisation Rules (and breaking them)

The Java programming language includes its own mechanism for serialisation of classes. This feature has several restrictions which aim to ensure that serialising a class will not have undesirable consequences; these restrictions are in place for good reasons. One such restriction in Java serialisation is that the class to be serialised must implement the 'serializable' interface. Failure to do this will cause run-time errors when attempting to carry out the serialisation.

This restriction does not apply when using Base64 serialisation. Classes can be converted to become a Base64 string, whether or not they implement the serializable interface[1]. As an example, the `ProfileStructure` class (which was successfully serialised and encapsulated in the social profile XML file), does *not* implement the serializable interface. Of course, by-passing the restrictions of Java serialisation removes the benefits and safeguards which are intended to be derived from them. Therefore, a working knowledge of the pit-falls of using serialisation without the benefits of Java's own protections is recommended when using Base64 serialisation of binaries in an application like the one in this thesis.

Wider Uses of this Mechanism

The mechanism presented may also be used in the wider area of web service creation and orchestration. Any distributed system which consists of 'stand-alone' applications which uses a data-structure to manage binary abilities would benefit from the configuration approach used here.

7.2 Integrating Search into Distributed Agents

The integration of search technology with a system of agents shows that it can be deployed in this distributed environment. The benefit of being able to do this is that essential search 'objects', the social profiles of this thesis (as Lucene documents) can be successfully exchanged between agents. Further to this, search query objects were

[1] When using Base64 encoding, this is true of any binary object.

also exchanged. The combination of these two actions means that one SGA can send a query to another and receive one or more social profiles in return. Both of these actions were successfully demonstrated. This can all take place in the resource allocation environment which SGAs are designed to occupy and yields positive answers to questions 6 and 7 of section 1.4.1.

In addition to this, agents are now provided with a tool which can measure similarity between social profiles (section 6.5). The similarity measurement integrated and tested in SGA provides such information and answers question 8 as posed in section 1.4.1. Although this was implemented as a prototype in this thesis, it brings together all of the mechanisms previously developed and provides a quantitative measurement of similarity for computing resources and their candidate work.

7.2.1 Encapsulated Binary Schema

In section 5.5.4, we saw the problem caused by extending a social profile XML file which was structurally 'bound' to the logic of the ability which processed it within an agent. The problem was not only one of interpreting a changed XML schema; it was also one of 'imparting' the logic needed to process the additional information in the XML file. This new logic also needed to be sent to the remote agent.

Question 9 of section 1.4.1 posed this question originally, and the *encapsulated binary schema* addressed the problem successfully. A pre-programmed binary (the `ProfileStructure` class) was serialised and transported to the agent within the amended social profile XML file.

The main contribution of the *encapsulated binary schema* is that it provides flexibility and scalability for the contents of the social profile XML file. It also allows the logic and functionality which indexes the social profile contents to be specified and controlled by the social profile creator.

Another significant contribution of the *encapsulated binary schema* is its capability to harvest descriptive system information from the underlying host system of an agent. This concept was exploited in section 6.9 when additional logic was added to the `ProfileStructure` class. This enabled the final social profile (in the agent's index) to be composed of both descriptive text and locally sourced system information. This functionality enhances the descriptive capacity of social profiles when applied to computing resources. For example, where the specifications of the underlying system are important (such as Grid or Volunteer computing), providing such system information from its definitive source is prudent. This contribution addresses question 10 of section 1.4.1.

7.2.2 Similarity Measurements

The similarity measurements used in Experiment 2 (section 6.4) to distinguish between descriptions from different categories were promising. The results for all four categories were positive, although, for one category, the result was not as strong as the others. Possible reasons for this are discussed in sub-section 6.4.3.

It is reasonable to speculate that by using a more refined method of social profile creation, the capacity to distinguish descriptions of different resource allocation entities will be at least as good, if not better than seen in Experiment 2 (section 6.4). In general, search depend on the effectiveness of the descriptions, how they are indexed and the types of queries used. Improvements in this area are in the domain of 'Information Retrieval', a mature but vibrant area of investigation in its own right.

Although similarity measurements between social profiles have been determined in this work, larger numbers of profiles will be needed in order to establish their usefulness for establishing measurable differences between the entities described. The MAS architecture of chapter 4 enables such large-scale experiments in the future, and Lucene's scalability also supports this (recall section 5.4.6).

The outcome of the *More-Like-This* experiment (sub-section 6.4.4) showed that using a social profile as a basis for a query is a reasonable alternative to using designed queries. In a resource allocation scenario, where social profiles are plentiful, this may well prove to be a more suitable way to create queries for searching between agents.

7.2.3 Discussion

Other uses for social profiles: The flexibility of the Lucene Document structure allows several additional fields to be added to a social profile. This may allow features to be implemented within an agent technology which are not directly related to social and economic descriptions. An example of this is adding *longitude and latitude* coordinates for an agent where the location of an agent is important, and where this information is available in the underlying system.

Given the capacity of a Lucene document to store Boolean or logical type data, the social profile format may also be used as a *record* in a directory service. An agent providing this service could aim to collect a large number of social profiles and could also implement a directory protocol, be it *hierarchical*, *relational* or other.

Third-Party information for social profiles: Chapter 5 discussed how descriptions of sellers on an auction website are generated by members of the public. Such generation of information is termed 'Social Production' by Benkler [52]. In a real-world use of social profiles, it may be possible for the third-party content to be generated by 'social production'. Social production in a system of SGAs could take the form of 'feedback' between agents after transactions and is one practical way third-party information for social profiles could be generated.

7.3 Final Thoughts

And so we are at the end, this section concludes the chapter and indeed this thesis. Here, the learning phase, exploration of new ideas and problem solving finally culminates in one written document.

Something from you youth reminds me that taking things apart is usually easier than putting them back together again; it turns out that working with software applications is no different.

One thing learned; problems and obstacles are often only challenges that need more thought and perhaps a different approach. And so it was with most of the challenges that came along while progressing this thesis. The enjoyable part was finding solutions to each one, then collectively putting them together and getting things to work.

Appendix A

Sample gLite Commands

Figure 1.1 shows some sample gLite user interface commands with some explanatory comments.

```
// Create a proxy file to allow access
[user@ui] $ voms-proxy-init

// Check if there is a matching CE for a JDL file
[user@ui] $ glite-wms-job-list-match jobDesc.jdl

// Run the job for this JDL file, collect the job number
// in the file job.id
[user@ui] $ glite-wms-job-submit -o job.id jobDesc.jdl

// Use the file job.id to check the status of a that job
[user@ui] $ glite-wms-job-status -i job.id

// Use the file job.id to get logging information of a that job
[user@ui] $ glite-wms-job-logging-info -i job.id

// Get the output (result) of job to the specified folder
[user@ui] $ glite-wms-job-output --dir ./myFolder -i job.id
```

Figure 1.1: Sample gLite commands

Appendix B

Class Loading Issues Specific to Tomcat

B.1 JVM and Tomcat Class Loader Architecture

The Tomcat container (being a Java implementation) builds on the hierarchy of class loaders found in the JVM and in some respects isolates the class loaders lower down the hierarchy in the container from those above it in the JVM[1]. It uses a number of additional class loaders to load for example, the classes used to generate the containers' web pages and other programs in the container which are common and available to all web applications. At the bottom of the hierarchy, Tomcat allocates a class loader to each new web application (each SGA in this case) that starts in the container, and the classes included in the WAR file for an application are loaded in to this. This is called the 'web application class loader' or `WebappClassLoader` and its use means that the classes included within a WAR file for a given application are isolated from those of other applications. However, this class loader does not follow the same delegation model of others above it in the hierarchy, instead it tries to load classes found in the application's own directory first before delegating the request to its parent.

[1] This class loader isolation is done by changing the paths which are used to load various classes which would be expected to be found in the JVM.

B.1.1 Loading De-Serialized Classes in Tomcat

When an object is to be deserialised and used, a user-defined class loader is created for this. In a standard JVM (i.e. outside of a Tomcat container), when the default object input stream (i.e. an instance in Java of `ObjectInputStream`) is used to deserialise an object, the JVM uses a (somewhat complicated) mechanism to determine the 'latest user-defined class loader', and this is used to load the class. Investigations for this thesis demonstrated that this works as expected when using the standard JVM and an IDE.

When deserialising an object in a web application (using its WebAppClassLoader), using the standard ObjectInputStream fails with a ClassNotFoundException. This is because the class loader used by the ObjectInputStream does not have visibility on classes loaded in the WebAppClassLoader. What causes this inconsistency is not relevant to this work. However, prototype work in this thesis has demonstrated that this problem exists and that it is reproducible.

A Solution Specific to Tomcat Technology

A Tomcat specific solution allows the correct ClassLoader to be passed to a new class which extends ObjectInputStream and thereby forces the ObjectInputStream to use the WebappClassLoader. The `CustomObjectInputStream` class takes as arguments, the `InputStream` to be converted to an object and an instance of the current `ClassLoader` of the application (in which it can expect to find a definition of that object).

The CustomObjectInputStream class (which is quite short) has methods that simply override the `resolveClass` method found in the super class of the standard ObjectInputStream. Both the byte array to be deserialised and this class loader are passed to the constructor of the Customised ObjectInputStream. The classloader passed is the same one into which the class definition was loaded and therefore overcomes the class visibility problem found in this environment.

Alternative Solution non-Specific to Tomcat Technology

An alternative solution (not specific to Tomcat) was devised specifically for this problem. It allows for instances of classes which contain state to be added to the data-structure within the agent. A separate class needs to be written and compiled for each ability that needs to have state. Although this is perhaps an unconventional approach to writing Java code, the mechanism used in chapter 4 could be adapted to edit (using the Linux command line program `sed`) the small number of classes required. In the development domain, an implementation of such a class is created and serialized. Then, the serialised class definition is added to the ACF for transport. Agent-side, the class definition is deserialised and loaded into the class loader of that web application. An instance of such a class is instantiated agent-side (having the member variables initialised to the required values). As all this is done by the same class loader within the agent, visibility of the class definition is not an issue. When this instance is loaded to the data-structure, it can be reused several times. If variables are changed, these state changes are 'remembered' from message to message. This replicates the behaviour seen when instances of abilities are added to the data-structure when the agent starts (as happened in the 2008 work [107]). This method was explored during prototype work for experiments. It is, however, somewhat more convoluted and laborious when compared to the alternative 'Tomcat specific' solution.

B.1.2 Usage of Both Solutions

The second solution proposed is not specific to using a Tomcat container, and allows the ObjectInputStream available in the JVM to be used. In the implementation for this thesis, both were tested and confirmed working in both a Tomcat container and in a standard development JVM. For the experiments of chapter 6, the CustomObjectInputStream is used as it reduces the workload and complexity required (and opportunities for errors) to prepare the necessary ability configuration files.

Appendix C

Inventory List of Classes Required for All SGAs in Experiment

This is a list of all the classes required for configuration of the eight SGAs used in the experiment described in section 6.3 on page 117. Each class name item in the list is preceded by the action-name associated with it at the SGA configuration stage.

```
AddToWallet - WalletSP
HowMuchInWallet - WalletSP
RemoveCreditInWallet - WalletSP
Accept - DummyAcceptanceSP
AddPolicyToJdl - TmpPolicySP
AddTokens - TokenManagerSP
Availability - JSAvSyncProcessor
DirectControl - DirectControlledProvider
EnforceAllocation - DummyAllocationSP
EnforcePolicies - PolicySyncProcessor
Execution - DirectControlledProvider
Execution - DummyGLiteJobStatusSP
Execution - JSAsyncProcessor
Execution - PubProvider
Execution - SimplePurchaseProvider
Execution - DummyGLiteJobStatusSP
Execution - JSAsyncProcessor
GetSelfId - DummySelfIdentitySP
GLiteJobListMatch - DummyGLiteJobListMatchPubSP
GLiteJobListMatch - DummyGLiteJobListMatchSP
GLiteJobManagement - DummyGLiteJobManagerSP
GLiteJobSubmission - DummyGLiteJobSubmissionPubSP
GLiteJobSubmission - DummyGLiteJobSubmissionSP
HowManyTokens - TokenManagerSP
MapPolicies - DummyPolicyMapperSP
PurchaseExecution - GT4ExecutionSP
Purchase - JobPurchaseAsyncProcessor
RemoveTokens - TokenManagerSP
SendMessage - DummyCommunicationSP
WriteStatus - StatusWriterSP
```

Appendix D

Abridged List of Classes and JAR files in a Basic SGA WAR Package

```
AgentServiceImpl.class
AgentService.class
AgentServiceInitializer.class
ResourceManager.class
AsyncProcessor.class
SyncProcessor.class
AsyncProcessorException.class
SyncProcessorException.class
GenericDescriptionUtilities.class
MessageDescriptionUtilities.class
DescriptionParser.class
TestActionTest.class
ActionValues.class
ObjectValues.class
GLiteWmsActionValues.class
StatusValues.class
SimpleIdentityParser.class
SemConInputParser.class
ObjectParserException.class
JdlObjectParserException.class
JdlObjectParser.class
StatusParser.class
ObjectValues.class
ObjectParser.class
ObjectTags.class
JobIdObjectParser.class
SemConInputValueParser.class
TestActionParser.class
SemConInputTags.class
ActionStatusTags.class
MessageDescriptionTags.class
MessageTags.class
ObjectTags.class
ActionTags.class
StatusTags.class
JdlTags.class
```

Appendix E

Log Outputs for Wallet Ability Configuration Details

Sample of the logged output from the configuration of the Wallet ability in the 2008 work of Pierantoni [107].

```
[ [ Requirements = ((other.Content.Description.Type=="Action")&&
(other.Content.Description.Name=="AddToWallet"));
Rank = 1 ] --> common.core.providers.sga.WalletSP@72f2a824 ]

[ [ Requirements = ((other.Content.Description.Type=="Action")&&
(other.Content.Description.Name=="HowMuchInWallet"));
Rank = 1 ] --> common.core.providers.sga.WalletSP@72f2a824 ]

[ [ Requirements = ((other.Content.Description.Type=="Action")&&
(other.Content.Description.Name=="RemoveCreditInWallet"));
Rank = 1 ] --> common.core.providers.sga.WalletSP@72f2a824 ]
```

Sample of the logged output from the configuration of the Wallet ability in the agent S-1 in the experiment described in section 6.3.

```
[ [ Requirements = ((other.Content.Description.Type=="Action")&&
(other.Content.Description.Name=="AddToWallet"));
Rank = 1 ] --> common.core.providers.sga.WalletSP@315a6eb9 ]

[ [ Requirements = ((other.Content.Description.Type=="Action")&&
(other.Content.Description.Name=="HowMuchInWallet"));
Rank = 1 ] --> common.core.providers.sga.WalletSP@315a6eb9 ]

[ [ Requirements = ((other.Content.Description.Type=="Action")&&
(other.Content.Description.Name=="RemoveCreditInWallet"));
Rank = 1 ] --> common.core.providers.sga.WalletSP@315a6eb9 ]
```

Appendix F

Log Outputs for JSAsyncProcessor Ability Configuration Details

Sample of the logged output from the configuration of the JSAsyncProcessor ability in the 2008 work of Pierantoni [107].

```
[ [ Requirements = ((((((other.Content.Description.Type=="Action")&&(other.Content.Description.Name=="Execution"))&&
(other.Content.Input.Description.Type=="Action"))&&(other.Content.Input.Description.Name=="Submission"))&&
(other.Content.Input.Input.Description.Type=="Object"))&&(other.Content.Input.Input.Description.Name=="JDL"));
Rank = 1 ] --> class common.core.processors.JSAsyncProcessor ]
```

Sample of the logged output from the configuration of the JSAsyncProcessor ability in the agent S-1 in the experiment described in section 6.3.

```
[ [ Requirements = ((((((other.Content.Description.Type=="Action")&&(other.Content.Description.Name=="Execution"))&&
(other.Content.Input.Description.Type=="Action"))&&(other.Content.Input.Description.Name=="Submission"))&&
(other.Content.Input.Input.Description.Type=="Object"))&&(other.Content.Input.Input.Description.Name=="JDL"));
Rank = 1 ] --> class common.core.processors.JSAsyncProcessor ]
```

Appendix G

Sample Outputs of Trial Messages to SGA Experiment

Sample from the logged output of agent S-1 while process messages during the experiment described in section 6.3. Note here how the credit in the wallet changes as each message is processed. In the implementation, the agent S-1 is named `lcg2wmssga1`.

```
Agent confirmed reachable at...
http://10.61.0.7:8080/lcg2wmssga3-1.0/AgentService

["lcg2wmssga1" - 18:49:58] - SimplePurchase is available.
There is 88 in the wallet of lcg2wmssga1
Removing 10 from 88 in wallet of lcg2wmssga1
Now credit is 78 in lcg2wmssga1
Now in CxfCommunicationSP.class in lcg2wmssga1

["lcg2wmssga1" - 18:50:04] - Submission successful !
Adding 10 to 78 in wallet of lcg2wmssga1
Current Credit after addition is: 88
Now credit is 88 in lcg2wmssga1
"PurchaseExecution" ---> "Billing"
Before performing action...
After performing action...
"Billing" ---> null

Agent lcg2wmssga1 has received a message

There is 88 in the wallet of lcg2wmssga1

["lcg2wmssga1" - 18:50:18] - Submission successful !
Adding 10 to 88 in wallet of lcg2wmssga1
Current Credit after addition is: 98
Now credit is 98 in lcg2wmssga1
"PurchaseExecution" ---> "Billing"
Before performing action...
After performing action...
"Billing" ---> null
```

Appendix H

Outcomes of Term Query Searches

Category: 1 - Commercial Companies and Corporations		
Rank	Similarity Score	Entity Name
1	0.129878	General Electric, USA
2	0.128846	Vodafone
3	0.108973	Ryanair
4	0.066199	US Govt Dept of Commerce
5	0.051682	Virgin Atlantic

Table 8.1: Extract of Results for Term Query Search, Category 1

Category: 2 - Universities, Academic Institutions		
Rank	Similarity Score	Entity Name
1	0.189299	University of California Berkeley US
2	0.181311	Princeton University US
3	0.162844	Trinity College Dublin
4	0.135752	The Carnegie Foundation
5	0.060671	Imperial College London UK

Table 8.2: Extract of Results for Term Query Search, Category 2

Category: 3 - Charities and Benevolent Foundations		
Rank	Similarity Score	Entity Name
1	0 .131318	Doctors Without Borders
2	0 .065304	Bill And Melinda Gates Foundation
3	0 .020517	Imperial College London UK
4	0 .014116	Irish Govt Dept of Agric., Food and Marine
5	0 .013708	Virgin Atlantic

Table 8.3: Extract of Results for Term Query Search, Category 3

Category: 4 - Government or Similar Departments and Institutions		
Rank	Similarity Score	Entity Name
1	0.11839	US Govt Dept of Commerce
1	0.073745	The Global Fund
1	0.072072	Irish Govt Dept of Education and Skills
1	0.06361	Irish Govt Dept of Agric., Food and Marine
1	0.054412	Vodafone

Table 8.4: Extract of Results for Term Query Search, Category 4

Appendix I

Output from Includes in XML Abilities Files

```
=================

Agent lcg2wmssga1 has received a message

< ...abridged... >

Agent lcg2wmssga1 has received a message

Now in FetchProfAbilitiesSP.class in lcg2wmssga1

< ...abridged... >

No of Suitable Actions found in
lcg2wmssga1_abilities_a.xml is:   5

< ...abridged... >

No of Suitable URLs found in lcg2wmssga1_abilities_a.xml is:  1

From lcg2wmssga1_abilities_a.xml, about to do recursive call on:
http://shiraz.cs.tcd.ie/profile/lcg2wmssga1_abilities_b.xml
```

```
No of Suitable Actions found in

lcg2wmssga1_abilities_b.xml is:   0


No of Suitable URLs found in lcg2wmssga1_abilities_b.xml is:  3


From lcg2wmssga1_abilities_b.xml, about to do recursive call on:

http://shiraz.cs.tcd.ie/profile/lcg2wmssga1_abilities_c.xml


No of Suitable Actions found in

lcg2wmssga1_abilities_c.xml is:   1


< ...abridged... >


No of Suitable URLs found in lcg2wmssga1_abilities_c.xml is:  0


From lcg2wmssga1_abilities_b.xml, about to do recursive call on:

http://shiraz.cs.tcd.ie/profile/lcg2wmssga1_abilities_d.xml


No of Suitable Actions found in

lcg2wmssga1_abilities_d.xml is:   2


< ...abridged... >


No of Suitable URLs found in lcg2wmssga1_abilities_e.xml is:  0


< ...abridged... >
```

Appendix J

Output from URL Class Loader HTTP Requests

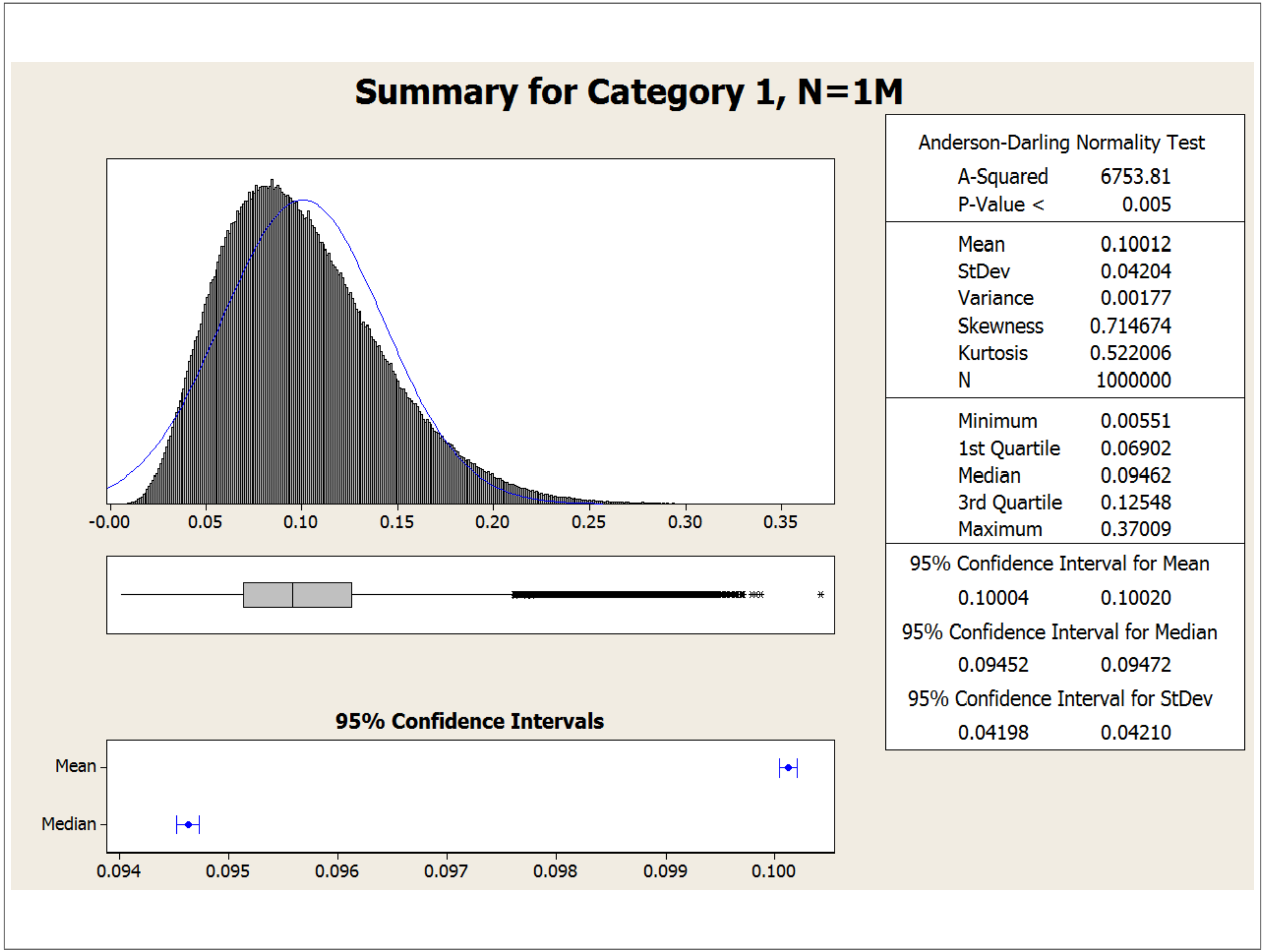
Figure 10.1 on page 211 is a sample from the HTTP log output from Experiment 5, section 6.7 on page 167.

```
10.61.0.5 - - [09/Apr/2013:08:41:47 +0100]
"GET /profile/workowner_001_abilities.xml
HTTP/1.1" 200 175030 "-" "Java/1.6.0_20"

10.61.0.5 - - [09/Apr/2013:08:41:52 +0100]
"GET /profile/Africahome_social.xml
HTTP/1.1" 200 21121 "-" "Java/1.6.0_20"

10.61.0.5 - - [09/Apr/2013:08:43:40 +0100]
"GET /classes/commons-math-2.2.jar
HTTP/1.1" 404 307 "-" "Java/1.6.0_20"

10.61.0.5 - - [09/Apr/2013:08:44:48 +0100]
"GET /classes/commons-math-2.2.jar
HTTP/1.1" 200 988514 "-" "Java/1.6.0_20"

10.61.0.5 - - [09/Apr/2013:08:44:54 +0100]
"GET /classes/commons-math-2.2.jar
HTTP/1.1" 200 988514 "-" "Java/1.6.0_20"
```

Figure 10.1: Output from URL Class Loader HTTP Requests

Appendix K

Statistical Charts for Experiment 2 in Chapter 6



Summary for Category 1, N=1M

Anderson-Darling Normality Test	
A-Squared	6753.81
P-Value <	0.005
Mean	0.10012
StDev	0.04204
Variance	0.00177
Skewness	0.714674
Kurtosis	0.522006
N	1000000
Minimum	0.00551
1st Quartile	0.06902
Median	0.09462
3rd Quartile	0.12548
Maximum	0.37009
95% Confidence Interval for Mean	
0.10004	0.10020
95% Confidence Interval for Median	
0.09452	0.09472
95% Confidence Interval for StDev	
0.04198	0.04210

Figure 11.1: Graphical summary of data from randomly generated queries for category 1, section 6.4.

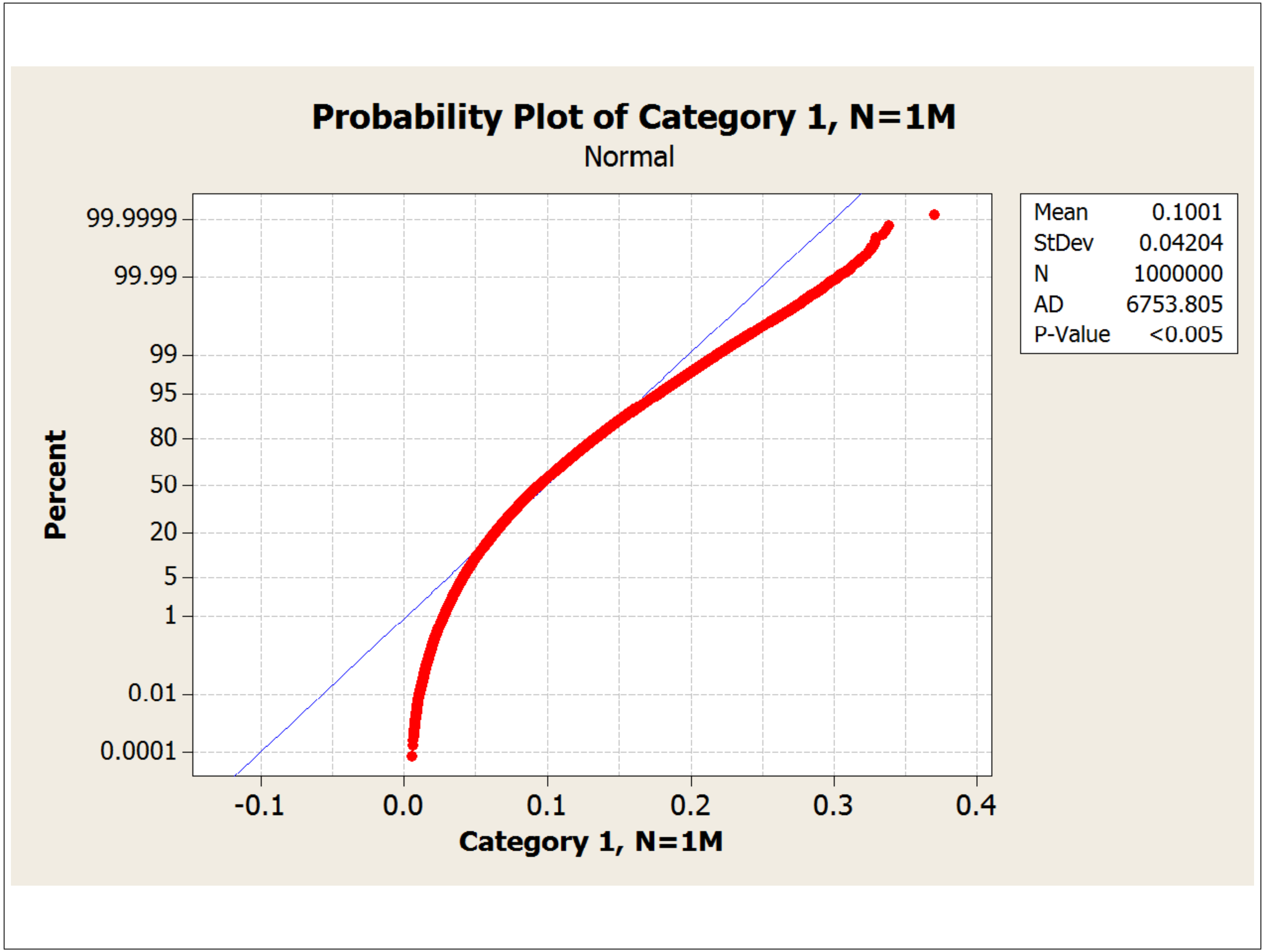


Figure 11.2: Plot from Anderson-Darling Normality test on data from randomly generated queries for category 1, section 6.4.

Appendix L

Issues with SGAs in GT4 Technology

L.1 Globus Implementation and SGAs

In the original SGA [107], each agent was individually developed and built to produce a Globus archive file (GAR) with a fixed number of Service-providers and Processors statically configured for the SGA. The GAR file was then transferred to the GT4 container's host computer and deployed there. Upon restarting the GT4 container, the SGA was available for use. Further changes to the SGA configuration required a repeat of this cycle.

In addition to this, building of a GAR file for a GT4 service did not support inclusion of additional SGA dependency JAR files in the GAR. This meant that deployment was not an 'atomic action' as multiple files had to be transferred to the web service container. Furthermore, these dependency JARs were also shared by all SGAs running in that container. This had the potential to cause problems if later SGAs needed modified dependencies.

Experimental work was carried out to manually add dependencies to GAR files using the command line tool `jar`. GAR files modified in this way were successfully deployed, but this step added complexity to the build process. The use of a shared file area for dependencies in the GT4 container meant that 'undeploying' one service may remove a dependency JAR needed by another SGA which may remain running. It should be pointed out that this was not a modification of GAR files intended by Globus. Instead, it was an attempt to overcome the necessity to transfer multiple files to a container for this thesis. GT5.0 when released did not support web services, instead focusing on a different area of grid job submission.

Appendix M

Discussion on CXF and Tomcat Frameworks

M.1 CXF — Tomcat and Globus Toolkit 4 Comparison

There are two significant differences between developing and deploying web services in Globus Toolkit and in the CXF-Tomcat architecture which are relevant to this thesis.

The WAR file created by building a 'basic' Social Grid Agent using CXF web services is just under 10 megabytes and contains just over 220 files. This consists of approximately five XML and MANIFEST files, 178 binary java classes which make up the web service (the agent) itself, and approximately 30 JAR files which are needed by the CXF web service. The remaining JAR files (about 5) are dependencies of the classes of the agent.

These dependencies are specified during development and can be added using the build program Maven at that time. Maven keeps track of the required dependencies in the Project Object Model file (i.e. the `pom.xml` found in each Maven project). The list of dependencies in this file is the basis for the dependencies included in the WAR file. As a result, for a given service, the corresponding WAR file will be many times bigger than the equivalent Globus archive file (or GAR).

When deployed in a Tomcat container, the contents of the WAR file are extracted to a directory specific to each service. This ensures that files associated with individual services are stored separately on the file-system of the container machine (it is also possible to configure Tomcat to run a service without expanding the WAR file).

The Maven build process yields a single WAR file which contains all the required files for an agent. This means deploying it is an 'atomic' action, only requiring the file to be copied to the hosting machine.

M.1.1 Archive File Structure and Storage

There is a notable difference between the structure of the WAR file and the Globus 4.2.1 GAR file. In GT4 deployment, only a small number of agent implementation classes are included in the GAR file, the remaining required files have to be made available to the agent in the GT4 container by manually packaging them in a separate JAR file and storing this with other JAR files on the GT4 container.

Along with creating the requirement of transferring two files to the container for agent deployment, this also meant that where a number of agents are deployed in the same container, all agents use classes from the same JAR file for their dependencies.

M.1.2 File Expansion on Host Machine

JAR files (external to the GAR) copied to the GT4 container must be stored in a location which is common to all services deployed in the GT4 container (SGAs and other services). Only the small number of files from within the GAR file are actually stored in an area designated specifically for that agent. This means that 'sandboxing' of binary files specific to a particular agent is more difficult.

M.1.3 Disk Usage on Host Machine

When compared to using GT4 technology, using CXF and tomcat requires marginally higher disk storage on the host machine. The benefits of 'atomic' (single file) deployment, such as isolation of dependencies from other agents outweighs this drawback.

Appendix N

Sample ClassAd Expressions

The following is an example of two simple *ClassAds* which describe two actors owning resources and work.

```
Expression1 :  [
    Type = ResourceOwner;
    CPUspd = 1.7E9;
    Mem = 2000000.0;
    Beneficiary = Actor N;
    Cost = 10;
    ObjectiveList = {
        Life Science,
        Education
    };
    Requirements = other.Type == WorkOwner
    && other.PriceExp >= 10
    && member(Life Science,other.ObjectiveList)
    && member(Weapons,other.NotList);
    Rank = other.PriceExp
]

Expression2 :  [
    Type = WorkOwner;
    Beneficiary = Actor N;
    PriceExp = 10;
    NotList = {
        Nuclear Research,
        Weapons
    };
    ObjectiveList = {
        Humanitarian,
        Life Science
    };
    Requirements = other.Type == ResourceOwner
    && other.Cost <= 10
    && Beneficiary == Actor N;
    Rank = other.Cost
]
```

Notes: The *Rank* output is available in an array structure and can be retrieved for example as follows;

```
   int[] rank = ca.match(Expression1, Expression2);
   System.out.println("Rank of expression 1 : " + rank[0]);
...
Rank of expression 1 : 10
```

Bibliography

[1] http://www.egi.eu/. The European Grid Infrastructure.

[2] http://www.amazon.com/ec2/. Amazon Elastic Compute Cloud (EC2) Retrieved Mar 2013.

[3] http://aws.amazon.com/s3/. Amazon Simple Storage Service Retrieved Mar 2013.

[4] http://www.rackcpace.com/. Cloud Service Provider, Retrieved March 2013.

[5] http://www.hpcloud.com/. Hewlett Packard Cloud Services, Retrieved March 2013.

[6] http://boinc.umiacs.umd.edu/about.php. Introduction to The Lattice Project. Accesssed March 2013.

[7] http://www.cs.wisc.edu/condor/classad/. The ClassAd Language Reference Manual, Solomon M. Retrieved January 2013.

[8] http://setiathome.berkeley.edu/. Search for Extraterrestrial Intelligence. Accesed February 2013.

[9] http://www.teragrid.org/. TeraGrid (US), Retrieved November 2012.

[10] http://www.opensciencegrid.org/. Open Science Grid, (US) OSG, Retrieved December 2012.

[11] http://glite.web.cern.ch/glite/. gLite middleware for Grid computing, Retrieved February 2013.

[12] http://portal.p-grade.hu/. P-Grade Portal, Retrieved Nov 2009.

[13] http://www.simbiosys.ca/. Simulated Biomolecular Systems Inc, Toronto, Canada, Retrieved March 2013.

[14] http://code.google.com/appengine/docs/whatisgoogleappengine.html. Google App Engine Retrieved January 2013.

[15] http://www.google.com/enterprise/apps/business/. Google Apps for business overview, Retrieved March 2013.

[16] http://office.microsoft.com/en-ie/business/what-is-office-365-for-business-FX102997580.aspx. Microsoft Office 365 for business, Retrieved April 2013.

[17] http://www.opennaas.org/. Retrieved May 2012.

[18] http://www.mersenne.org. Accesssed March 2013.

[19] http://boinc.berkeley.edu/. Accesssed January 2013.

[20] http://www.worldcommunitygrid.org. Accesssed March 2013.

[21] http://www.ibm.com. IBM Accesed June 2012.

[22] http://www.edges-grid.eu/. EDGeS - Enabling Desktop Grids for e-Science, Retrieved September 2012.

[23] http:www.globus.org. The Globus Alliance, Globus Toolkit, Retrieved November 2012.

[24] http://home.web.cern.ch/. European Organization for Nuclear Research, Retrieved March 2013.

[25] http://www.grid.ie/. Grid Ireland, The National Computational Grid for Ireland.

[26] http://www.freerainbowtables.com/. The Free Rainbow Tables Project, Retrieved March 2013.

[27] http://www.snic.vr.se/about-snic/documents/snac-strategic-documents/SNAC-policy.pdf. Swedish National Allocations Committee, published by SweGRID, the Swedish Grid Initiative, Retrieved March 2013.

[28] http://eu-datagrid.web.cern.ch/eu-datagrid/. The EU DataGrid Project, Retrieved Oct 2012.

[29] http://public.eu-egee.org/. European Grids for Science, Retrieved Nov 2011.

[30] http://www.opennebula.org/. Open Source Toolkit for Data Center Virtualization, Retrieved March 2012.

[31] http://www.eucalyptus.com/. Eucalyptus, open-source, cloud software, Retrieved March 2013.

[32] https://www.scss.tcd.ie/disciplines/computer_systems/cag/. Computer Architecture and Grid Research Group, Trinity College Dublin (see also http://www.grid.ie). Accessed January 2013.

[33] http://cxf.apache.org/. Open source web services framework, Retrieved July 2012.

[34] http://www.fipa.org/. Foundation for Intelligent Physical Agents, Retrieved Nov 2012.

[35] http://www.eclipse.org/. The Eclipse Integrated Development Environment, Retrieved July 2012.

[36] http://www.stratuslab.eu/. The StratusLab Project, Virtualization and Cloud Technologies for Grid Infrastructures. Accessed May 2012.

[37] http://lucene.apache.org/. The Apache Lucene project, Retrieved March 2012.

[38] http://nutch.apache.org. Apache Nutch project page, Retrieved March 2013.

[39] http://twitter.com/about/. Twitter Micro-blog Network website.

[40] http://www.ebayinc.com/, http://www.ebay.com/. eBay, Consumer-to-consumer and business-to-consumer online auction website. Accessed March 2013.

[41] http://www.britannica.com. Encyclopedia Britannica Online Edition, Retrieved March 2013.

[42] http://tika.apache.org. Apache Tika - digital content analysis tool, Retrieved Aug 2012.

[43] Oxford english dictionary - second edition 1989. http://www.oed.com/. Accessed March 2013.

[44] G. Aad. The ATLAS experiment at the CERN large hadron collider. *Journal of Instrumentation*, 3(08):S08003, August 2008.

[45] David Anderson. BOINC: A system for Public-Resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, November 2004.

[46] David Anderson, Carl Christensen, and Bruce Allen. Designing a runtime system for volunteer computing. In *Supercomputing '06, The International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2006.

[47] David P. Anderson and Kevin Reed. Celebrating diversity in volunteer computing. In *System Sciences, 2009. HICSS '09. 42nd Hawaii*, January 2009.

[48] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, February 2009.

[49] R. Barbera, A. Falzone, V. Ardizzone, and D. Scardaci. The GENIUS grid portal: Its architecture, improvements of features, and new implementations about authentication and authorization. In *WETICE '07: Proceedings of the*

16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, pages 279–283, Washington, DC, USA, 2007. IEEE Computer Society.

[50] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE: A software framework for developing multi-agent applications, lessons learned. *Information and Software Technology*, 50(1-2):10–21, January 2008.

[51] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with JADE intelligent agents. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII Agent Theories Architectures and Languages*, volume 1986 of *LNCS*, chapter 7, pages 42–47. Springer, Berlin, Heidelberg, 2001.

[52] Yochai Benkler. *The wealth of networks : how social production transforms markets and freedom*. Yale University Press, October 2006.

[53] Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(3):349–355, September 1988.

[54] V. Breton, N. Jacq, and M. Hofmann. Grid added value to address malaria. In *Sixth IEEE International Symposium on Cluster Computing and the Grid*, volume 2, page 40. IEEE, May 2006.

[55] S. Burke, S. Andreozzi, and L. Field. Experiences with the GLUE information schema in the LCG/EGEE production grid. *Journal of Physics: Conference Series*, 119(6):062019+, July 2008.

[56] Tom Burton-West. Practical relevance ranking for 10 million books. Technical report, copyright cG2012 remains with the author, September 2012. The unreviewed pre-proceedings are collections of work submitted before the December workshops. They are not peer reviewed, are not quality controlled, and contain known errors in content and editing. The proceedings, published after the Workshop, is the authoritative reference for the work done at INEX. (p. 114).

[57] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-Time search at twitter. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1360–1369. IEEE, April 2012.

[58] Franck Cappello and Samir Djilali. Computing on large-scale distributed systems: Xtrem web architecture, programming models, security, tests and convergence with grid. *Future Gener. Comput. Syst.*, 21(3):417–437, March 2005.

[59] B. Chaib-draa and F. Dignum. Trends in agent communication language. *Computational Intelligence*, pages 89–101, May 2002.

[60] Tze-Chiang Chen. Overcoming research challenges for CMOS scaling: industry directions. In *Solid-State and Integrated Circuit Technology, 2006. ICSICT '06. 8th International Conference on*, pages 4–7. IEEE, October 2006.

[61] A. Daouadji, K. K. Nguyen, M. Lemay, and M. Cheriet. Ontology-Based resource description and discovery framework for low carbon grid networks. pages 477–482, October 2010.

[62] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. In *Journal of the American Society for Information Science*, volume 41, pages 391–407, 1990.

[63] Ian Dickinson. Agent standards. Technical report, Foundation for Intelligent Physical Agents, 1994.

[64] Joshua M. Epstein and Robert L. Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. Complex adaptive systems. The MIT Press, 1996.

[65] Z. Farkas and P. Kacsuk. P-GRADE portal: A generic workflow system to support user communities. *Future Generation Computer Systems*, 27(5):454–465, May 2011.

[66] G. Fedak, C. Germain, V. Neri, and F. Cappello. XtremWeb: a generic global computing system. *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 582–587, 2001.

[67] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an agent communication language. In *CIKM '94: Proceedings of the third international conference on Information and knowledge management*, pages 456–463, New York, NY, USA, 1994. ACM.

[68] Ian Foster. What is the grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[69] Ian Foster and Adriana Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *In 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 118–128, 2003.

[70] Ian Foster, Nicholas R. Jennings, and Carl Kesselman. Brain meets brawn: Why grid and agents need each other. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 8–15, Washington, DC, USA, 2004. IEEE Computer Society.

[71] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Yao Zhang, and V. Volkov. Parallel computing experiences with CUDA. *Micro, IEEE*, 28(4):13–27, July 2008.

[72] Michael Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. The Belief-Desire-intention model of agency intelligent agents v: Agents theories, architectures, and languages. In Jörg P. Müller, Anand S. Rao, and Munindar P. Singh, editors, *Intelligent Agents V: Agents Theories, Architectures, and Languages*, volume 1555 of *Lecture Notes in Computer Science*, chapter 1, pages 1–10. Springer Berlin / Heidelberg, Berlin, Heidelberg, May 1999.

[73] Vladimir Gorodetsky, Oleg Karsaev, Vladimir Samoylov, and Sergey Serebryakov. P2P agent platform: Implementation and testing. In Samuel Joseph, Zoran Despotovic, Gianluca Moro, and Sonia Bergamaschi, editors, *Agents and Peer-to-Peer Computing*, volume 5319 of *LNCS*, chapter 4, pages 41–54. Springer, Berlin, Heidelberg, 2010.

[74] Yanan Hao, Yanchun Zhang, and Jinli Cao. Web services discovery and rank: An information retrieval approach. *Future Generation Computer Systems*, 26(8):1053–1062, October 2010.

[75] Zellig S. Harris. Papers in structural and transformational linguistics. formal linguistics series, volume 1., 1970.

[76] Carlos A. Iglesias, Mercedes Garijo, and José C. González. A survey of Agent-Oriented methodologies. In *Intelligent Agents V. Agent Theories, Architectures, and Languages: 5th International Workshop, ATAL'98, Paris, France, July 1998. Proceedings*, Lecture Notes in Computer Science, page 630. Springerlink, 2000.

[77] N. Jacq, V. Breton, H. Chen, L. Ho, M. Hofmann, V. Kasam, H. Lee, Y. Legre, S. Lin, and A. Maas. Virtual screening on large scale grids. *Parallel Computing*, 33(4-5):289–301, May 2007.

[78] N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, M. J. Wooldridge, and C. Sierra. Automated negotiation: Prospects, methods and challenges. *Group Decision and Negotiation*, 10(2):199–215–215, March 2001.

[79] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, April 2001.

[80] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, March 1998.

[81] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), October 2006.

[82] P. Kacsuk, Z. Farkas, and G. Fedak. Towards making BOINC and EGEE interoperable. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 478–484, Washington, DC, USA, 2008. IEEE Computer Society.

[83] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. May 2011.

[84] R. Khare, D. Cutting, K. Sitaker, and A. Rifkin. Nutch: A flexible and scalable Open-Source web search engine. In *CommerceNet*, CN-TR-04-04, November 2005.

[85] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Softw: Pract. Exper.*, 32(2):135–164, February 2002.

[86] Yannis Labrou. Standardizing agent communication. In Michael Luck, Vladimir Marik, Olga Stepankova, and Robert Trappl, editors, *Multi-Agent Systems and Applications*, volume 2086 of *Lecture Notes in Computer Science*, pages 74–97–97. Springer Berlin / Heidelberg, 2006.

[87] Peter Lavin and Brian Coghlan. Work allocations governed by social profiles for large scale heterogeneous infrastructure. In *Information Society (i-Society), 2011 International Conference on*, pages 276–278. IEEE, June 2011.

[88] Peter Lavin and Brian Coghlan. Dynamic proliferation of agents in a Multiple-Agent system. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 189–197. IEEE, February 2013.

[89] Peter Lavin, Eamonn Kenny, and Brian Coghlan. Computing resource and work allocations using social profiles. *Computer Science Journal*, 14(2):273–293, January 2013.

[90] Peter Leong, Chunyan Miao, and Bu S. Lee. Agent oriented software engineering for grid computing. *Cluster Computing and the Grid, IEEE International Symposium on*, 2:2+, 2006.

[91] Gareth Lewis, Gergely Sipos, Florian Urmetzer, Vassil Alexandrov, and Peter Kacsuk. The collaborative P-GRADE grid portal computational science ICCS

2005. volume 3516 of *Lecture Notes in Computer Science*, chapter 51, pages 64–107. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[92] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[93] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, June 1988.

[94] Charles Loomis, Mohammed Airaj, Marc-Elian Bégin, Evangelos Floros, Stuart Kenny, and David O'Callaghan. *StratusLab Cloud Distribution*, pages 260–282. Cambridge Scholars, 12 Back Chapman St, Newcastle upon Tyne, UK, January 2012.

[95] M. Luck, P. McBurney, and C. Preist. Agent technology: Enabling next generation computing (a roadmap for agent based computing). *Agentlink*, 2003.

[96] N. Maudet and B. Chaib Draa. Commitment-based and dialogue-game-based protocols: new trends in agent communication languages. *The Knowledge Engineering Review*, 17(02):157–179, 2002.

[97] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action.* Manning Publications, 2 edition, May 2010.

[98] John C. Mitchell. *Concepts in Programming Languages.* Cambridge Univeristy Press, 2004.

[99] Raffaele Montella, Giulio Giunta, and Angelo Riccio. An integrated ClassAd-latent semantic indexing matchmaking algorithm for globus toolkit based computing grids. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 4967 of *LNCS*, chapter 100, pages 942–950. Springer, Berlin, Heidelberg, 2008.

[100] D. S. Myers, A. L. Bazinet, and P. Cummings. *Expanding the Reach of Grid Computing: Combining Globus and BOINC Based Systems*, chapter 4, pages 71–85. John Wiley & Sons, 2008.

[101] Sharon M. Ordoobadi. Development of a supplier selection model using fuzzy logic. *Supply Chain Management: An International Journal*, 14(4):314–327, 2009.

[102] Charles E. Osgood, George J. Suci, and Percy Tannenbaum. *The Measurement of Meaning*. University of Illinois Press, September 1957.

[103] Abdelkader Outtagarts. Mobile agent-based applications: A survey. *IJCSNS International Journal of Computer Science and Network Security*, 9(11):331–339, 2009.

[104] Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Found. Trends Inf. Retr.*, 2(1-2):1–135, January 2008.

[105] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up?: Sentiment classification using machine learning techniques. In *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - Volume 10*, EMNLP '02, pages 79–86, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[106] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, October 2003.

[107] Gabriele Pierantoni. *Social Grid Agents*. PhD thesis, University Of Dublin, Trinity College, September 2008.

[108] Gabriele Pierantoni, Brian Coghlan, and Eamonn Kenny. Social grid agents. In Nikolaos P. Preve, editor, *Grid Computing*, Computer Communications and Networks, chapter 6, pages 145–170. Springer London, London, 2011.

[109] Rajesh Raman, Miron Livny, and Marv Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, 2(2):129–138, September 1999.

[110] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 28–31, 1998.

[111] Magnus Sahlgren. *The Word-Space Model: using distributional analysis to represent syntagmatic and paradigmatic relations between words in high-dimensional vector spaces.* PhD thesis, Stockholm University, 2006.

[112] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[113] Dru Sepulveda and Sebastien Goasguen. The deployment and maintenance of a Condor-Based campus grid. In *Advances in Grid and Pervasive Computing*, pages 165–176. 2009.

[114] Munindar Singh. Agent communication languages: Rethinking the principles. In *Communications in Multiagent Systems*, pages 37–50. 2003.

[115] Esteban L. Soto. FIPA agents messaging grounded on web services. In *Proceedings 3rd International Conference on Grid and Pervasive Computing*, May 2008.

[116] M. Taufer, D. Anderson, P. Cicotti, and C. L. Brooks. Homogeneous redundancy: a technique to ensure integrity of molecular simulation results using public computing. *Parallel and Distributed Processing Symposium, International*, 2:119a+, 2005.

[117] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 17:2–4, 2005.

[118] Tommy Thorn. Programming languages for mobile code. *ACM Comput. Surv.*, 29(3):213–239, September 1997.

[119] David Toth and David Finkel. Comparison of distributed file based tasks for PRC. In *The 17th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 398–403, November 2005.

[120] Andrew Trotman, Charles L. A. Clarke, Iadh Ounis, Shane Culpepper, Marc A. Cartright, and Shlomo Geva. Open source information retrieval: A report on the SIGIR 2012 workshop. *SIGIR Forum*, 46(2):95–101, December 2012.

[121] M. Ughetti, T. Trucco, and D. Gotta. Development of Agent-Based, Peer-to-Peer mobile applications on ANDROID with JADE. In *The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 287–294. IEEE, September 2008.

[122] Etienne Urbah, Peter Kacsuk, Zoltan Farkas, Gilles Fedak, Gabor Kecskemeti, Oleg Lodygensky, Attila Marosi, Zoltan Balaton, Gabriel Caillat, Gabor Gombas, Adam Kornafeld, Jozsef Kovacs, Haiwu He, and Robert Lovas. EDGeS: Bridging EGEE to BOINC and XtremWeb. *Journal of Grid Computing*, 7(3):335–354, September 2009.

[123] Pascale Vicat-Blanc, Sébastien Soudan, Romaric Guillier, and Brice Goglin. Bandwidth on demand. pages 189–202.

[124] John Walsh, Brian Coghlan, and Stephen Childs. An introduction to grid computing using EGEE. In José Gracia, Fabio Colle, and Turlough Downes, editors, *Jets From Young Stars V*, volume 791 of *Lecture Notes in Physics*, pages 47–80. Springer Berlin Heidelberg, 2009.

[125] M. Wooldridge. Agent-based software engineering. *IEE Proceedings - Software Engineering*, 144(1):26–37, 1997.

[126] Michael Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, Massachusetts, 2000.

[127] Michael Wooldridge and Paolo Ciancarini. *Agent-Oriented Software Engineering*, volume 1, chapter 21, pages 487–522. World Scientific Publishing, December 2001.

[128] Michael Wooldridge and Nicholas Jennings. Agent theories, architectures, and languages: A survey. In *Intelligent Agents, Springer-Verlag LNAI 890*, volume 890 of *Lecture Notes in Artifiicial Intelligence*, pages 1–39. 1995.

[129] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(02):115–152, 1995.

[130] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. A methodology for agent-oriented analysis and design. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 69–76, New York, NY, USA, 1999. ACM.