Distributed Coordination of Policy Execution across Autonomous Elements

Paddy Meyler

A dissertation submitted to the University of Dublin,

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

2006

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: ______________

Paddy Meyler

9th October 2006

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: ______________

Paddy Meyler

9th October 2006

Acknowledgements

I would like to thank my supervisor, Mr. Vinny Wade, for his guidance, help and patience over the past nine months.

To the NDS class, for all the laughs and support during the year.

Finally, I would like to dedicate this dissertation to Tom and Sean who arrived, appropriately enough, during Chapter 2 and to Noreen who keeps it all together.

Paddy Meyler
University of Dublin, Trinity College
9th October 2006

// Abstract

Centralised and hierarchical management architectures are often unsuitable for managing large-scale distributed systems based on autonomous peers, such as service-oriented solutions, ubiquitous systems, grid computing and dynamic communications networks. Current research is investigating the use of more distributed management architectures incorporating policy-based management and autonomic computing to provide more cost-effective, scalable and robust solutions for managing these systems.

Current policy-based management approaches are based on central or hierarchical architectures where a single decision point is responsible for policy execution regarding global decisions. However, in distributed autonomous systems the autonomic elements must collaborate with their peers to make global decisions, which raise a number interesting questions and challenges.

The purpose of this dissertation was to investigate the key issues and challenges associated with implementing collaborative policy execution and to propose and prototype distributed coordinated mechanisms for collaborative policy execution for a system of autonomous services.

Specific requirements for policy coordination in an environment of autonomous peers were identified. A prototype solution was designed and implemented using an agent-based autonomic manager that could be integrated in the application specific autonomous elements to support global and local policy execution.

The prototype evaluation showed that policy execution across autonomous nodes based on the agent approach is feasible for small numbers of simultaneous policy executions and that specific policy types and interaction mechanisms and are needed to support global coordination while preserving element autonomy.

Table of Contents

1 Introduction 1

1.1 Background 1

1.2 Motivation 1

1.3 Objectives 2

1.4 Research Approach 2

2 State of the Art 3

2.1 Introduction 3

2.2 Policy-Based Management 3

2.2.1 Overview of Policy-Based Management 3

2.2.2 Standards and Approaches 4

2.2.3 Implementations 6

2.2.4 Research Issues 7

2.3 Autonomic Computing 8

2.3.1 Overview of Autonomic Computing 8

2.3.2 Approaches 9

2.3.3 Implementations 10

2.3.4 Research Issues 11

2.4 Associated Research Areas 12

2.4.1 Multi-Agent Systems 12

2.4.2 Distributed Management 13

2.5 Summary/Conclusions 14

3 Design 15

3.1 Introduction 15

3.2 Example Scenario 15

3.2.1 Basic Description 16

3.2.2 Management Data 16

3.2.3 Policies 17

3.3 Considerations 18

3.3.1 Functional Requirements 18

3.3.2 Non-functional Requirements 21

3.4 Approach 21

3.4.1 System Overview 21

3.4.2 Policy Handling 23

3.4.3 Shared Information and Community Handling 26

3.4.4 Distributed Coordination 26
3.4.5 Policy Execution 28
3.5 Main Traffic Cases 29
3.5.1 Element Start-up 29
3.5.2 Global Policy Execution 31
3.6 Design Issues and Decisions 33
3.7 Summary/Conclusions 34
4 Prototype Implementation 35
4.1 Introduction 35
4.2 Technical Architecture 35
4.2.1 System Overview 35
4.2.2 Component Description 36
4.3 Technologies 38
4.3.1 P2P Agent Layer and Messaging 39
4.3.2 Service Discovery and Lookup 40
4.3.3 Policy Implementation 40
4.3.4 Persistence and Transactions 41
4.3.5 Host, Libraries, Tools 41
4.4 Agent and Behaviour Implementation 42
4.4.1 Agent and Behaviour Design 42
4.4.2 Implementing Concurrency 42
4.4.3 Implementing Conversations 43
4.5 Integrating the Managed Element 44
4.5.1 Implementing the Scenario 44
4.5.2 Integrating a new Element 47
4.6 Implementation Issues and Decisions 48
4.7 Summary/Conclusions 49
5 Evaluation 50
5.1 Introduction 50
5.2 Evaluation Criteria 50
5.3 Evaluation Tests 51
5.4 Findings 51
5.4.1 Functional Requirements 51
5.4.2 Non-Functional Requirements 54
5.5 Summary/Conclusions 61
6 Conclusions 62
6.1 Main Conclusions 62
6.1.1 Objectives and Achievements 62

6.1.2 Improvements 64
6.2 Future Work 64
6.2.1 Summary 64
7 Appendix 66
7.1 Policy Schemas and Policy Examples 66
7.1.1 Policy Schemas 66
7.1.2 Policy Examples 68
7.2 Abbreviations 72
8 Bibliography 74

List of Figures

Fig. 1: Example Policy Specified in Ponder 3

Fig. 2: IETF Policy Architecture. 5

Fig. 3: Autonomic Manager Based on IBM Approach 10

Fig. 4: Application Environment 16

Fig. 5: Example Managed Objects for Resources 17

Fig. 6: Example Policies 18

Fig. 7: System Overview 22

Fig. 8: Role Hierarchy 23

Fig. 9: Policy Schema 23

Fig. 10: Concurrent Global Executions 28

Fig. 11: Element Start-up Sequence Diagram 30

Fig. 12: Global Policy Execution Sequence Diagram 31

Fig. 13: System Architecture 35

Fig. 14: Reference architecture of the FIPA Agent Platform 36

Fig. 15: Utility Function for Thread Resource 45

Fig. 16: Extract from Example Thread MO Interface 46

Fig. 17: Extracts from an Example Optimisation Policy 46

Fig. 19: E2E Response Times for Concurrent Global Policy Executions. 55

Fig. 20: Average Local Manger Processing Times 55

Fig. 21: Affect of Buffer Time on E2E Response. 56

Fig. 22: Time Spent Processing Duplicate 56

Fig. 23: Start-up Times 57

Fig. 24: System Message Model 58

Fig. 25: Example System Message Volumes 58

List of Tables

Table 1: Policy Handling Requirements 19

Table 2: Shared Information and Community Handling Requirements........ 20

Table 3: Global Execution Requirements 20

Table 4: Element Specific Requirements 20

Table 5: Non-Functional Requirements 21

Table 6: Autonomic Manager Components 38

Table 7: Summary of Technologies Considered and Selected........ 39

Table 8: Agent and Behaviour Model 42

1 Introduction

1.1 Background

The complexity of current networks and systems, and the cost of managing these in the face of changing business objectives, has motivated the development of two complementary disciplines Policy-Based Management and Autonomic Computing.

Policy-based management is an approach to managing systems using business-oriented rules which are translated into low-level configurations required by the underlying technologies. Autonomic computing systems are systems that can manage themselves given high-level objectives or goals expressed as policies.

Current centralised and hierarchical management architectures are often unsuitable for managing large-scale distributed systems, such as service-oriented solutions, ubiquitous systems, grid computing and dynamic communications networks. Current research is investigating the use of more distributed management architectures to provide more scalable and robust solutions suitable for peer oriented systems and networks.

Autonomic elements, such as devices or software services, are managed using local policies specific to an element or global policies related to a group of elements. For example, a global policy could be used to configure the allocation of some shared processing resource. Current policy-based management approaches are based on a central or hierarchical architecture where a single decision point makes the global decision based on a global view of shared resources. However in a distributed solution the autonomic elements must collaborate with their peers to make the global decision, which raises a number of challenges.

1.2 Motivation

The main challenge is to determine how autonomic elements can interact to fulfil the global policy objective. This involves two separate but complementary approaches based on the following questions:

- What mechanisms are needed to support high level policy refinement for autonomic behaviour in the local elements?

- What interaction and shared knowledge is needed for the autonomic elements to coordinate in order to fulfil the policy objective?

This project concentrates on the second question and attempts to determine the necessary interactions between the autonomous elements with particular reference to the use of policies for configuring shared resources.

1.3 Objectives

The purpose of this project is to investigate the key issues and challenges associated with implementing collaborative policy execution and to propose and prototype distributed coordinated mechanisms for collaborative policy execution for a situation involving aggregated services.

This raises four main questions that the project attempts to answer:

- What types of policies are needed for configuring global resources and how are they specified and distributed to the elements?
- What information, or knowledge, is needed centrally to support the coordination?
- What coordination mechanisms are required?
- How can this be generalised for different applications?

1.4 Research Approach

The following research approach was adopted and reflected in the dissertation chapters:

- Investigate the current approach to policy coordination between autonomic elements in a distributed environment and coordination mechanisms used in associated areas
- Identify key requirements for policy coordination in an environment of autonomous peers based on the four main questions outlined in the objectives
- Design a prototype with suitable mechanisms for policy coordination
- Develop the prototype using relevant technologies
- Use the prototype to evaluate the implemented mechanisms and determine its strengths and weaknesses according to the requirements and standard evaluation criteria for relevant areas
- Form key conclusions and ideas for future research in this area.

2 State of the Art

2.1 Introduction

This chapter provides a brief overview of a number of relevant disciplines, including: Policy-Based Management (PBM), Autonomic Computing, Multi-Agent Systems (MAS) and Distributed Management. The key concepts are explained and current research issues, which are often common to the different areas, are highlighted.

2.2 Policy-Based Management

2.2.1 Overview of Policy-Based Management

Policy-based management is an approach to managing systems using business-oriented rules which are translated into low-level configurations required by the underlying technologies. There are two main, but not mutually exclusive, interpretations of the term 'policy' best captured by the following IETF definition [1]: "*A definite goal, course or method of action to guide and determine present and future decisions*" and "*a set of rules to administer, manage, and control access to network resources*".

PBM attempts to remove the business rules from the managing software and to express them in a form that is more flexible and easy to change by administrators. This simplifies the administration and management tasks, which is especially important for current "*complex and heterogeneous systems*" [2]. The rules maybe expressed as high-level goals representing business rules or service level agreements (SLAs) and are mapped or refined to the necessary low-level device (or software) settings. The most common approach to specifying policy is through an (event)-condition-action (ECA) rule or action policy as shown in the example below.

```
backup01 O+                      Name and Modality
at 02:30                         Event
/archiver                        Subject
{ backup() }                     Action
/db                              Target
when dbJobsState == complete     Condition
```

Fig. 1: Example Policy Specified in Ponder

An ECA policy rule normally consists of a number of basic attributes, as defined by Moffet [3], and shown in the example above. The modality indicates the general class of policy, that is, Authorisation (what activities the subject allowed, +, or not allowed, -, to do) or Obligation (what activities the subject should or should not do). The subject specifies the object(s) that carry out the policy and the target specifies the objects that the policy applies to. These objects are usually specified as sets so that they can apply to a range, or domain, of objects that have a certain function.

PBM has been used for different management scenarios, such as network, system, application and enterprise management) and for different management functions (FCAPS). It has been applied successfully to security and network QoS [4] and more recently to SLA monitoring in e-business and autonomic computing. Some hold the view that PBM is only suitable for specific areas as business policies from different domains differ so much, while others believe it can be used where ever decisions need to be made.

Besides the authorisation and obligation policy classes as defined in [5], there are many other ways to categorise policies (Weiss [6]), but there is less agreement on these classifications. Examples include categorising policies by: function ('ordinary' policies and meta-policies - policies for making decisions about policies), purpose (such as, configuration, security, and installation) and representation (action, goal or utility). Policy level is used to distinguish between high-level (human-oriented or abstract) policies which are not directly interpretable by an element or low-level (concrete) policies which are directly interpretable by some element. Further policy levels or views can exist between these extremes, and Strassner [7] suggests a policy continuum consisting of five levels or views: business, system, network, device and instance. Policies are also distinguished by whether they are static or adaptive - adaptive policies can be changed or selected automatically in response to their dynamic environment. A further classification is based on when they are executed, beforehand to pre-provision some resource or executed on-demand to set the resource at run-time, usually in response to some traffic requirement.

2.2.2 Standards and Approaches

Policy Architectures

The IETF/DMTF [1] developed a standard architecture for PBM called the Policy Management Framework. Most current PBMs are based on this architecture or the newer architecture adopted by the TMF [8]. Many PBMS based non-standard architectures are also implemented. The Policy Management Framework contains the following logical components:

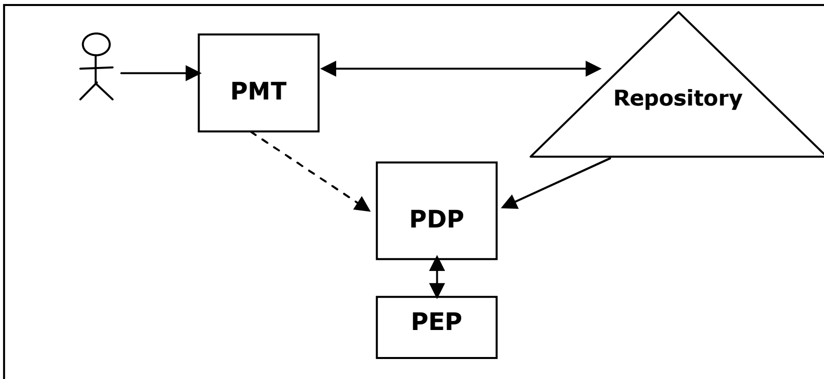

Fig. 2: IETF Policy Architecture.

Policy Management Tool: The PMT is used to manage the policy lifecycle (create, distribute, activate, deactivate, modify etc.). When policies are defined they are usually validated, checked for conflicts, mapped to low-level policies and checked to ensure they are feasible for the devices before being stored in the repository.

Policy Repository: This component stores the (high and low-level) policies so that they can be accessed at a central point by the PDP(s). Policies are ideally stored with the descriptions of the resources (e.g. a network router or service) that they apply to according to some standard information model, used to ensure device independence. The information model is then mapped to some underlying storage schema, such as LDAP server, an RDBMS or the basic file system.

Policy Decision Point: The PDP retrieves the relevant policies, transforms them into a form understood by the PEP, checks the policy conditions based on environmental state and keeps track of policy changes as notified by the PMT.

Policy Enforcement Point: This node enforces the policy's decision by changing the managed device or software.

The four logical components can be deployed in a number of ways. For example, the PDP and PEP can be collocated, or a PDP can control a number of PEPs or there maybe several PDPs arranged hierarchically with local PDPs deferring certain decisions to a higher level PDP.

Policy Specification

There are a number of approaches to specifying policies, largely based on policy languages or frameworks (policy information models).

ECA or action policies typically consists of the attributes discussed earlier, but may include additional attributes, such as priority [9], exceptions and pre and post-conditions [7]. Policy subject and target sets are usually specified by some domain expression as exemplified in Ponder [10]. In addition events can be composite and parameterised.

These are many specific policy languages, suitable for security management (XACML, PERMIS) or network QoS (for example, SRL and PPL used for low-level traffic flow), Cfengine, as well as more general languages such as Ponder, PDL [11] or PMAC's ACPL [12] . Newer 'languages', such as Rei and KAoS represent policies using logic, which enables them to be more easily analysed for conflicts and inconsistencies but these are often more difficult to use [13] [14].

Policy frameworks or information models are standardised objected-oriented schemas that represent policy in a language and vendor-independent manner. This makes them more suitable for mixed-vendor environments, which most environments either are, or are becoming, today. These models can be mapped to specific policy languages, for example CIM-Ponder [15]. The two main information models for representing policy are PCIM [16], which is part of CIM [9, 17] and DEN-ng [8] which is part of the TMF Shared Information/Data Model (SID). These models are largely similar. There are also particular refinements, such as the QoS Policy Information Model (QPIM), which is based on PCIM.

2.2.3 Implementations

Many vendors supply systems incorporating PBM, although the degree to which they are policy-based is sometimes questionable [7]. Examples of real PBM solution include Nortel's WEnterprise Network Management System, Cisco's QPM and Security Policy Manager, Hitachi's PolicyXpert and Intelliden's management products.

PBM is being incorporated into many new telecommunications products, largely influenced by TMF's NGOSS initiative and has being applied to products at the different network layers, including switching (Netmon for SARAS from Bell Labs), 3G QoS [18] and service layer enablers [19] and OSS [20]. Other notable PBM prototype implementations include PECAN (network based management of MPLS), DIOS++, Globus (PBM was used for GRID security), CASSIS (based on IBM PMAC).

There is now a general acceptance that policy is a 'good thing' by industry and by the main standards bodies (IETF, DMTF, TMF, W3C). PBM is being incorporated into many new systems and is being extended beyond its traditional network and security use into areas such as autonomic computing, grid computing (Globus) and service oriented computing. However the success of PBM, like any other technology, depends on overcoming many industry challenges, such as interoperability, and scientific challenges as described next.

2.2.4 Research Issues

Many of the key PBM research issues are neither new nor particular to policy, and often have their roots in Artificial Intelligence.

Policy Authoring

There are many questions here in terms of how best to capture the semantics of business policies and SLAs and how to represent these in policies.

Policy Representation

There is much disagreement over 'the' best way of representing policies, and whether there is one way to represent policies from diverse domains. If one language or framework is to be used then should it be procedural (e.g. PDL) or declarative (e.g. Ponder) and should it be logic based and support reasoning over ontologies (e.g. Rei). Those against attempting to develop the 'Java' of the policy world point to the difficulty of this since the language must provide support for: a wide and varied range of policies from different domains, conflict detection and resolution and preferably some formal analysis as well a supporting ECA rules, goal and utility policies.

Conflict Detection and Resolution

The automatic detection and resolution of conflicts has, from early PBM research [3], been identified as a difficult problem. This is part of a larger issue on how to formally check the policies to guarantee their correctness and function.

There are many causes and types of conflicts [21], which can be classified into conflicts based on the policy specification (where the same event and constraint leads to two different policies) or based on the application (overlap due to application of different policies to a resource). Conflicts can only occur when there is some overlap between the subject and/or target objects. Analysing policies for conflicts is difficult to automate and suited to declarative policy representations.

There are two types of conflict analysis. Static analysis is ideally performed by the policy management tool when the policies are specified and before deployment. However only certain conflicts are detected in this way as it is not possible to automatically determine what was meant by the policy author. Some success has been made in static conflict analysis, while dynamic conflict detection (that is, during runtime), still requires a lot of research. Some techniques for resolving conflicts include setting policy priorities, for example as used in PMAC (although this is not scalable) or the use of meta-policies, as used in Rei or based on some learning technique (Refined Learning [22]).

Policy Refinement

The objective of policy refinement is to transform the high-level, abstract, policy specifications into low-level, concrete specifications that are capable of being directly interpreted by the target device. The ultimate aim of PBM is to automate this process as far as possible. The main objectives of policy refinement have been defined by Moffet and Sloman [3] as to:

- Determine the resources that are needed to satisfy the requirements of the policy.
- Translate high-level policies into operational policies that the system can enforce.
- Verify that the lower level policies actually meet the requirements specified by the high-level policy.

There are different approaches to policy refinement [23], and some successful approaches have been implemented but with very limited application (such as table lookup - Network QoS (Verma)) or using refinement templates (POWER toolkit). More recent approaches are based on the use of formal specification languages [24], but it is still an open question on how far this process can be automated.

Adaptive policy

While policies are much easier to change than modifying management software they still require costly human intervention to update. An interesting area of research is adaptive or dynamic policies, where the PBMS changes the policy parameters, usually based on some learned values, or uses some meta-policy to select suitable policies to execute at runtime [25].

2.3 Autonomic Computing

2.3.1 Overview of Autonomic Computing

'Autonomic' means acting or occurring involuntary, such as the autonomic nervous system. The term 'autonomic computing' was defined by IBM as "*systems that have the ability to manage themselves and dynamically adapt to change in accordance with policies and objectives*" [26].

While the term 'autonomic computing' maybe new, the concept has been applied in advanced telecommunications, space and military systems for some time. General computing technologies such as RAID and PC self-configuration can also be considered autonomic to some degree.

The move towards autonomic computing has been motivated by the need to reduce the escalating costs of system management due to the complexity of current systems. Other goals

include making the systems more adaptable to the business needs, improving service levels and 'hiding' the complexity of managing from the humans. Note the similarity here with PBM, which is a key component of autonomic computing.

There are four key elements or properties of an autonomic computing system [27]:

- Self-configuring: Can configure and reconfigure itself dynamically based on high-level policies.
- Self-optimise: Monitors and tunes itself to achieve goals.
- Self-healing: Discovers problems and tries to keep the system running smoothly.
- Self-protecting: Ensures security and integrity.

The autonomic system must also be aware of its own resources, capabilities and its connections to other systems. It must be able to discover knowledge about its environment, function in a heterogeneous environment as well as anticipate and adapt to user needs.

It is difficult to achieve these properties in the short term for large complex systems despite the industry will and availability of the necessary technologies. It is expected that systems will evolve over time to become fully autonomic and systems can be categorised according to their level of autonomicity, as defined by IBM in [28].

2.3.2 Approaches

This section provides a brief description of an autonomic manager based on the IBM approach, which is the most widely known approach, however other companies have developed their own approaches, such as Motorola, Cisco (Adaptive Services Framework) and Intel (Proactive Computing).

The central component in an autonomic system is the autonomic manager as shown below [29].

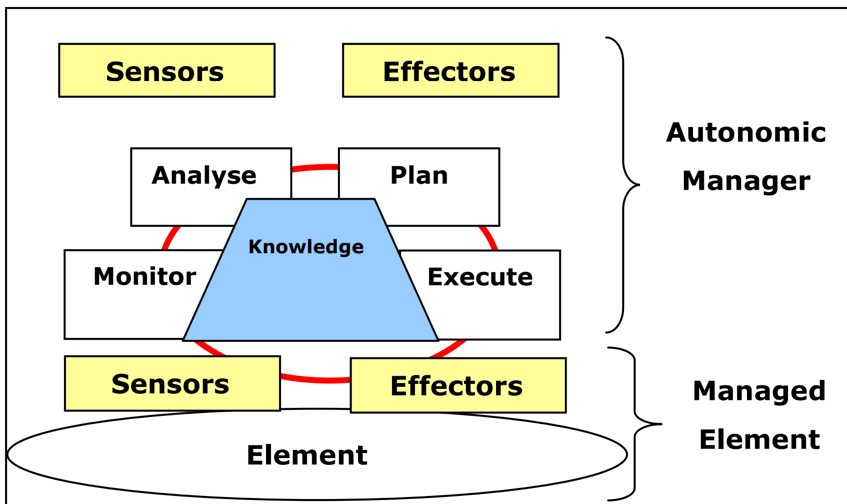

Fig. 3: Autonomic Manager Based on IBM Approach

This implements the MAPE loop, which is a control loop for managing the element (for example, server, database, storage area network or software application). The Monitor function collects and filters information, such as performance metrics, from the managed element via the sensor 'interface'. The Analyse function uses modelling and forecasting techniques, such as queuing theory, to try to predict future changes in the environment. The Plan function decides what actions need to be taken and the execute function executes the plan, or policy, to affect the element via effectors. Autonomic managers do not act in isolate and usually need to collaborate with other managers to fulfil some overall function. Note that policy can be used in any of the MAPE functions.

The Motorola approach [30] is similar but with slightly different terminology and the emphasis is on autonomic networking. For example, they refer to an observe-learn-plan-execute-understand loop and include a model based translation layer to convert to device specific managed resource as well as sharing learned knowledge in the system.

2.3.3 Implementations

There are several commercial implementations of products displaying higher levels of autonomicity and many prototypes have been developed, however this is still a new and growing area. Examples include IBM's SMART DB2 which includes self-optimising and self-configuration of the database, IBM's Tivoli management product and Sun's N1 Datacentre.

Notable prototypes/projects [31] includes an autonomic data storage system called OceanStore and Oceano which manages computing resources for software farms and many prototypes from telecommunications vendors such as Motorola [22], concentrating on network-centric or Grid approaches (Motorola).

IBM provides an autonomic toolkit called Emerging Technologies Toolkit (ETTK) which can be used to incorporate autonomic behaviour into products [32], which included the PMAC product for supporting PBM in autonomic systems.

2.3.4 Research Issues

General Research

Since this is a relatively immature area there are many industry and scientific challenges to overcome, as outlined by Parashar [31] and Murch [27]. Some of these issues are the policy challenges discussed above and not repeated here, such as refinement and goal specification.

Main autonomic research issues can be summarised as:

- Defining models for specifying and implementing autonomic behaviour
- Handling the relationships between the autonomic elements in terms of discovery, negotiation, establishing relationships dynamically and dealing with different contexts.
- Learning and optimisation theory.
- Developing autonomic applications, composing elements and testing.

Policy Based Management for Autonomic Computing

Kephart and Walsh [33] have proposed three policy types: Action (ECA), Goal and Utility policies, for autonomic computing as they believe that action polices alone are too restrictive in that they require the policy author to know in advance what has to be done. Goal policies are more flexible in that they specify a desired state to achieve and some planner/optimisation algorithm generates the behaviours to achieve that state. Utility policies are based on utility theory [34], which is concerned with the maximisation of some level of utility (an abstract measure of happiness/satisfaction/benefit) gained from the use of some resource(s). A utility function/curve maps a state (or sequence of states) onto a real number that represents the degree of happiness. This value can be used to decide the best allocation of the resources. A Utility function policy uses a utility function to represent different states and then tries to achieve the state to provide the best overall utility. However, the main problem with utility theory is that it is difficult to come up with utilities functions that express the relative benefits of different states. The advantage from a PBM perspective there is less (but still some) chance of conflicts.

Other PBM related topics of interest to autonomic computing include adaptive policy as outlined previously.

2.4 Associated Research Areas

Concepts from a number of additional areas have been applied in this dissertation.

2.4.1 Multi-Agent Systems

Approaches to distributed decision making are largely based on traditional distributed systems or Multi-Agent Systems (MAS). The main differences is that traditional distributed systems hardwire coordination and cooperation at design time whereas agents in a MAS act autonomously by making decisions at run time and can dynamically coordinate their activities with others [35].

General MAS can be divided into two main classes, Cooperative Distributed Problem Solving (CDPS) and Multi-Agent Systems (MAS), based on whether they act in their own interest or cooperatively. General cooperative problem consists of three main steps:

- Problem decomposition : Problem is decomposed and allocated to agents
- Sub-problem solution : Each agent solves individual problem
- Solution integration : Individual solutions are combined.

An agent can be defined as "*an autonomous problem-solving computational entity capable of operating in dynamic and open environments*". Agent technology is relatively new and agents provide a suitable design model for service-oriented computing [36] and autonomic systems [27]. Many agent based systems have been developed for manufacturing, process control, e-commerce, transportation and entertainment. Agents normally posses the following properties:

- Autonomous: Acts on its own to achieve it goal, is knowledgeable, persistent, reactive and proactive.
- Cooperative (Communicative): interact with other agents to fulfil some common goal. Involves some negotiation, coordination strategy and interaction protocol.
- Adaptive: Agents adapt to their environment, through machine learning and dynamic interaction.

In MAS coordination models tend to be control driven (usually event based) or data-driven (shared data spaces). The main standards bodies for MAS is FIPA (Foundation for Intelligent Physical Agents [37]) and the OMG. Current research [38] areas include coordination mechanisms, negotiation and learning.

2.4.2 Distributed Management

Management Architectures

Management architectures can be divided in to three types:

- Centralised
- Hierarchical
- Distributed.

The main difference is that the centralised model is prone to failure but has low communication or coordination overhead, while the hierarchical and distributed (P2P and levels of P2P) approach is more robust, splits the management load but involves far more communication.

PBM Distribution

There are different ways of deploying the standard IETF/DMTF architecture as outlined above. For example, having one central PDP making all the decisions or a hierarchical approach where a local PDPs making local decisions. The DEN-ng architecture also provides for different deployments and hierarchies [7] where a number of domain PDPs exist within policy servers which are interconnected by policy brokers that are used to coordinate the application of different policies.

Specific examples of hierarchical approaches include Tsai [39] who argues that centralised PBMs are not suitable for service-oriented systems due to their size, distribution and range of policies they support. A centralised PDP would be a bottleneck and due to communication from the large number of components. A hierarchical policy-based framework for service-oriented systems is then proposed based on local and global policy enforcers. Another hierarchical approach, for ad hoc networks based on three levels of policy agents, is presented in [40].

The IETF/DMTF architecture does not make any reference to a fully distributed architecture for PBM and in fact makes "*no provision for inter-PDP communication*" [41] as PBM approaches generally assume a single administration domain where all resources are controlled by a single domain.

There is currently a lot of interest in using the P2P approach for management (e.g. Madeira project [42] and [43]), which is necessary to mange the size and complexity of current and future systems and the interconnection of systems existing in different domains as exemplified by the service-oriented computing, grid computing, ubiquitous computing and dynamic communications networks.

This is a relatively new area and very little work has been published regarding distributed PBM. There are many questions regarding how best to apply global policies to autonomous elements. Some relevant papers include the following. Baliosian et. Al in [22] propose a fully decentralised approach for the self-configuration of NGN radio access networks where a PDP is embedded in each node (base station) and receives events from other network elements. Carey in [44] uses a refinement approach to determine the local policies for ensuring QoS for composite services. In [45] Chadwick proposes a solution for coordinating security policies in a grid environment by using coordination objects stored in a distributed DB or shared space.

2.5 Summary/Conclusions

This dissertation includes concepts from a number of areas, including policy-based management, autonomic computing, multi-agent systems and distributed management. An overview of the main concepts from these, often overlapping, areas was presented along with current research issues to solve.

3 Design

3.1 Introduction

The main objective is to design a system where policies can be executed across autonomous elements while preserving the autonomy of the elements. This objective raises four main questions:

- What types of policies are needed and how are they specified and distributed to the elements?
- What information, or knowledge, is needed centrally to support the coordination?
- What coordination mechanisms are required?
    - How do we know who is involved in the distributed decision making?
    - What generic interactions are necessary between the coordinating elements?
    - How do the elements agree on the result?
- What application specific information or logic is required?

Designing a suitable system raises many issues and challenges regarding communication mechanisms, heterogeneous elements, reaching agreement, PBM issues - conflicts, refinement, representation and modelling – and typical distributed computing issues, such as element discovery, message ordering, failures, delays and concurrency. While it is not practical to cater for all of these issues it is important to consider them in the design.

This chapter lists the requirements of a suitable solution and outlines mechanisms for solving particular problems. Traffic cases are presented in an attempt to tie the various parts together and the chapter concludes with a short discussion of the many design decisions taken along the way.

3.2 Example Scenario

The application for this system can be any set of autonomous elements. The emphasis is placed on services that can be composed and the important element features are their autonomy and

their part in the sharing of some resource. Elements therefore need to coordinate the management of the resource based on local and global constraints.

3.2.1 Basic Description

A number of basic services are used by a simple financial web site. These include a foreign exchange converter, interest calculator and a database service for storing the necessary rates used by the other two services. Each service implements a number of application threads for handling user requests. A finite number of threads exist for the platform and assuming the services share this platform they must somehow decide the thread allocation between them. Requests are queued by each service while awaiting a free thread from its local pool

These autonomous services are composed to provide an overall service. Each basic service consists of a functional part that exposes a functional interface, used to perform the actual service, and a management part that exposes a management interface used to configure and control the functional part and general QoS.

The management part consists of an autonomic manager that implements an MAPE loop and implements policies to guide the functional part. The management part monitors the use of the elements resources and reconfigures these resources using policies.

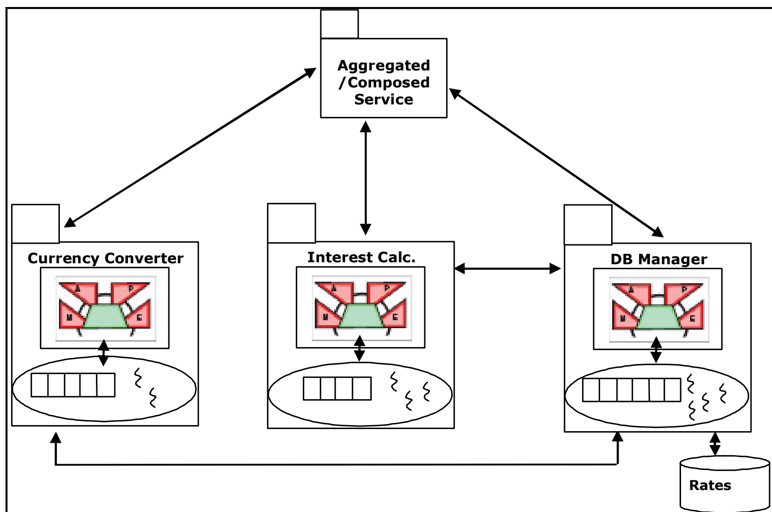

Fig. 4: Application Environment

3.2.2 Management Data

Managed Objects

Each service has a set of managed objects (MOs) as shown in the figure below. The statistics MO is updated with current usage values every monitoring period. Threads are considered

global resources and the thread MO in each service represents a local set of threads allocated from a global pool. The queue MO is a local resource, in that each service is free to allocate its own queue size.

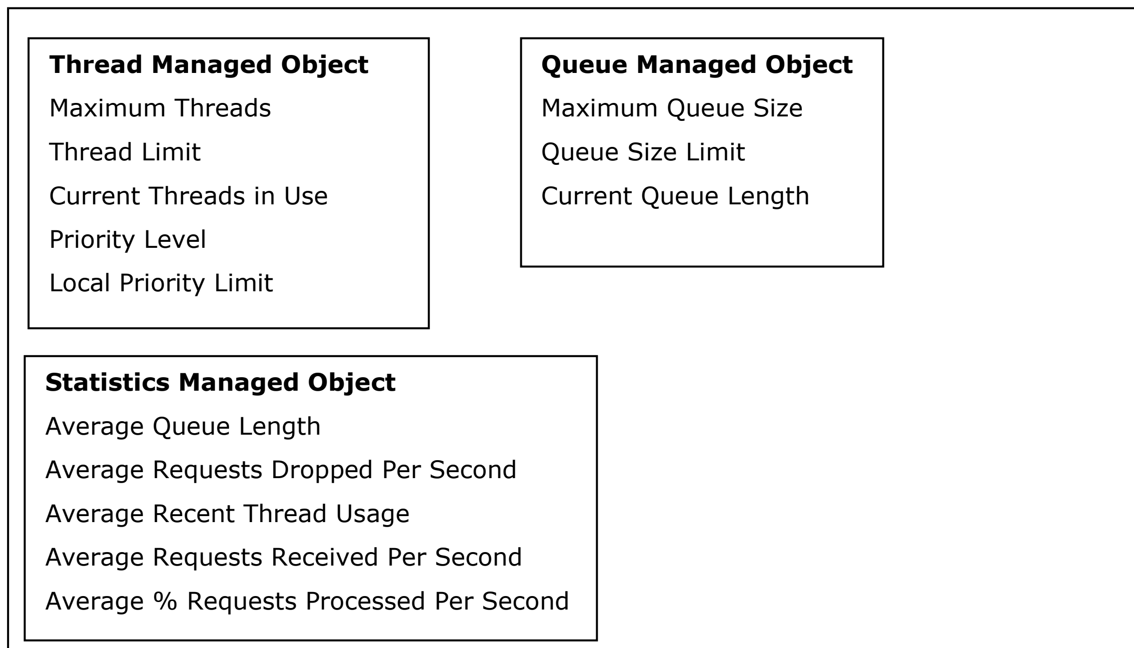

Fig. 5: Example Managed Objects for Resources

Each of the services has QoS data that affects the performance of the service. Requests for a service are queued and handled by a thread from the application thread pool once available. If the queue is full then the request is dropped. The thresholds indicate the level of dropped requests in a monitoring period before a warning or severe error situation arises. Increasing the number of threads reduces the number of dropped transactions and waiting time in the queue but increases the processing time per request. Therefore there is a trade off between the processing time per request and overall requests handled.

The non-functional aims of the application are to provide a suitable QoS to customer requests and to optimise throughput. For example, overall throughput can be increased by reducing queuing time and reducing processing time in both services and by reducing the number of dropped requests. This means reducing the request response time, reducing the number of dropped requests and increasing throughput. The autonomic manager uses policies to reconfigure the resource in an attempt to achieve its QoS objectives.

3.2.3 Policies

The operation of the non-functional behaviour (QoS assurance) of the services is governed or guided by a set of local and global policies. Global policies effect changes in more than one

service in keeping with the service's local constraints. All policies are positive obligation policies concerned with configuration and can be static or dynamic (on-demand).

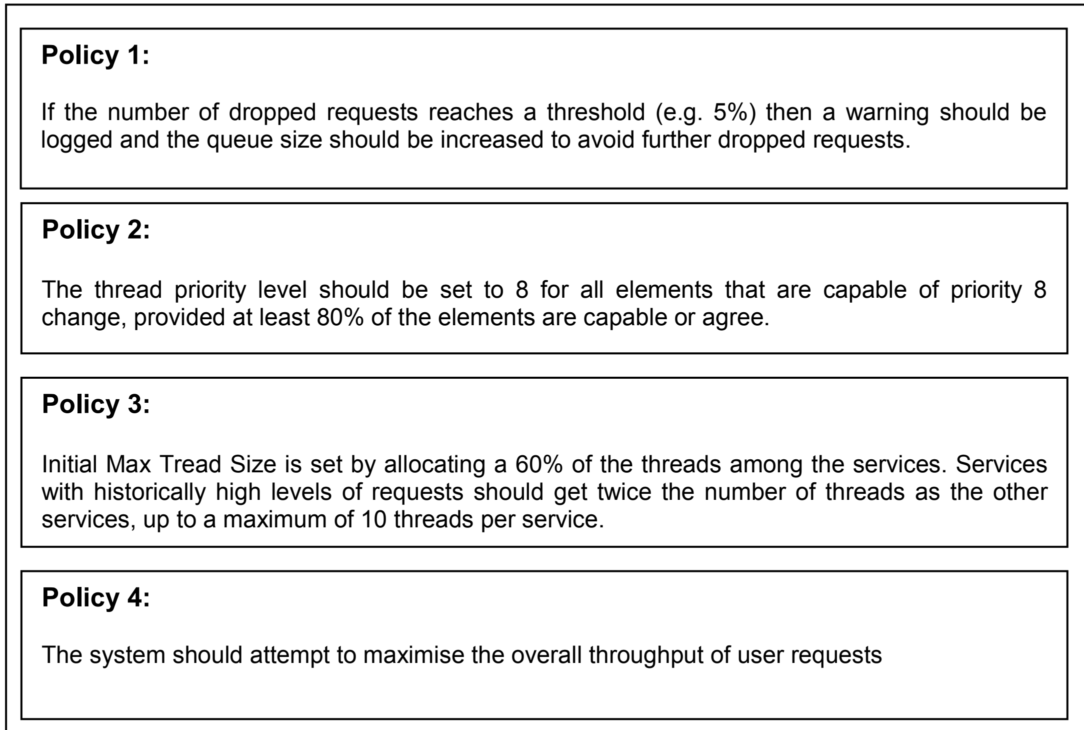

Fig. 6: Example Policies

Policy 1 is a local policy as it is concerned with the queue resource, while the other polices are global. Policy 2 requires agreement on the setting of a local aspect of a shared resource. Policy 3 allocates the resource initially and Policy 4 represents an optimising, or utility, policy where the allocation of the threads is based on the overall system benefit as determined from local benefits. In each of the last three policies a global decision is made based on local constraints.

3.3 Considerations

3.3.1 Functional Requirements

The functional requirements for the four main aspects of the system are listed. Some of the requirements are necessary for a real implementation but not for the prototype. These are indicated as 'Assumed' or 'Partially Assumed', that is, they are assumed to exist and are therefore outside the scope of the dissertation.

Policy Handling

Tag	Description	

PH-1	A simple policy management tool is necessary to manage the policy lifecycle, that is, to create, distribute, activate, deactivate, read, modify and delete policies. This tool also validates the policies and checks them for static conflicts when the policies are defined. A mechanism is required to notify elements of policy changes.	Partly Assumed
PH-2	The system must support two main types of policies: Global policies, which involve more than one element and Local, or element specific, policies.	Provide
PH-3	Polices must be specified formally using a simple specification mechanism or language, based on a standard policy model that is precise and understandable to all autonomous elements.	Provide
PH-4	Policies are stored in a persistent central repository and relevant polices are stored in a persistent local storage on download.	Assume
PH-5	A simple policy deployment model must be provided to deploy policies from a suitable central location. Only policies relevant to the type of element must be returned to the element.	Provide
PH-6	Each element must be capable of generating local events that trigger local or policy execution.	Provide
PH-7	It must be possible to limit the scope of a global policy to an individual element or to any subset of elements.	Provide
PH-8	'Weak' policies must be supported. These are polices that represent general goals or that represent wish states rather than absolute rules.	Provide
PH-9	Each element must be capable of making independent local decisions. Global decisions should be based on the local decisions of the elements involved.	Provide

Table 1: Policy Handling Requirements

Shared Information and Community Handling

Tag	Description	
CH-1	Element admission control is required - join, leave and state monitoring.	Provide
CH-2	It must be possible to locate elements based on their name and/or function irrespective of their distribution across hosts.	Provide
CH-3	Elements must agree on the concepts and semantics in the area covered by the policies (e.g. QoS world) so that they can communicate with other services.	Assume

CH-4	The system should allow coordination between different compositions of element groups (as elements may be added or removed) without software reconfiguration.	Provide

Table 2: Shared Information and Community Handling Requirements

Global Policy Execution

Tag	**Description**	
GE-1	The autonomy of all elements must be respected. That is, no element can directly change the resources controlled by another element or be subject to decisions it was not involved in. All local decisions and changes are subject to local constraints.	Provide
GE-2	Global policy fulfilment must take place within a group of affected elements without a permanent central co-ordinator node. That is, any of the elements should be capable of coordinating the global policy execution.	Provide
GE-3	A standard protocol, implemented by the elements, must be provided to facilitate interaction between elements during global policy execution.	Provide
GE-4	Policy execution should not lead to unstable states between the services – recursive loops or 'route flapping', and the interaction protocol must be efficient.	Provide
GE-5	A number of policies may execute concurrently coordinated by the same or different autonomic managers. Some mechanism must exist to handle simultaneous executions of the same policy.	Provide
GE-6	The system will ensure that coordination interactions display the ACID properties as far as possible.	Assume

Table 3: Global Execution Requirements

Application

Tag	**Description**	
AP-1	Example elements must be provided to demonstrate the policy execution.	Provide
AP-2	Composite services can be constructed from subsets of autonomous services according to some orchestration description using some workflow engine.	Assume
AP-3	Each autonomous element has an autonomic manager that monitors its QoS and takes appropriate action using policies.	Partially Assumed

Table 4: Element Specific Requirements

3.3.2 Non-functional Requirements

Some of the non-functional requirements that this system would demand in a real setting are provided in this prototype.

Tag	Description	
PF-1	Global policies should be completed within a reasonable time period to be useful for dynamic configuration requests.	Provide
SC-1	Up to 10 elements must be able to conclude a global policy execution.	Provide
RL-1	The system must be reliable in a simple way, in that faults are reported and catered for where possible	Partially
EX-1	The system must be extensible, in that it is easy to implement new policies, elements and element specific implementation logic	Provide
IO-1	The system should interoperate with different elements that conform to the interaction protocol	Provide
SR-1	Some mechanism is responsible for the security/authorisation of the system	Assume

Table 5: Non-Functional Requirements

3.4 Approach

3.4.1 System Overview

This diagram shows the four main system components, which can be distributed across different host machines.

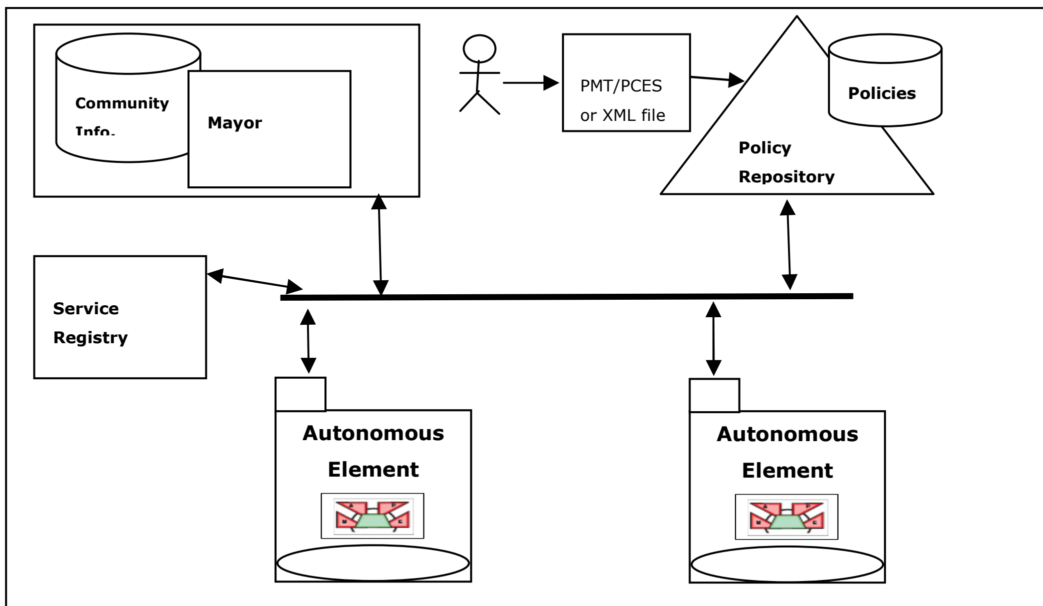

Fig. 7: System Overview

Mayor

The Mayor performs two community based functions: It allows the autonomous elements to join the community based on their capabilities and it provides a member lookup service based on element role.

Policy Repository

Policies are typically supplied to the policy repository from a Policy Management Tool (PMT), Policy Console and Editing System (PCES) or simply from a policy definition file. The policy repository checks and loads the policies and stores them using a suitable policy model. The relevant policies are deployed to the elements on request.

Service Registry

This is an infrastructure node that is used to locate all elements in the community, including the Mayor and Repository components. It offers a 'white page' and 'yellow page' service for locating community members by name and/or by the service they provide.

Autonomous Element

Each autonomous element consists of an autonomic manager and an application part or managed element (ME). The manager monitors the underlying ME and guides it using policies. It also communicates with other managers to implement global policies.

3.4.2 Policy Handling

Policy Modelling and Policy Specification

Element Roles

It is important to allocate roles to the elements in a manner that provides a flexible assignment of policies. A simple mechanism is used based on a role hierarchy where each element is assigned a role at the leaf element of the hierarchy. The element's role is the full path to its position in the hierarchy. For example, an element in the *R8* branch has a role *R1/R3/R8*.

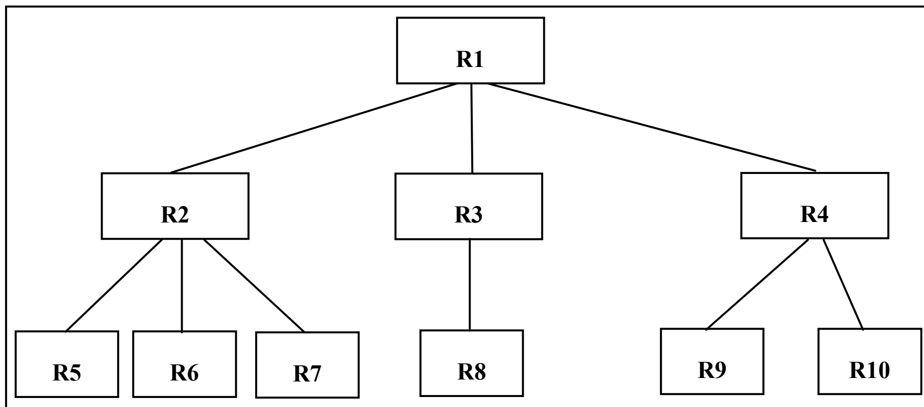

Fig. 8: Role Hierarchy

A subject list of up to five roles can be specified in a policy. For example, policy *GP1* can have a subject list of *R1/R2/R5* and *R1/R4*. This is more flexible than just specifying base roles for each element and less complex than implementations such as Ponder domain expressions [10]. (PH-7)

Policy Schema

The following diagram shows the model for the policy types.

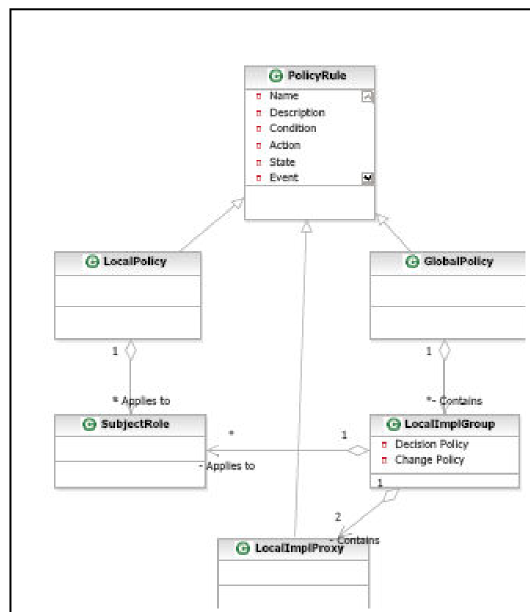

Fig. 9: Policy Schema

A global policy must have a local implementation, that is, policies that are enforced locally to fulfil the global objective. A number of local implementations can be specified in Local Implementation Groups, each with its own subject list. This allows for different local implementations depending on the element's role (PH-2). Each implementation group contains a decision policy and a change policy that is enforced locally. (PH-6)

There are two main policy types, local and global. The global policy has two implementations, decision and change. The local, global and global implementation policies are all executable policy rules in their own right and consist of the following attributes:

- *Event*: Triggering event.
- *Condition*: Boolean condition specified in script language format
- *Action*: Action specified in script language format
- *Exception*: Action specified in script language format that is executed when the normal action execution fails
- *Priority*: Assigns a weighting, or level of importance, used to decide the order of execution when multiple policies are triggered for the same event. Priority 1 is highest.
- *Subject List*: This contains the roles of the elements that will interpret the policy. Up to five roles can be specified for flexible element association.

A Rawlsian approach [46] is adopted to ensure autonomy in that an action cannot be performed on an element that was not involved in the decision making (GE-1).

Comparison to Standard Policy Specifications

The global policy and local implementations are different to the PBM concept of high-level and low-level policies [3]. While the global policy consists of local (implementation) policies, the global policy is not abstract, unlike a high-level policy, and is capable of been executed without refinement.

This policy format and schema was designed to use suitable, but simplified ideas, from Ponder [10], DEN-ng [7] and PMAC [9] and all policies are positive obligation policies as defined by Sloman [5].

This schema differs from the IETF and DTMF PCIM(e) schemas [17] and from the TMF's SID [8] in two main respects.

- Events, conditions and actions were not modelled using separate classes, so it is not possible to create policies from reusable components, but this is not important for the prototype in any case. Only one non-parameterised event is supported.

- The global policy specification required a unique structure that is not found in the standard policy schemas, that is PCIM [16] and DEN-ng [8]. A special schema was designed loosely based on the *PolicySet*, *PolicyRule* and *PolicyGroup* classes from the standards, but not conforming to the standards.

Policy Deployment Model

Specifying Policies

Policies are specified using XML according to the policy schema (XSD) shown in the appendix. The application is responsible for implementing the appropriate local and global decisions and actions. (PH-5)

Loading Policies

The policy loader loads all XML policy files found in the default policy directory, validates them against their XML schemas and populates the policy schema in the repository (PH-1). For simplicity a persistent repository based on an LDAP directory or relational database was not implemented, although an LDAP schema would be most suitable as it is optimised for read access. (PH-4)

Policy Distribution

The relevant policies are downloaded to the autonomic managers on request and stored locally for efficiency. To preserve autonomy it is important that elements only see policies that they can trigger and implement. The element's role specifies a position in the role hierarchy so that policies can be associated with the role easily and flexibly (PH-5). As a global policy may have a number of implementation groups for different roles an aggregated subject list is constructed from the implementation groups subject lists and associated with the global policy for efficiency reasons (PH-7). This is sent to the element with the global policy so that they know what other roles are affected without knowing the polices that these other roles enforce.

The fetched policies must be prepared for local use. This includes adapting them to the local script language, parsing the policies to ensure they can be executed by the local script language and sorting them by priority.

3.4.3 Shared Information and Community Handling

A 'community', as used here, can be defined as a group of elements who share common resources and who are subject to global policies. Elements play roles in a community and must meet minimum capabilities before being admitted as members, such as being policy enabled or implementing the interaction protocol. This use of community is similar to that defined in RM-ODP [47].

It is necessary to store certain information at the 'community' level (CH-1):

- Information about members, their role and capabilities
- How to locate members based on their function.

The community model could also include composition constraints but this is not pursued here

Each autonomic manager must store the following locally:

- A resource model for the element information it manages as part of the element.
- Relevant policies.
- Its own role and capabilities
- Information relating to the current problem state.

3.4.4 Distributed Coordination

Coordination in general is concerned managing the dependencies between activities. Before deciding on the coordination mechanism it was necessary to consider the various types of interactions and dependencies that may exist between the elements concerning shared resources.

Coordination Requirements

Two broad groups were identified from a number of dependency types between configurable attributes:

I. An attribute value in one element is dependent on an attribute value in another subject to some constraint expression, e.g. element1.X must be > element2.Y or Z must be equal in all elements in the role or Q must be unique within an element role.

II. A global resource needs to be allocated among the resources according to the marginal utility they derive from it, according to some local need or agreement, or an attribute value must be negotiated between a number of services.

The first group concerns constraints on setting a value locally while the second group is concerned with setting values based on global decisions. The first group can be satisfied by providing read access to other elements values in order to check the constraints. The second group is more interesting in that attribute settings are based on some group decision. As the nodes are autonomous, and can reject any proposed changes, the policies used to configure the second group are what I term 'weak' policies.

Coordination Mechanism

The coordination mechanism must support the group II dependencies outlined above between any subset of elements in the community based on their role.

There are many approaches for providing this type of coordination in a decentralised environment, with the two main approaches being the use of a shared space or blackboard for sharing problem state and the use of messages to convey problem state. The message approach is more suitable and a direct message type of interaction was employed as opposed to a more loosely coupled one based on events.

This leads to a leader-type protocol that is structured like a two-phase-commit (2PC), which gathers local decisions and sends out change suggestions (GE-3) to configure resource attributes. The difference between the protocol and a traditional 2PC is that an element will not necessarily receive the change message and no state is held or resources locked from the first to the second message.

A generic set of local options were defined to cover the types of 'weak' policies previously mentioned. These options are carried as local decision responses:

- Utility Decision: Where the marginal utility for one extra resource and one less resource is returned
- Value Decision: Where a resource setting is returned.
- Agreement Decision: Where a yes/no response is returned.

The global decision logic, which is application dependent, must be able to cater for the relevant local options or mixture of these responses (GE-1) from the elements. Each element uses the

local implementation of the global policy and its own specific actions to determine the local options based on the policy and local hardwired constraints.

3.4.5 Policy Execution

Each autonomic manager can play two roles, a coordinator role when it detects an event for a global policy (GE-2) and a local manager, who responds to coordinator requests. At any time an autonomic manager can play one or both of these roles for one or more concurrent policy executions.

Concurrent Execution

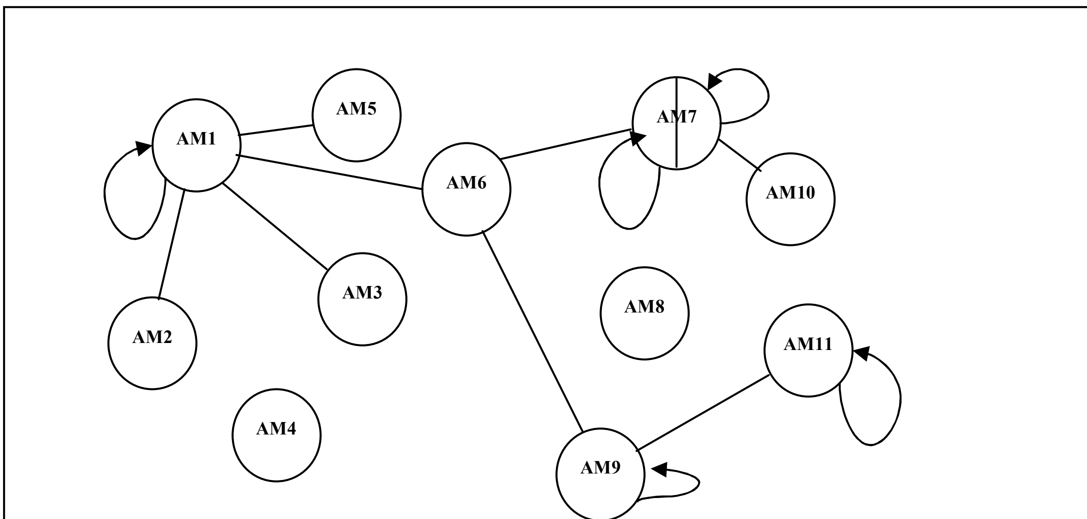

Fig. 10: Concurrent Global Executions

This diagram shows the possibilities that can occur when multiple global policies are executing concurrently. The shaded circles represent coordinator autonomic managers. Note that an autonomic manager can be involved in a number of concurrent executions as a coordinator for one policy and as a local manager for another. Normal local policies can be executing on the elements at the same time. The line through AM7 indicates that this is the coordinator for two different global policy executions, one with AM6 and the other with AM10. A coordination instance is created for execution of each global policy. AM6 is a local manager for 3 concurrent global executions coordinated by AM1, AM9 and AM7.

The subject list specifies the roles that are affected by the policy and these can be defined for overlapping domains.

Duplicate Policy Handling

The application logic must ensure that an event that triggers a global policy is ignored if it has triggered it within a certain period (GE-4). This is to allow the effects of a policy to take place and prevents flapping. However, a particular event may be triggered in more than one autonomic manager at the same time. The same global policy cannot be executing in the system more than once at any time and each local manager and coordinator ensures that it only processes the 'correct' policy and rejects the duplicate policy. A simple mechanism is used here to determine the 'correct' policy. Each coordinator is given a unique system clock or tick value by the Mayor when it retrieves the members from the community store. The rule is to select the coordinator that has already performed most of the work, (that is, has sent out all its requests or has received responses) over one that has performed less work. If they are both at the same stage then the coordinator with the lowest tick value wins. The local manager will process the 'correct' policy executing the local decision policy and return the results to that coordinator. Duplicate policies are not processed and a negative status is returned in the response message to the coordinator. If a request is received at an elements local manager while its coordinator is about to send requests for the same policy then the lowest tick wins and the coordinator instance is removed.

As the autonomic managers can be distributed across a network there is no guarantee that the request messages are received at any local manager in the correct 'tick' order, nor in the same order at each local manager, that is, there are no ordering guarantees. To ensure that a local manager identifies the 'correct' policy it must buffer received messages for a period not less than the period a message with a lower tick can be received after a message with a higher tick (an out-of-sequence period). The buffer messages are then ordered by tick with the earliest tick been processed first. Any duplicates with a higher tick are immediately responded to thus indicating that they are duplicates.

3.5 Main Traffic Cases

A number of traffic cases are provided to show the interaction between the high-level components and the use of the design mechanisms just discussed.

3.5.1 Element Start-up

This traffic case is a combination of two use cases: Join Community and Distribute Policies.

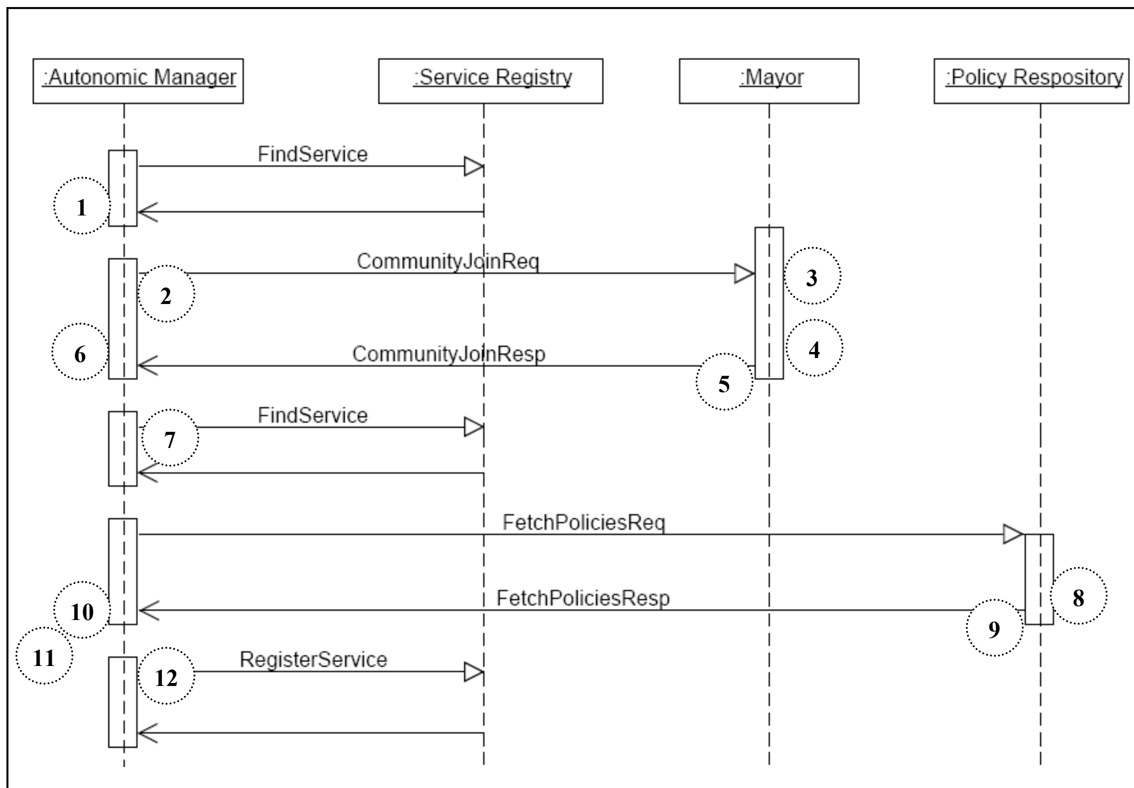

Fig. 11: Element Start-up Sequence Diagram

0. Preconditions:

   - The policies have been checked and loaded into the repository store.
   - The Mayor and Policy Repository services are executing and registered in the service registry
   - The autonomic manager (AM) is starting and has loaded its element information, that is, its identity, service provided, role and capabilities.

1. Upon starting the element locates the finds the Mayor service for the community from the Service Registry

2. The AM sends a request: `CommunityJoinReq(ServiceID, Name, Role, Capabilities).`

3. The Mayor checks if the element has already joined and ensures that the capabilities are adequate for joining, for example, does it support the interaction protocol.

4. If the element is allowed to join then the Mayor adds the element to the member table in the community store.

5. The Mayor sends a response message to the element: `JoinResp(Status).`

6. The element locates the community's policy repository service from the Service Registry

7. The element requests the relevant policies using: `PoliciesReq(Role)`.

8. The repository fetches all local (and local implementations of global policies) as well as the associated global policy where the member's role matches any policy subject role.

9. The repository returns the relevant policies `PoliciesResp(Status, Policies)`.

10. The element installs the policies, by converting them to the local script language, parsing them and sorting them by priority before storing them locally based on the policy schema.

11. The AM starts the Policy Handler, the Event Handler and the application.

12. The AM registers with the Service Registry to become available to other elements for global policy execution.

3.5.2 Global Policy Execution

Note that the Service Registry is queried each time any of the elements or community services needs to be contacted. This is omitted here to simplify the diagram.

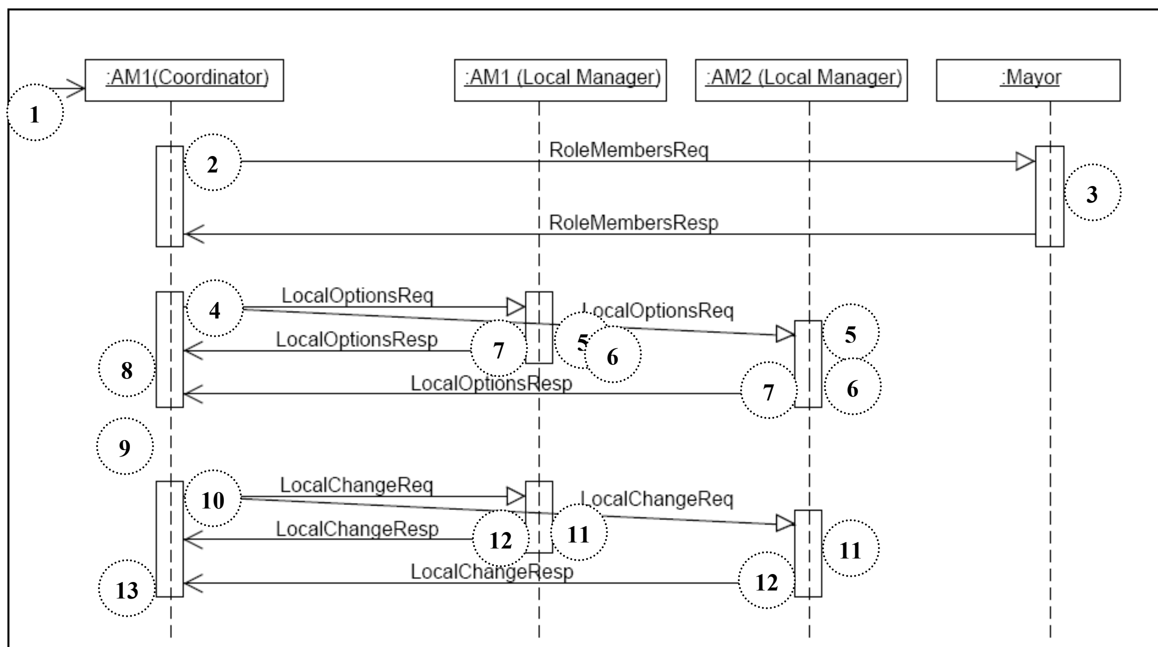

Fig. 12: Global Policy Execution Sequence Diagram

0. Preconditions:

   - Elements are fully started and registered as community members in the community store and service registry.

   - Policies are distributed and installed in the elements.

- An element manager or managed element triggers an event after checking that the policy has not been triggered within a settle period.

1. The event is received by the Policy Handler who looks up the relevant policies for the event. If the policy is a local policy then it executes as normal. If it is a global policy and the policy condition is true then an acting coordinator role is created to coordinate this policy execution.

2. The AM contacts the Mayor and requests all the current elements for the roles in the aggregated global subject list. `GetRoleServices(SubjectList).`

3. The Mayor finds the elements that have any role in the subject list. It also generates a global sequence number or tick value. These two values are returned in the `GetRoleServicesResp(Members, Tick).`

4. The acting coordinator contacts each of the relevant local managers by sending a *LocalOptionsReq(GlobalPolicyID, Tick).*

5. The local manager buffers the requests and selects the one with the earliest tick. It then checks if the policy has been executed already and which request should be responded to favourably.

6. If the request is to be processed, the local manager finds the local implementation for the Global Policy ID and triggers the decision policy which performs the managed element specific action and generates the local option response. If the local constraints are not met then the status is set accordingly.

7. The local options are sent back to the coordinator in the *LocalOptionsResp(MemberID, Status, GlobalPolicyID, Resource, LocalOptions).*

8. The coordinator gathers all local options. When all responses have been received they are processed by executing the global policy action, which is managed element specific.

9. The global policy action analyses the local responses and determines the appropriate local change for each local manager according to the best global benefit or rule. Note that the result could be no local changes or changes for only some of the local managers.

10. The coordinator sends the changes to the relevant local manager: *LocalChangeReq(GlobalPolicyID, Resource, Setting).*

11. Local Managers receive this message, find the associated local change policy for the global policy, check the policy constraint and executes the action using the setting.

12. A response is returned to the coordinator indicating if the change was possible `LocalChangeResp(GlobalPolicyID, Member, Status)`.

13. The coordinator for this global policy execution terminates.

Note that it is assumed (GE-6) that some two-phase commit is used during the change part so as to coordinate the changes across all affected services. This would involve logging the change locally or making the change to a temporary area before sending back the `LocalChangeResp`. The coordinator checks the responses. If all respond positively in the required time then a commit message is sent, for example `LocalChangeCommitReq(GlobalPolicyID)`, otherwise and abort message is sent, for example, `LocalChangeAbortReq(GlobalPolicyID)`. The Local Managers commit or rollback their local change and send an acknowledgement to the coordinator. They then clear the local state for this Global policy as it is competed.

3.6 Design Issues and Decisions

The main issues and decisions that arose during the design stage for the different functional areas are summarised as follows:

- Policy Handling: It would have been useful to use a policy language, such as, Ponder, PDL or ACEL and more importantly a standard policy schema, such as PCIM(e) or DEN-ng and several issues arose in trying to adapt the standard schemas to the policy model required by this system. A compromise was reached by developing a simple policy specification method that uses good parts from policy languages and a schema that is loosely based on DEN-ng.

- Policy Distribution: The main decision made here was which policies (global and/or local implementations) should be downloaded and at what point to download them so as to achieve the right balance between autonomy and efficiency during execution.

- Shared Information and Community: There were many issues here concerning the type of information that needs to be shared and whether a high-level generic resource model should be maintained here for global resources without turning the Mayor into a centralised PDP. The decentralisation of the member-role service was also considered.

- Coordination Requirements: It was necessary to develop a simple model for the types of dependencies between the elements. This raised many issues regarding the scope of the design, the difference between a centralised and decentralised policy approach and how these dependencies can be catered for in a generic way. A resource dependency model

was considered at one point to capture these dependencies in a general way, but this is more suitable to a centralised system.

- Coordination Mechanism: The issues here cover how best to provide the coordination interactions. Should they be part of the policy language, embedded in the element, downloaded to the element on admission or built into the element and declared as interaction capabilities for admission. Another option considered was to extend the 'policy language' to include coordination functions, such as optimise, agree, negotiate. I also considered adopting the role-relationship briefly referred to in the Ponder specification [10] or the use of roles and relationships in conversation patterns as defined by Stergiou [48]. Also considered using a hierarchy of events (compound and filterable) as a way of coordinating policy triggering in that the global policy triggers events that trigger local policies. I felt that this would be very difficult to control and would be more of a command hierarchy approach than consensus based, and therefore difficult to aggregate local decisions to make higher decision based on agreement.

- Coordinator Role: The main issue here was how to select a coordinator. Election methods were considered, which are robust but incur high overhead, especially here where delays are important.

3.7 Summary/Conclusions

This chapter proposed a general solution based on distributed elements and three central services. A number of policy-related and coordination mechanisms were developed to satisfy the system requirements. Traffic cases were presented to show how the mechanisms fit together to provide an overall solution.

4 Prototype Implementation

4.1 Introduction

This chapter describes the prototype in terms of its structure, main components and the technologies selected for its development. The scenario outlined in the previous chapter is used to illustrate how the generic solution can be adapted by application specific elements to support distributed coordination. The last section discusses some of the many implementation decisions.

4.2 Technical Architecture

4.2.1 System Overview

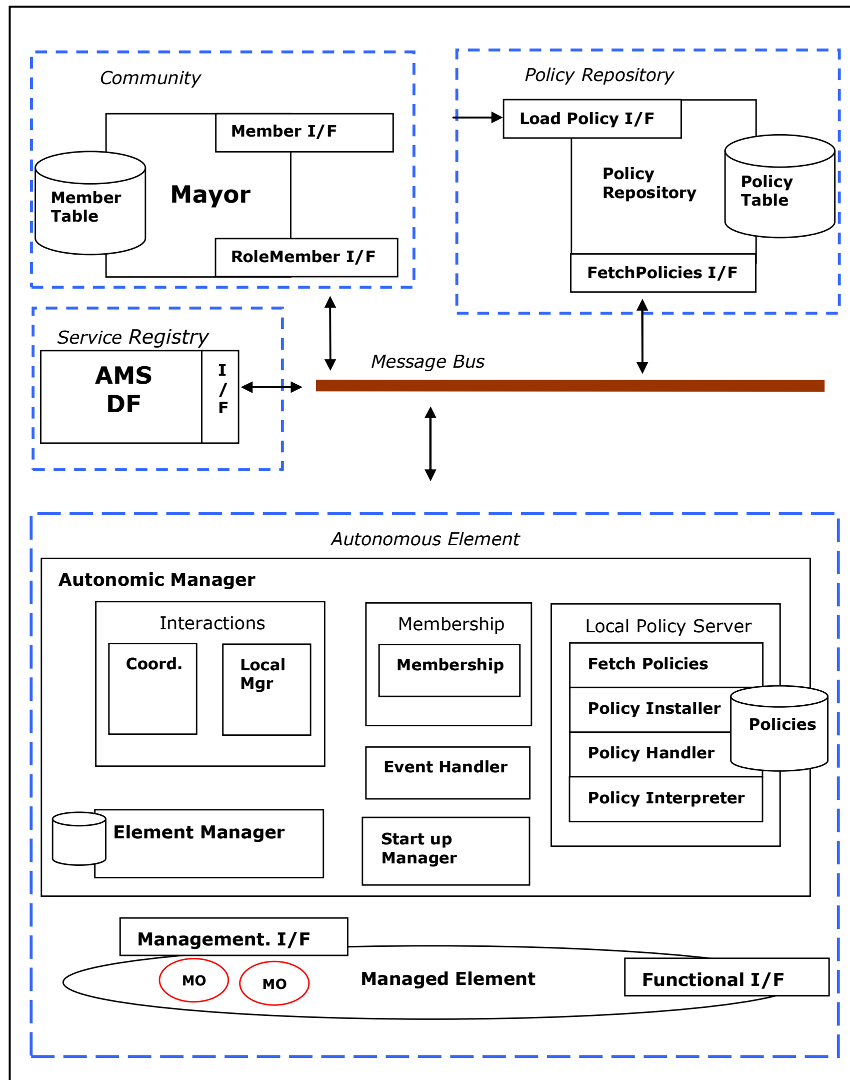

Fig. 13: System Architecture

The system consists of four main parts, or sub-systems, interconnected by a messaging bus. The components can be deployed across different hosts or on the same host machine. An agent-based approach was adopted, based on the Foundation for Physical Intelligent Agents (FIPA) specifications [49] in order to provide a more open environment to facilitate autonomic manager interaction and interoperability. The Java Agent Development Environment (JADE), which is considered the 'de facto' reference implementation of the FIPA standard [50], provides the middleware for the development of distributed multi-agent applications based on the peer-to-peer communication architecture. JADE is further discussed in the technology section.

4.2.2 Component Description

Service Registry

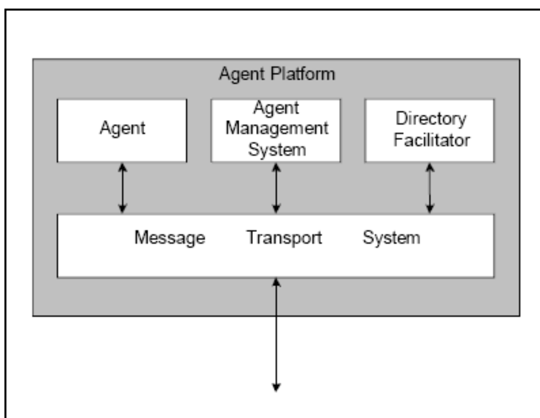

Fig. 14: Reference architecture of the FIPA Agent Platform

The Agent Management System (AMS) and Directory Facilitator (DF) are two agents provided by JADE and conform to the reference architecture of a FIPA Agent Platform. An agent platform consists of several application agents monitored and controlled by one AMS and can be distributed across several host machines. Agents automatically register with the AMS on start-up and receive an Agent ID (AID). The AMS provides two important services: a 'white pages' service for agent location, and an agent monitoring and control service [51].

The DF provides an optional 'yellow pages' service within the platform. Agents can publish one or more services by specifying the service name and service type with the DF. Other options, such as ontologies, protocols and languages supported by the agent and its services can also be registered so that other agents can discover and interact with the appropriate agent services. Several DFs may exist in a federation to provide multi-domain applications.

Message Bus

The Message Bus used for inter-agent communication, is provided by JADE's Message Transport System (MTS), also known as the Agent Communication Channel (ACC). The MTS controls the exchange of messages within the platform and between platforms. MTS uses three types of communication, depending on whether the agents exist in the same container (Java event mechanisms), different containers in the same platform (RMI) or on different JADE platforms (IIOP or HTTP). One JADE platform is implemented in the prototype architecture and includes a number of containers. The containers can be distributed across several hosts.

Community

The Mayor is modelled as a FIPA agent and implements the *Member* interface, for member admission, and the *RoleMember* interface, for finding members within the role hierarchy.

Policy Repository

The Policy Repository agent implements the *LoadPolicy* interface. This is responsible for validating and loading policies, specified as XML files in the policy directory, into a policy table based on the policy schema. The agent's *FetchPolicies* interface returns relevant polices in a policy table to the requesting autonomic managers.

Autonomous Element

This component consists of the autonomic manager and managed element. A number of elements will exist in the community, each providing an application specific service (provided by the managed element) and managed by the, mostly generic, autonomic manager. The autonomic manager will usually require a specific element manager that monitors and controls the managed element according to some control loops or MAPE loop [27]:

- Monitor: Fetch performance data from managed element
- Analyse: Determine QoS values and generate events based on these values
- Plan: Select relevant policies
- Execute: Execute the policies to configure the managed elements.

Autonomic Manager

The generic autonomic manager consists of the following components:

Component	**Description**
Element Manager	The functions provided by the element manager are particular to the element being managed. For example, in our scenario this may include a monitoring function that periodically gathers QoS performance data from the managed element and

	calculates relevant statistics. The data is analysed and checked to see if the values are a cause for concern in which case appropriate events are generated. The application specific manager component acts on application specific MOs that represent the resources been configured.
Event Handler	Events are triggered from the application or application specific management component and passed to the event handler which implements a simple blocking queue to supply events to the policy handler.
Membership	For joining the community.
Local Policy Server	The Fetch Policies function downloads policies from the repository on start-up. These are converted to the local script language, checked and sorted on priority by the Policy Installer and stored in a local table for fast access. The PDP function is implemented by the policy handler, which selects relevant policies based on the event it receives. The policy interpreter implements the PEP function by executing the policies selected by the policy handler.
Interactions	An instance of a coordinator is created to coordinate each global policy execution. A local manager continually executes waiting and dealing with decision requests and change requests from coordinators.
Start up Manager	This performs the start-up function. It locates the Mayor, joins the community, fetches its policies, starts the event and policy handlers, starts the managed element and starts the Local Manager before finally registering itself in the DF.

Table 6: Autonomic Manager Components

Managed Element

The managed element resources, such as threads and queues are modelled as MOs and are configured via a management interface, which are essentially the MO interfaces. The managed element also includes a functional interface to provide its main service, such as foreign exchange conversion or mortgage interest calculation. It is not necessary to implement the managed element as an agent and this purely depends on the application. An example managed element is presented later in this chapter, showing how it can be integrated with the autonomic manager.

4.3 Technologies

This section gives a brief overview of the technologies considered, and evaluated, for the implementation. The decision to select certain technologies was based on the following criteria: availability as open source, range of functions and flexibility, maturity, availability of documentation and support, scalability and ease of learning. Some technologies, although

eminently suitable, were not selected because they would have required too much effort for non-core functions, however they are recommended for an 'industrialised' version of the system.

Function	**Technologies Considered**	**Selection**
Agent P2P Layer and Messaging	J2EE, WS-*, JADE, FIPA-OS, Tryllian (JXTA)	JADE
	RMI, CORBA, JMS, Web services	RMI
Service Discovery/Lookup	Jini, JADE DF	JADE DF
Policy Rep. and Handling,	Ponder, XML, JESS, PMAC	XML
Events	JMS, JADE messages or blocking buffer	Blocking buffer
Script language	JRuby, Jython, Groovy	Groovy.
Persistence and Transactions	RDBMS (MySQL) with Hibernate, LDAP (OpenLDAP), filesystem. JTA for transactions.	File system
Libraries and tools	JDOM using Xerces, Log4J, JUnit, Maven.	All used
Programming environment and Platform	J2SE 5 on Dell Latitude D400 running XP	Used

Table 7: Summary of Technologies Considered and Selected

4.3.1 P2P Agent Layer and Messaging

A number of technologies were considered for providing the 'agent layer'. An approach based on the agent paradigm was adopted because the autonomic managers and their environment meet the main criteria for agents, that is, they are autonomous, proactive and social. Agent development environments simplify the interaction between components and are flexible in terms of open protocols and support for heterogeneity using ontologies. Services share many of the features of agents in that they are autonomous and heterogeneous and agents provide a suitable model for services [36].

ADE was most suitable agent development framework considered for the following reasons: it fully implements the FIPA standard, provides many features that simplify agent development and communication, is widely used and supported within the MAS community, is scalable and implements flexible and efficient messaging as outlined in [52]. Other agent platforms, found in [53] such as, FIPA-OS and Tryllian, based on JXTA, were also considered.

One negative aspect of using JADE is that the FIPA standards have not developed over the past two years and it was felt that new agent platforms will be based on WS-standards, such as Web Services Conversation Language (WSCL) and WS-Agreement. However, FIPA has recognized

this and has set up an Agents and Web Services Interoperability Working Group (AWSI) to "*fill the communication gap between agents and web services*". The standards were originally due in Sept-2006, but are currently delayed [54].

In terms of messaging, RMI was selected as it is the default inter-container communication method used in JADE and performs well. The use of web services was initially considered but it is widely accepted that web services perform poorly compared to technologies such as RMI [55]. I also felt that the relevant WS-* standards, such as WS-Agreement, WS-Transaction, WS Conversation Language are neither mature enough or provided adequate tools to use.

4.3.2 Service Discovery and Lookup

Jini™ was considered for the community membership component as it allows dynamic membership and autonomic service discovery (in addition to other distributed computing mechanisms) [56]. Jini™ uses Javaspaces, a technology that provides loosely coupled communication using the distributed shared memory approach. This was considered as a method of coordination between the autonomic managers initially but rejected in favour of a message based approach. However, if the role-member lookup was to be distributed, then the use of Jini™ for dynamic service discovery and Javaspaces for sharing the role-member relationship would provide a good solution.

JADE provides a more traditional naming and directory services lookup via the AMS and DF. This was supplemented by a higher level, or application level, lookup based on roles and community members. It was not possible to incorporate this into the DF and so it had to be provided as a separate function in the community.

4.3.3 Policy Implementation

The IBM Policy Management for Autonomic Computing (PMAC) [57] provides an infrastructure for managing autonomic environments using policy. It provides a ready made policy language (ACPL) and deployment environment. It also provides a Policy Analysis Toolkit that can be used to identify conflicting policies and an expression language (ACEL) for embedding expression in the policy specification. Unfortunately it was not flexible enough to support the schema necessary for the distributed policies. Also, its useful functions, such as its policy language and toolkit were not central enough to the implementation to warrant the extra work necessary to use them.

JESS is a java based rule engine that can be used by agents to "*reason declaratively using knowledge represented in rules and facts*" [36] . Jess is sometimes used as a policy language but

I did not find it suitable for use here, despite the fact that it is often used with JADE as a reasoning mechanism for agents. The Ponder toolkit is no longer available [58] and not obviously adaptable to the policy schema. While it would provide a very flexible specification language it was too 'heavy weight' to be used in the prototype.

Once it became apparent that a system wide event handler was not necessary to distribute events to the autonomic managers and that only one event receiver (policy handler) internal to the element was needed, a simple event communication based on a blocking queue was implemented rather than a publish-subscribe approach, such as provided by a JMS implementation.

XML was used to represent the policies. The condition, action and exceptions were specified in the Groovy scripting language [59], which was selected for its flexibility and ease of use from within Java. A custom built Java based policy handler and policy interpreter were developed.

4.3.4 Persistence and Transactions

Persistence storage was not implemented as it was not a core part of the problem. If it was to be added then the best solution would be LDAP for policy repository (OpenLDAP), due to its efficient read access and an RDBMS (MySQL) for community and local autonomic manager data implemented with Hibernate to aid flexible development. The DB can also be used to implement transactions. A more heavy weight transaction handler, JTA was considered for implementing the distributed transactions. This is modelled on the X/Open XA architecture and implements 2PC but it was too awkward to implement with the JADE approach. A better solution would be to introduce an extra round of messages to implement a 2PC, as suggested in the design. This approach has already been developed using FIPA standards and JADE by Lockeman [60]

4.3.5 Host, Libraries, Tools

JDOM was used to validate and parse the XML policies, Log4j for logging, Maven for build, JUnit for certain tests. The Java J2SE 5 environment on Windows on Dell Latitude D400 was used for all development.

4.4 Agent and Behaviour Implementation

4.4.1 Agent and Behaviour Design

An agent based approach was taken based on the FIPA standards for intelligent agents [49] and [50]. The Gaia methodology [61] and agent design methodology used in [62] were referenced to determine the roles (agents), interaction model (protocols) and behaviours. Behaviours implement the main agent tasks and are used primarily in this system as state machines for handling interactions between agents.

Agent	Behaviour	Protocol
Mayor	MembershipImpl	COMMUNITYJOINREQ / RESP
Mayor	RoleMemberImpl	ROLEMEMBERSREQ / RESP
PolicyRepository	FetchPoliciesImpl	FETCHPOLICIESREQ / RESP
Autonomic Manager	CommunityMembership	COMMUNITYJOINREQ / RESP
Autonomic Manager	ReqPolicies	FETCHPOLICIESREQ / RESP
Autonomic Manager	StartupManager	
Autonomic Manager	CoordGlobalPolicyImpl	LOCALOPTIONSREQ / RESP LOCALCHANGEREQ / RESP
Autonomic Manager	ReqRoleMembers	ROLEMEMBERSREQ / RESP
Autonomic Manager	LocalOptionsImpl	LOCALOPTIONSREQ / RESP
Autonomic Manager	LocalChangesImpl	LOCALCHANGEREQ / RESP

Table 8: Agent and Behaviour Model

4.4.2 Implementing Concurrency

Concurrency was an important issue in the implementation and the concurrency requirements of the design, as well as JADE’s implementation of concurrency, required careful consideration in order to cater for the different ways the agents can be deployed, (that is, on same or different containers and on the same or different hosts). JADE uses three levels of concurrency [52]:

1. Container level: Containers can be distributed on the same host or on different hosts. Each container has its own independent JVM.

2. Agent level: Each container holds one or more agents, where each agent executes within its own thread.

3. Behaviour level: The agent supports a form of concurrency by implementing agent behaviours.

A round-robin non-pre-emptive scheduling policy is used by the agent scheduler to give each queued behaviour a turn in executing within the agent thread, as described in [51]. The actual function, or amount of work, that is performed by a behaviour before it yields the thread to the next behaviour in the queue (by returning from its `action()` method) depends on the implementation of the state machine for the behaviour. This requires careful design to achieve the right balance between sharing processing resources and coordinating the interactions. For example, an instance of the *CoordGlobalPolicyImpl* behaviour is created for each concurrently executing global policy within the agent. This behaviour has four main processing stages. Each stage is executed without interruption by any of the other behaviours belonging to the agent. The agent thread is then released and the behaviour blocks itself while waiting for the next message in order to proceed with the next stage. The use of behaviours is quite efficient as no time-consuming context switching is necessary when switching between behaviours. However, normal thread context switching occurs between the threads of different agents on the same host.

It was necessary to coordinate the behaviours using JADE's higher-level sequence and parallel behaviours. However, these techniques were not always adequate, nor flexible, and it was necessary to employ other techniques for behaviour coordination and for sharing information between behaviours, which JADE is not so good at.

Each element requires at least four threads: one for the autonomic manager agent, one for the event handler, one for the policy handler and at least one for the element manager, depending on its application. Normal synchronisation techniques were employed to prevent thread interference.

4.4.3 Implementing Conversations

Designing the Interaction Protocol

FIPA specifies 22 message types for agent interaction, called communicative acts or performatives, for example, REQUEST, REPLY, INFORM [63]. These acts can be combined into interaction protocols which are used as templates for agent conversations. JADE has implementations for the FIPA standard interaction protocols, such as Request Interaction, Subscribe Interaction and Contract-Net Interaction protocols.

The conversation protocols used for agent interaction in this system are built on top of the FIPA-Request Interaction Protocol [64]. The request messages, such as *LocalOptionsReq*, use

the REQUEST performative, while the response messages, such as *LocalOptionsResp* use the INFORM performative. All messages are passed using the FIPA Agent Communication Language (ACL). ACL messages have a standard structure [65] that includes the performative, sender, receiver(s), conversation id, protocols, in-response to, reply by time and ontology as well as the message content, which can be text, but in most cases, serialisable objects. A standard serialisable class called MessageObject was defined in the prototype as a base class for all objects passed in messages.

Message and Conversation Handling

For efficiency purposes it is possible to block a behaviour's `action()` method from being processed by the agent's scheduler until some message arrives or a time period has passed. However, when any message is received by an agent all behaviour `action()` methods are woken. It was therefore necessary for each agent's behaviour, and behaviour instance, to filter the received messages so that only relevant messages are processed. The filtering mechanism used in this prototype was based on the protocol (e.g. *LocalOptionsResp*) and a unique conversation ID.

The conversation ID was generated and attached to each request message that was sent out as part of the conversation, for example in the *LocalOptionsReq*. This is used as a filter in the corresponding 'receive' messages, such as *LocalOptionsResp,* to ensure that the correct behaviour instance reads the relevant responses. JADE has provision within its ACL message for a conversation ID, although its generation is application dependent.

JADE Agents communicate using asynchronous messaging. Local Managers may receive *LocalOptionReq* messages from different coordinators for the same global policy out-of-sequence, in terms of tick order. To ensure all earlier requests for the same policy have been received, so that the correct one can be processed, it is necessary to implement a buffer at the local manager, as outlined in the design chapter.

4.5 Integrating the Managed Element

This section provides an example of how the generic autonomic manager and conversation protocol can be used for an element from the scenario outlined in the design chapter.

4.5.1 Implementing the Scenario

The Interest Calculator managed element gathers instrumentation parameters such as, total requests received per interval, total requests dropped and time request queued while processing

user requests. These raw values are periodically gathered by its element manager, and summary statistics are produced for the monitoring period, including values such as, average requests received per-second, average requests processed per second, average queue length and average recent thread usage.

The element manager analyses the statistics and triggers an event to instigate a local or global policy, provided of course that sufficient time has passed to allow the system to settle since the policy's last execution.

The relevant MOs used in the condition and action part of the policies must be available to the script environment and are registered by the application (element manager and/or managed element). Functions must be provided for local and global reasoning. In this example some resource specific optimise function must be provided to determine local utility and determine global settings. For example, the thread MO of the Interest Calculator service has a local utility calculator that determines the marginal utility for one extra resource and one less resource using the formula below, where WQ (average queue wait time), LQ (expected queue size) and U (utilisation) are determined using an M/M/c queuing model [66] based on average requests per second, number of threads and average processed per second.

$$Utility = (1/WQ + ((1/LQ) * 100) + (100 * U)) - (T * ((10 - T) * 100))$$

Fig. 15: Utility Function for Thread Resource

This utility function, although of little value in itself, was implemented to illustrate how utility type policies can be supported and to introduce computationally intensive queuing calculations in order to determine the reaction time of the policies.

Other services such as the Foreign Exchange Converter can have its own utility calculator, provided it adheres to the MO interface, that is, it returns a *LocalUtilityResponse* object that can be processed by the global optimisation function, as shown in the example below.

```
public interface ThreadMO {

    :         :             :             :             :
    :         :             :             :             :
    // Local Decision
    public abstract LocalUtilityResp optimise();

    // Global version to determine changes.
    public abstract LocalChangeList optimise(List<LocalUtilityResp> responses);

    // Local decision
    public abstract LocalBooleanResp oktoSetPriority(int newPriority);

    // Global decision
    public abstract LocalChangeList setPriority(List<LocalBooleanResp> responses,
            int priorityLevel, int quorum);

    // Local Decision
    public abstract LocalValueResp getInitialDemand(int avgThreadUsage);

    // Global decision
    public abstract LocalChangeList initialSet(List<LocalValueResp> responses,
            int allocatePcnt, int proportion, int maxAlloc);
}
```

Fig. 16: Extract from Example Thread MO Interface

The decision and action methods for the local and global policies are implemented in the resource MO in this example. These could be implemented anywhere as long as they are available to the policy via the scripting language, as shown in the policy extracts below. The MO however must provide a set method for setting the resource attributes from the policy, not shown above.

```
:                    :                    :
:                    :                    :

<event>ON LowThroughput</event>
<condition>IF statsMO.avgReqReceivedSec >1 && statsMO.avgPcntProcessedSec < 10
</condition>
<action>
           THEN threadMO.optimise(responses)
</action>
:                    :                    :
:                    :                    :
<event>ON DetermineThreads</event>
<condition>
       IF threadMO.maxThreads <= threadMO.threadLimit && threadMO.maxThreads > 0
</condition>
<action>
       THEN threadMO.optimise()
</action>
:                    :                    :
:                    :                    :
<event>ON UpdateThreads</event>
<condition>
       IF threadMO.maxThreads <= threadMO.threadLimit
</condition>
<action>
       THEN threadMO.setMaxThreads(newvalue);
            println "Updated Value: " + threadMO.getMaxThreads()
```

Fig. 17: Extracts from an Example Optimisation Policy

Note that another set of elements, for example the Database service, may have a different type of local policy and this would be specified in another implementation group within the global policy specification.

Further policy examples can be found in the appendix.

The following diagram shows the current set of decision objects that can be used and mixed in any policy decision:

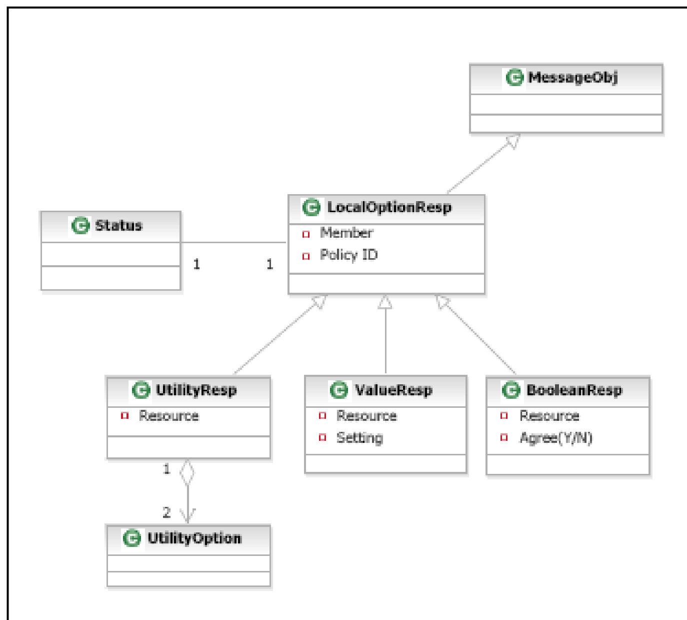

Fig. 18: Response Types for Local Decisions

4.5.2 Integrating a new Element

A number of steps are required to develop a new element.

1. Create the element's autonomic manager based on the generic autonomic manager class. This provides the autonomic manager component as outlined previously. The specific element manger must be implemented. The elements properties (service name, type, role and capabilities) must be specified in a properties file, which is loaded and used by the autonomic manager.

2. Specify the application specific managed element and its MOs

3. Implement application specific functions, based on the MO interface, which can be invoked from the policies.

4. Register all MOs that can be referenced in the policy with the script environment.

4.6 Implementation Issues and Decisions

The following list highlights some of the many implementation decisions and issues encountered and not already discussed above:

- What FIPA performatives and interaction protocols to employ? A protocol based on the basic Request-Interaction protocol was developed.

- Designing and coordinating behaviours. How much work should a behaviour do within an `action()` call, what is its FSM and how to share information between behaviours.

- Whether to employ the Mayor as a higher-level 'yellow pages' service to provide members for roles or to use the DF for this. It was not possible to use the DF for this without extending certain DF classes as the DF has no place for role.

- The 'tick' was selected as an arbitration mechanism as it was unique and it was necessary to go to the Mayor anyway to find the services based on roles. An alternative to the 'tick' would be to use the conversation id which is also unique.

- The implementation of the event handler and policy handler in their own threads but not as internal agents. The handlers needed their own threads to handle concurrent triggering and policy execution and the agent model would not have suited their purpose.

- Whether to implement the settle time check in the application specific part (element manager or managed element), in the policy or as part of the event or policy handlers. Decided to include this check in the element manager as this is would be an analysis function for the autonomic manager (based on the MAPE loop).

- Handling transactions and resource locking. There is an issue regarding resource access when several global policies are executing at once. The resource values used in a decision may not be the same before a change request arrives. This is a well discussed conflict problem in the policy world and difficult to identify and solve. The solution commonly used is to lock the resources between the messages, however, unlike a normal 2PC there may not be a change request message so some timeout value would need to be employed on the lock. Decided not to implement any solutions here as it is not core to the problem and many solutions exist.

- Whether to implement the delay to ensure out-of-sequence messages are ordered. There is a trade off here between processing the first out-of-sequence request for a duplicate policy versus the delay incurred.

4.7 Summary/Conclusions

This section has provided a brief description of the system components and the technologies considered and used in its construction. It also discusses some of the main implementation considerations in terms of using JADE. An example is used to show how the solution can be integrated into application specific elements. Finally, some implementation decisions and issues were listed in order to better explain some of the decisions taken.

5 Evaluation

5.1 Introduction

This main dissertation question is to determine the interactions and shared knowledge that is needed for the autonomic elements to coordinate in order to fulfil a policy objective. A number of sub-questions were raised based on this, regarding policy representation and handling, shared knowledge and coordination. These sub-questions were then refined into a set of functional requirements and less important non-functional requirements. This chapter uses test results and critical analysis to determine the extent to which the key requirements have been met and the strengths and weaknesses of the design and implementation mechanisms.

5.2 Evaluation Criteria

In addition to evaluating the solution based on the degree to which the requirements are satisfied, it is also evaluated using relevant criteria from Autonomic Computing (AC), Policy-Based Management (PBM), Multi-Agent Systems (MAS) and general distributed computing.

For example, a number of metrics have been defined for evaluating autonomic systems [67]. Most of these metrics relate to the autonomic application and are therefore of concern only to the element manager and managed element. However, the *adaptation time* metric (from event to self-adjustment) and overall *stabilisation time* (defined as event-interpret-adjust-steady state) are important metrics that are dependent on the performance of the global policy execution.

PBM solutions have specific requirements regarding response (provisioning) time, policy specification language flexibility and conflict handling, as outlined in [41] [68]and [7], and can be evaluated with respect to these.

The multi-agent systems literature suggests that multi-agent systems should be evaluated in terms of coherence (how well the system behaves as a unit) and coordination (level of conflict between the agents) [69]. Specific criteria, such as guaranteed agreement, distribution, symmetry and efficiency are also relevant, while agent interaction criteria such as, Pareto efficiency and maximisation of social welfare are of concern to actual policies and optimisation functions, and are not of interest here.

While the non-functional requirements are less important in a prototype, it is essential to determine if there are any intrinsic problems that would prevent the system from operating

usefully in a normal environment and which cannot be solved by external solutions, such as extra hardware or fault handling. The non-functional requirements evaluated include performance and scalability, protocol efficiency, adaptability (and how well it can be integrated), dependability and interoperability which are all desirable characteristics of general distributed systems [70].

5.3 Evaluation Tests

A number of tests were performed on the solution using a simple test harness and skeleton applications (element managers and managed elements). All tests were performed using a Dell Latitude D400 laptop with an Intel Pentium™ processor (1.60GHz) , 512MB RAM and running Windows XP.

One local and three different global polices were created. The tests involved triggering one or more policies from one or more agents at the same time or with a delay. The tests were repeated with different numbers of elements (2..10 and 15) in the subject group.

5.4 Findings

The key findings are presented with reference to the fulfilment, or non-fulfilment, of the requirements listed in the design chapter.

5.4.1 Functional Requirements

Policy Handling

A simple method for defining and loading policies (PH-1) was implemented and tested. The policies are specified using local and global XML specifications templates with embedded Groovy scripts. While this method lacks the flexibility of a full specification language as outlined in [71], and therefore omits features such as event and domain specifications, it is flexible in that different local implementations can be specified for a global policy based on different subject roles. A simple but effective mechanism was used to separate the role from the element and to organise elements for easy policy assignment (PH-7). Policy deployment is efficient since only relevant policies (global and local implementations) are deployed and the policies are sent in policy table object based on the policy schema. A mechanism for notifying subjects of policy changes is required in a real system, so that subjects can fetch the relevant changed policies (PH-1).

A problem specific policy model was developed, based on existing standards, but it was not possible to fully comply with these standards (PH-3). For simplicity the model does not separate the policy components (event, condition, action) for reusability as would be required by a normal PCES/PMT [7].

Three 'weak' or consensus type policies were identified from analysing element dependencies and executed to cover the most common resource allocation situations. This model can be easily extended to cater for other types of replies, such as multi-resource utility.

Local and global decisions are supported based on element specific decision logic. These local decisions are subject to the local decision and change policy constraints and constraints specified in the local decision logic (PH-9).

In general, the implementation meets of all of the policy handling requirements, albeit with minimal or simplified techniques. The lack of a suitable standard policy schema is a potential problem but could not be overcome without limiting the policy function.

Shared Information and Community Handling

Element admission control (CH-1) is provided by the Mayor using a simple join mechanism based on member capability fulfilment. Member, or element, state is provided by JADE's AMS.

Three services are used to provide provides location transparency (CH-2): A high-level 'yellow pages' service based on element role (Mayor), a 'yellow page' service based on element service (DF) and a 'white pages' service based on element name (AMS). This provides a flexible solution (CH-4), but one that is centralised, which, depending on the failure semantics of the application, which may or may not be a problem.

To overcome this single point or failure, the Mayor or, more importantly, its *RoleMember* service can be distributed, so that coordinators can find members in the event of failures. A shared space solution (perhaps based on Javaspaces) could provide this. The JADE environment also allows replication of the DF and AMS for greater fault tolerance. This involves replicating the main container - which contains the AMS, DF, ACC and internal RMI registry - into a master and number of backup containers as outlined in [72]. Another option would be to extend the DF service to cater for member roles, therefore providing the *RoleMember* service, and to use JADE's replication mechanisms to replicate the DF.

Global Policy Execution

The autonomy of each element (GE-1) is respected in a number of ways: no element can directly change another element, no element can be involved in a decision without it possibly

being affected by that decision and each element is free to implement its own local decision constraints and decision making mechanism (provided the interface is respected). In addition, any element can become a coordinator for any global policy it triggers, irrespective of current co-ordinations (GE-2). This may have an impact on load and a number of techniques could be employed to mitigate this, such as load sharing or coordinator election based on load.

A simple interaction protocol (GE-3) is implemented that provides the necessary communication for satisfying the so-called 'weak' policies. However, this is quite restricted and forces elements along a particular conversation path. Different interaction types, such as multi-stage negotiation, would require changes to the interaction protocol. Fortunately, several of these complex interaction types are already specified by the FIPA standards, such as Contract-Net, and provided for in JADE. It is also possible to develop specific interaction protocols using the underlying message performatives.

Unstable states are avoided (GE-4) in two ways: through the completion of the protocol, despite the presence of duplicates and by the implementation of a settle period after resource adjustment, to avoid continuous policy execution or 'flapping'. The element manager checks the settle period for an event before it triggers the event and each local manager keeps track of when a particular policy was last processed to ensure it is outside its settle period. The current implementation of the settle period is too simple and potentially problematic. For example, there can be mismatches between an element's settle period and a local manager's settle period for the same policy, resulting from delay's in message reception at the local manager, and causing a correct policy to be rejected as a duplicate.

Concurrent global policy execution (GE-5) was tested by triggering different policies in different autonomic managers and different policies in the same autonomic manager for one, two and three global polices involving a number of local managers. Simultaneous execution of a local and global policy in an autonomic manager was also tested. Duplicate policy handling was tested by triggering the same policy twice in a particular autonomic manager and in different autonomic managers with different delay times. In each case the protocol completed successfully.

The fulfilment of the ACID properties [70] is assumed (GE-6) as this was not the focus of the dissertation and many solutions already exist to ensure these properties. Solutions are typically based on resource locking and a 2-Phase-Commit protocol to ensure that changes are consistent across the elements. The interaction protocol developed for the dissertation assumes a 2PC message round after the *LocationChangeResp* as outlined in the previous chapter. A good example of such a protocol developed for FIPA based agent interaction in outlined in [60].

There are also several other technologies for ensuring ACID properties including JTA and the use of a database for controlling resource access.

5.4.2 Non-Functional Requirements

Performance Evaluation

Although this is a prototype, and lacks complete code and optimisations found in a more developed system, it is worth looking at the general performance to see if it meets the requirement for reasonable execution time (PF-1) that is vital for autonomic computing and policy-based management solutions.

The system is implemented using one JADE platform with each element deployed in a separate container. Note that each element employs four threads (agent, policy handler, event handler and managed element) and that element behaviours are queued while waiting for the agent thread.

Time readings are taken using the Java's `System.currentTimeMillis()` method which is accurate to 10ms. The buffer period used in the local managers to detect out-of-sequence messages is set to 200ms, unless otherwise stated. There were small variations between the same tests each time they were executed but these variations were not significant to affect the averages and general trends considered here. In each case the tests were performed 10 times, where they were 2 to 10 elements and 15 elements. To ensure consistency between the tests, the necessary application values were defined to ensure that all elements were the subject of the global policy and that they all received a *LocalChangeReq*. This is the maximum participation scenario as all elements in the set are fully involved.

<u>End-to-End (E2E) Response Time</u>

The E2E response time measures the time from when the event was triggered to when the final *LocalChangeResp* message is received by the coordinator. This period is slightly longer than the response (provisioning) time referred to in PBM or adaptation time as used in autonomic computing.

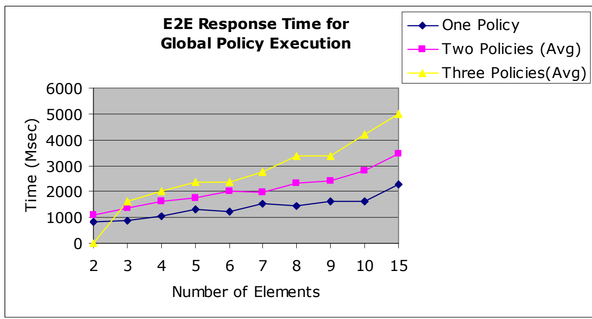

Fig. 19: E2E Response Times for Concurrent Global Policy Executions.

The times taken for concurrent policy executions coordinated by different autonomic managers are shown in the figure. A single policy involving 7 elements takes about 1.5 seconds to complete. This rises to 2 seconds per policy when two policies are executing in the element set and 2.7 seconds per policy when three policies are concurrently executing.

The response time is acceptable for the example scenario, since the optimisation policy is executed well within its 30 second settle period, but will not suit applications involving many more concurrent policy executions or many more elements. Even executing 3 concurrent global policies over 15 elements resulted in a 5 second E2E response time due to the load on the machine. The increase in response time is linear and therefore more controllable using element distribution and more powerful hosts. The policy type (optimisation, value set or agree decision) did not appear to have any significant affect on the overall E2E time despite the extra processing involved in the example optimisation policy.

On average: 6% of the E2E response time involves handling the initial event, finding the policy and creating a coordinator instance; 4% is spent fetching the element names from the Mayor; 52% is spent sending and waiting for local decisions; 30% making the global decision and 7% performing the changes and processing the responses.

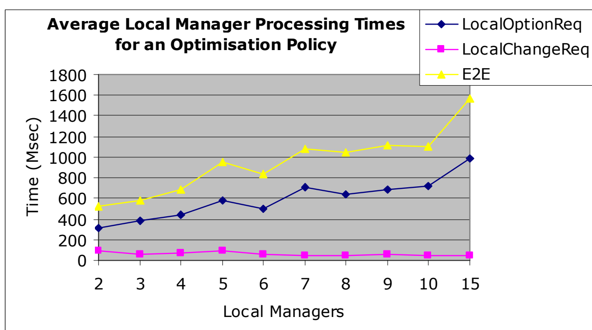

Fig. 20: Average Local Manger Processing Times

The local manager's E2E processing time is incorporated in the overall policy E2E response time discussed above. The local manager's E2E processing time covers the period from

receiving the *LocalOptionsReq* to sending the *LocalChangeResp,* including processing times and the interval between the request messages. Of the actual processing time about 90% is spent processing the *LocalOptionsReq* and 10% processing the *LocalChangeReq*. The *LocalOptionsReq* request processing time is heavily affected by the 200ms delay buffer setting.

The buffer time has an impact on the delay in the local manager, and therefore the overall policy E2E time as shown below.

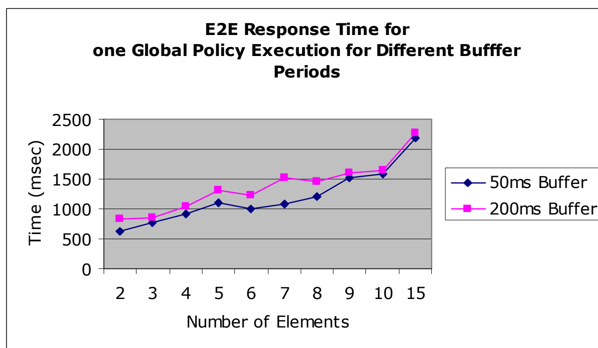

Fig. 21: Affect of Buffer Time on E2E Response.

The buffer time has a proportionally greater affect for a smaller element set, as would be expected. This parameter needs to be set carefully on a multi-host system to cater for the maximum expected out-of-sequence delay without causing unnecessary delay in the local manager.

<u>Duplicate Policies</u>

Duplicate policies delay overall processing time. The following test shows the E2E time spent on executing two global policies, where one is a duplicate.

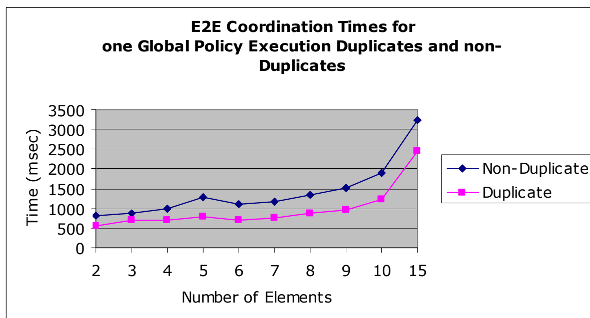

Fig. 22: Time Spent Processing Duplicate

The time spent on the duplicate is quite significant in comparison to the 'correct' policy. This is because the coordinator waits for all responses from the local managers even if it is a duplicate policy. However, the local manager deals with duplicates efficiently by responding immediately with minimum local processing.

Start-up Times

The following graph gives an indication of the element start-up time, when starting different numbers of elements concurrently on the same host. About 90% of the time is spent installing the policies locally, approximately 5 seconds, as script engine parsing is slow. The time to join the community and fetch the policies is about 200-300ms on average.

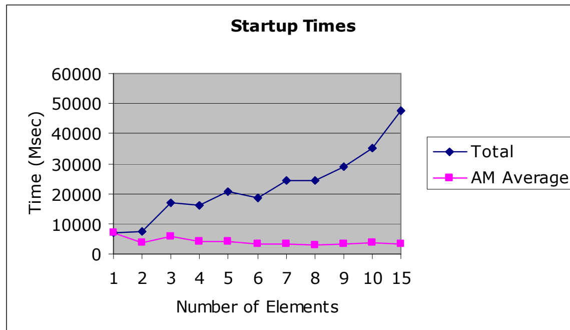

Fig. 23: Start-up Times

Element Distribution

Unfortunately it was not possible to check the performance when the elements are located in the same container or on different hosts. However, Braun [73] shows that the extra time for negotiation across different host machines (when the agent platform is distributed), or when more than one JADE platform is used is not significant. However, if more than one JADE platform is used it seriously impacts the DF search time which could be a problem in this solution as the DF is referenced each time a request message is sent, but the use of more than one JADE platform is only relevant for inter-domain solutions. Burbek [52] shows that locating agents on different containers increases message time as RMI, as opposed to events, is used for communication. Locating agents on different host machines connected by a fast network can sometimes provide better performance, especially if there are many agents, which may thrash the processor.

Scalability

The JADE environment is quite scalable, as demonstrated by Burbek in [52] as agents can be distributed across many hosts. The performance tests in the previous section show the response time when up to 15 elements (SC-1) are located on the same host, which is poor due to thread context switching on the host. To get some idea of how the system may scale beyond this number of elements and concurrent policies a simple message model is defined and used to determine the message volume for the system.

Message Model

In general the number of messages in the system at any time, or message volume, can be expressed as follows:

$$\text{Message Volume} = \sum_{i=1}^{P} (2n_i + 2) + (c_i * 2n_i)$$

Where :

P = Number of simultaneous unique global policy executions.

n = Number of elements that are the subject of a particular policy.

c = Proportion of elements subject to a change resulting from a particular policy decision.

Fig. 24: System Message Model

Note that the model assumes unique executions, or no duplicates, as the occurrence of duplicate triggers is application dependent.

Using the message model and assuming maximum participation, as in the performance tests, sample message volumes were calculated, as shown below.

Concurrent Policies	Elements Involved	Probability Change	Number Messages
1	50	1	202
1	100	1	402
1	50	1	202
2	100	1	804
3	50	1	606
3	100	1	1206
5	50	1	1010
5	100	1	2010
8	50	1	1616
8	100	1	3216
10	50	1	2020
10	100	1	4020
20	50	1	4040
20	100	1	8040
50	100	1	20100

Fig. 25: Example System Message Volumes

The number of concurrent policy executions has the largest impact, as would be expected. Assuming the same conditions in the performance tests above, where an average message incurred 35ms, the execution time for 10 concurrent policies with 50 elements becomes 70 seconds, however it is unlikely that the system would scale this way when one host is used.

With such high traffic volumes there is the possibility of bottlenecks. The Mayor is unlikely to become a bottleneck as it only takes a short time (60-80ms) and is only accessed once per global

execution, however, the DF (and AMS) is also centralised and is searched for each request message that is sent to an element.

One issue to consider is the distribution of the coordinator load, that is, the number of simultaneous global policy executions being co-ordinated by each autonomic manager. On average this load should be spread evenly among the autonomic managers but an uneven load is possible if one particular element tends to trigger events before others or its role is subject to many different global policies. A number of techniques could be used to spread this burden including: coordinator election based on lowest current load, the use of some shared state showing current load or a mechanism where a coordinator checks its load and passes the coordination function if it is too busy. However it is important that any solution is fast and does not generate a lot of messages or extra work for the already over burdened autonomic manager.

In general the fact that the elements can be distributed across different hosts makes the system more scalable than a centralised solution.

Reliability

The current solution provides some techniques to make the system reliable (RL-1), including, policy exceptions, message timeout handling, message status codes and general error handling. However, they are many weaknesses when it comes to element and message faults. The failure semantics specified for the application determine the required level of reliability and this can vary a lot. There are a number of failure points that are not currently catered for as they are beyond the scope of the system, including:

- Coordinator loss: This can occur at any point in the global policy execution.
- Local manager loss: This can occur at any point
- Mayor loss: This prevents global executions
- Service registry loss: Prevents all interactions
- Message loss/delay: Partially handled by application and underling transport system. Message ordering is handled by local manager buffer.

The system can be made more reliable by replicating or distributing the central nodes as outlined earlier and introducing persistence in the Mayor (DB), Repository (LDAP) and DF (can be provided by JADE using HSQLDB) as well as replication or distributed as outlined previously. Message delivery can be made more reliable by using JADE's persistent delivery mechanism. The loss of the local manger needs to be handled by the element or AMS monitor

and a new instance started. There are a number of techniques from distributed computing to handle coordinator failure, namely three-phase-commit, which would allow another autonomic manager to continue the protocol if the coordinator fails.

Integrity

Integrity problems can easily arise in the prototype when two simultaneously executing policies access or change the same resource. There is a high likelihood of this occurring in the prototype as there is no resource locking and there is a long delay between the decision and change policies. Such conflicts occur in all PBM systems and are an active area of research. Lupu [21] and Moffet [74] describe two main types of conflicts: conflicts resulting from the policy model, due to conflicts in actions based on the same event-condition, or conflicts resulting from the application, as a result of overlapping target objects. Policy conflict detection is notoriously difficult to perform, especially dynamic detection and a number of techniques exist to identify or deal with this ranging from implementing conflict detection elements [7] to deriving policies from high level goals and techniques outlined in [75]. Policy based conflicts are catered for in a simple way in this system through the use of policy priorities. There is no easy solution to application related conflicts and any solution would be based on some form of resource locking. This however leads to inefficiencies during the 'decision – change' conversation as a change request may not be received.

Application Integration

An important question is can the solution be easily adapted and is it generic enough to be useful (EX-1)? A generic autonomic manager was defined which is can be easily extended to provide element specific management. The element must supply certain properties for member admission and implement local MOs based on their interface to ensure the policies execute (AP-1). Simple test elements were defined with varying MO values and local decision functions for testing purposes. The MO is probably the best place to put the local and global decision action and the use of the generic and extensible local option response class worked well in that it was easy to define new local policies and new MOs.

Interoperability

The solution provides interoperability (IO-1) if all elements agree on the interaction protocol, have the same understanding of the policy subject 'world' and adhere to the common MO interface. The interaction protocol is built into the autonomic manager which is inherited by the element so any elements based on this can interoperate. However, this is quite restrictive and there maybe scope for the elements to discover the protocol or to use ontologies to support interactions between more heterogeneous elements.

Security

There is no provision for security in the system (SR-1). However, JADE has a security model that enables authentication and authorization of agents and secure communication is also provided using Secure Socket Layer (SSL).

5.5 Summary/Conclusions

There are a number of strengths and weaknesses with the prototype solution. Many of these relate to non-functional requirements, which were out of scope for the prototype, but are important to consider.

The prototype provides a useful autonomic manager framework and simple interaction protocol, however the protocol is not standardised and is somewhat inflexible. A flexible policy model was defined but not fully compliant with standards. Autonomy is preserved across the elements. The solution incorporates centralised services that are potential failure points and bottlenecks. However, several solutions are recommended for decentralising these if this is critical to the application. The general performance is adequate for the original small scale system. Larger scaling will require element distribution across a number of hosts and some optimisations. There are several reliability problems that are not catered for in the prototype but can be solved using techniques outlined above. The general agent approach is quite suitable to the problem and allows for further extensions and benefits from techniques from the MAS world. Application specific elements can be developed using the generic autonomic managers defined here. In virtually all cases the original requirements were met.

6 Conclusions

6.1 Main Conclusions

6.1.1 Objectives and Achievements

The main motivating question was to determine the *interactions and shared knowledge needed for the autonomic elements to coordinate in order to fulfil the policy objective.*

This raised four key sub-questions used to determine a number of specific system requirements. This section attempts to determine the degree to which these four questions were answered and the general conclusions that can be drawn from the implementation with respect to these questions.

What types of policies are needed for configuring global resources and how are they specified and distributed to the elements?

Two main types of policies, global and local, were required. Global decisions required implementation policies for making local decisions and local changes (in order to respect local autonomy and group decision making) were required. The global policy specification required an executable policy rule to guide the global decision and so this was not a high level policy. This policy model was not initially obvious and seemed cumbersome, but proved very suitable and flexible for specifying the policies and is in a form that was possible to implement. It is also flexible in the use of subject list and implementation groups to cater for different role groups that require a different decision and change policy.

It was necessary to identify specific polices, in terms of function, for an autonomous environment of peers. A democratic model was defined based on autonomy and consensus and 'weak' policies identified based on the likely dependencies between the elements. Three types of weak policies were identified to cover most group related policies that required group decisions.

A simple mechanism was provided to distribute the relevant policies from a central repository in the policy schema and informing the elements of who is involved in a global decision without revealing their policies, which worked well.

What information, or knowledge, is needed centrally to support the coordination?

Very little central knowledge was needed, except for the infrastructure registry function. A traffic- related service, the Role-Member service, was centralised in the Mayor component and accessed during global execution to locate current elements in the role. This also returned the 'tick' for duplicate policy arbitration. Neither of these functions really need to be centralised and can be decentralised using techniques have already been outlined in the evaluation chapter.

The global resource model was the service information held by the Mayor. All other resources were at the element level and stored as local MOs, as the global resource usage is the aggregation of local uses.

What coordination mechanisms are required?

The autonomic manager needed to play some coordinator role for each instance of a global policy that it triggers. It also plays a local manager role for a number of simultaneous policy executions. There needs to be a way of forming dynamic element groups per policy which was determined quite quickly by the coordinator when the policy was triggered. There maybe issues regarding coordinator load, but this depends on the application and how the distribution of triggered events.

There were many ways to implement the element communication. A simple interaction protocol was most suitable as it provided close control, unlike events that required a decision interaction and optional change interaction.

Mechanisms were implemented for handling duplicate policy triggers locally within an element and within the element set. This was tricky due to the different autonomic manager roles and system states but successful in terms of actually working and not generating large numbers of extra messages, however there maybe scope for improvement here.

How can this be generalised for different applications?

A generic autonomic manager was developed which was easily adapted to a specific element. The testing framework generated a number of these application specific elements by only varying values in the specific MOs. The element specific parts were the element manager and the managed element itself with its managed objectives.

The element implements element specific decision logic according to an interface for the MO.

There is an issue regarding resource conflicts due to multiple policy execution affecting different managed resources. This was out of the project scope but some solutions have been proposed in the evaluation chapter.

6.1.2 Improvements

Several improvements have been already outlined in the evaluation chapter and can be summarised as follows: Improve reliability by incorporating resource locking and a 2PC similar to that proposed by [60]. Distribute the Role-Member service by perhaps extending the DF to include it. Use the CID as a the 'tick' for duplicate arbitration. Replicate the AMS and DF and ass message persistence.

Extend the message to handle multiple resource setting as may be required by utility poilices.

6.2 Future Work

The current solution provides a working prototype and for further extending or pursuing distributed policy questions, such as the following:

- Can policies be refined into the format specified by the policy model?
- Is it possible to make the autonomic manager framework more generic to the application and perhaps downloadable or with a more generic interface to the MOs?
- To what extent can the interaction protocol be extended to support negotiation and other dependencies?
- What mechanisms are best employed to select a coordinator to spread the coordinator load?
- How can JADE's support for ontologies best employed to support more heterogeneous elements?
- How can the decision making be extended to cover subgroups and hierarchies of subgroups?
- How can hierarchies of global resources be best modelled and shared?

6.2.1 Summary

In order to provide a way of coordinating policy across autonomous elements it was necessary to provide the following:

- A flexible policy model to represent the global and local decisions
- Policy execution that preserves autonomy across the elements.
- A method for coordinating policy decisions between subgroups of elements
- A simple extensible model for consensus policies termed weak policies base on utility, voting, and value.
- Mechanisms for handling concurrent policy execution.

The evaluation showed that policy execution across autonomous nodes based on the agent approach is feasible for small numbers of simultaneous policy executions and that specific policy types and interaction mechanisms and are needed to support global coordination while preserving element autonomy.

Distributed policy execution is a relatively new research area and there is much scope for further investigation of distributed policy execution.

7 Appendix

7.1 Policy Schemas and Policy Examples

7.1.1 Policy Schemas

Local Policy Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="local_policy">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="description"/>
        <xs:element ref="event"/>
        <xs:element ref="subject_list"/>
        <xs:element ref="condition"/>
        <xs:element ref="action"/>
        <xs:element ref="exception"/>
        <xs:element ref="priority"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="description" type="xs:string"/>
  <xs:element name="event" type="xs:string"/>
  <xs:element name="subject_list">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="subject" maxOccurs="5"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="subject" type="xs:string"/>
  <xs:element name="condition" type="xs:string"/>
  <xs:element name="action" type="xs:string"/>
  <xs:element name="exception" type="xs:string"/>
  <xs:element name="priority">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="1"/>
        <xs:maxInclusive value="10"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

Global Policy Schema

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
   <xs:element name="global_policy_definition">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="global_policy"/>
            <xs:element ref="local_implementations"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="global_policy">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="description"/>
            <xs:element ref="event"/>
            <xs:element ref="condition"/>
            <xs:element ref="action"/>
            <xs:element ref="exception"/>
            <xs:element ref="priority"/>
         </xs:sequence>
         <xs:attribute name="name" type="xs:NCName" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="local_implementations">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="local_policy_impl_group" maxOccurs="5"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="local_policy_impl_group">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="subject_list"/>
            <xs:element ref="decision_policy"/>
            <xs:element ref="change_policy"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="subject_list">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="subject" maxOccurs="5"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="subject" type="xs:string"/>
   <xs:element name="decision_policy">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="description"/>
            <xs:element ref="event"/>
            <xs:element ref="condition"/>
            <xs:element ref="action"/>
            <xs:element ref="exception"/>
            <xs:element ref="priority"/>
         </xs:sequence>
         <xs:attribute name="name" type="xs:NCName" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="change_policy">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="description"/>
            <xs:element ref="event"/>
            <xs:element ref="condition"/>
            <xs:element ref="action"/>
            <xs:element ref="exception"/>
            <xs:element ref="priority"/>
         </xs:sequence>
         <xs:attribute name="name" type="xs:NCName" use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="description" type="xs:string"/>
   <xs:element name="event" type="xs:string"/>
   <xs:element name="condition" type="xs:string"/>
   <xs:element name="action" type="xs:string"/>
   <xs:element name="exception" type="xs:string"/>
   <xs:element name="priority">
      <xs:simpleType>
         <xs:restriction base="xs:integer">
            <xs:minInclusive value="1"/>
            <xs:maxInclusive value="10"/>
         </xs:restriction>
      </xs:simpleType>
```

7.1.2 Policy Examples

Global Optimisation Policy Example

```xml
<!--
    The XSD file is stored in the directory above the policies ../
-->
<global_policy_definition xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../GlobalPolicySchemaPA2.xsd">
  <global_policy name="Throughput">
    <description>Optimise the throughput when dropping transactions</description>
    <event>ON LowThroughput</event>
    <condition><![CDATA[
        IF statsMO.avgReqReceivedSec > 1 && statsMO.avgPcntProcessedSec < 10
      ]]></condition>
    <action>
      THEN threadMO.optimise(responses)
    </action>
    <exception>log()</exception>
    <priority>2</priority>
  </global_policy>
  <local_implementations>
    <!--
    There can be a number of groups, each containing a decision and change policy
    for a particular subject list. see XSD for rules
   -->
    <local_policy_impl_group>
      <subject_list>
        <subject>Service/Financial</subject>
      </subject_list>
      <decision_policy name="DetermineThreads">
        <description>Determine the optimal local max threads</description>
        <event>ON DetermineThreads</event>
        <condition><![CDATA[
            IF threadMO.maxThreads <= threadMO.threadLimit && threadMO.maxThreads > 0
          ]]></condition>
        <action>
          THEN threadMO.optimise()
        </action>
        <exception>log()</exception>
        <priority>3</priority>
      </decision_policy>
      <change_policy name="SetMaxThreads">
        <description>Sets the max threads available.</description>
        <event>ON UpdateThreads</event>
        <condition><![CDATA[
            IF threadMO.maxThreads <= threadMO.threadLimit
          ]]></condition>
        <action>
          THEN threadMO.setMaxThreads(newvalue);println "Updated Value: " + threadMO.getMaxThreads()
        </action>
        <exception>log()</exception>
        <priority>2</priority>
      </change_policy>
    </local_policy_impl_group>
  </local_implementations>
</global_policy_definition>
```

Global Value Policy Example

```
<!--
    The XSD file is stored in the directory above the policies ../
-->
<global_policy_definition xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../GlobalPolicySchemaPA2.xsd">
  <global_policy name="SetPriorityLevel">
    <description>
      Set the Thread Priority Level to X (8) if at least Y% (80) aggree. Only set
      for those that agree.
    </description>
    <event>ON SetThreadPriority</event>
    <condition>IF true</condition>
    <action>
      THEN threadMO.setPriority(responses, 10, 80)
    </action>
    <exception>log()</exception>
    <priority>2</priority>
  </global_policy>
  <local_implementations>
    <!--
    There can be a number of groups, each containing a decision and change policy
    for a particular subject list. see XSD for rules
   -->
    <local_policy_impl_group>
      <subject_list>
        <subject>Service</subject>
      </subject_list>
      <decision_policy name="CheckSetPriorityLevel">
        <description>Checks if can set priority level here.</description>
        <event>ON DecideThreadPriority</event>
        <condition>
            IF true
        </condition>
        <action>
          THEN threadMO.oktoSetPriority(10)
        </action>
        <exception>log()</exception>
        <priority>2</priority>
      </decision_policy>
      <change_policy name="SetPriorityLevel">
        <description>Sets the priority level.</description>
        <event>ON SetPriorityLevel</event>
        <condition>
            IF true
        </condition>
        <action>
          THEN println "Before Value: " + threadMO.getPriorityLevel();
              threadMO.setPriorityLevel(newvalue);
              println "Updated Value: " + threadMO.getPriorityLevel()
        </action>
        <exception>log()</exception>
        <priority>2</priority>
      </change_policy>
    </local_policy_impl_group>
  </local_implementations>
</global_policy_definition>
```

Global Agreement Policy Example

```
<!--
    The XSD file is stored in the directory above the policies ../
-->
<global_policy_definition xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../GlobalPolicySchemaPA2.xsd">
  <global_policy name="SetInitialThreadSize">
    <description>
      Set the Initial Max Tread Size by allocating a % (60) among services with
      historical high use services getting x (2) times as much as those with low usa
      up to a maximum of 10 threads per service.
    </description>
    <event>ON Initial</event>
    <condition>IF true</condition>
    <action>
      THEN threadMO.initialSet(responses, 60, 2, 10)
    </action>
    <exception>log()</exception>
    <priority>8</priority>
  </global_policy>
  <local_implementations>
    <!--
    There can be a number of groups, each containing a decision and change policy
    for a particular subject list. see XSD for rules
   -->
    <local_policy_impl_group>
      <subject_list>
        <subject>Service</subject>
      </subject_list>
      <decision_policy name="InitialThreads">
        <description>Set the Initial Max Threads</description>
        <event>ON InitialThreads</event>
        <condition>
            IF true
        </condition>
        <action>
          THEN threadMO.getInitialDemand(statsMO.avgRecentThreadUsage)
        </action>
        <exception>log()</exception>
        <priority>6</priority>
      </decision_policy>
      <change_policy name="SetInitialMaxThreads">
        <description>Sets the max threads available.</description>
        <event>ON UpdateInitialThreads</event>
        <condition>
            IF true
        </condition>
        <action>
          THEN println "Before Value: " + threadMO.getMaxThreads();
              threadMO.setMaxThreads(newvalue);
              println "Updated Value: " + threadMO.getMaxThreads()
        </action>
        <exception>log()</exception>
        <priority>6</priority>
      </change_policy>
    </local_policy_impl_group>
  </local_implementations>
</global_policy_definition>
```

Local Policy Example

```
<!--
      The XSD file is stored in the directory above the policies ../
-->
<local_policy name="SetQueueSize" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../LocalPolicySchemaPA2.xsd">
   <description>Sets the Queue Size</description>
   <event>ON DroppedWarning</event>
   <subject_list>
      <subject>Service/Financial</subject>
   </subject_list>
   <!-- Use CDATA when using XML type symbols in script -->
   <condition><![CDATA[
         IF queueMO.maxSize < queueMO.sizeLimit
      ]]></condition>
   <action>
      THEN println "Before Queue Size: " + queueMO.maxSize;
           queueMO.setMaxSize(statsMO.avgQueueLength + statsMO.avgReqDroppedSec);
           println "After Queue Size: " + queueMO.maxSize;
   </action>
   <exception>log()</exception>
   <priority>7</priority>
</local_policy>
```

7.2 Abbreviations

2PC	Two-Phase Commit
ACID	Atomicity, Consistency, Isolation, and Durability.
ACL	Agent Communication Language
AM	Autonomic Manager
AM	Autonomic Manager
AMS	Agent Management System
CDPS	Cooperative Distributed Problem Solving
CIM	Common Information Model
DEN-ng	Directory Enabled Networking Next Generation
DF	Directory Facilitator
DMTF	Distributed Management Task Force
ECA	Event-Condition-Action
FCAPS	Fault, Configuration, Accounting, Performance and Security
FIPA	Foundation for Intelligent Physical Agents
IETF	Internet Engineering Taskforce
IIOP	Internet Inter-ORB Protocol
JADE	Java Agent Development Environment
JESS	Java Expert System Shell
JMS	Java Message Service
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol

MAPE	Monitor Analyse, Plan, Execute
MAS	Multi-Agent Systems
ME	Managed Element
MO	Managed Object
NGOSS	New Generation Operations Systems and Software
PBM	Policy Based Management
PCES	Policy Console and Editing System
PCIM	Policy Core Information Model
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PMAC	Policy Management for Autonomic Computing
PMT	Policy Management Tool
RMI	Remote Method Invocation
RM-ODP	The ISO Reference Model for Open Distributed Processing
SID	Shared Information/ Data
SLA	Service Level Agreement
TMF	TeleManagement Forum
W3C	The World Wide Web Consortium
XSD	XML Schema Definition

8 Bibliography

1. IETF, *RFC 3198 - Terminology for Policy-Based Management*. 2001, IETF.

2. Verma, D.C., *Policy-based networking : architecture and algorithms*. Technology series. 2001.

3. Moffett, J.D. and M.S. Sloman, *Policy Hierarchies for Distributed Systems Management.* 1993.

4. Flegkas, P. *A Policy-Based Quality of Service Management System for IP DiffServ Networks*. 2002 [cited.

5. Sloman, M., *Policy driven management for distributed systems.* Journal of Network and Systems Management, 1994. **bf 2**: p. 333.

6. Wies, R. *Using a Classification of Management Policies for Policy Specification and Policy Transformation*. in *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management,* . 1995. Santa Barbara, CA.

7. Strassner, J.S., *Policy-based network management : Solutions for the Next Generation*. The Morgan Kaufmann series in networking. 2004, Amsterdam, Boston: Morgan Kaufmann Publishers. xvii, 516 p.

8. TeleManagementForum, *Shared Information/Data (SID) Model, Addendum 1-POL Common Business Entity Definitions – Policy.* 2003(NGOSS Release 3.5).

9. Dakshi Agrawal, K.-W.L., and Jorge Lobo, IBM T. J. Watson Research Center, *Policy-Based Management of Networked Computing Systems.* IEEE Communications Magazine, 2005.

10. N. Damianou and N. Dulay and E. C. Lupu and M.Sloman, M., *Ponder: a language for specifying security and management policies for distributed systems.* Imperial College Research Report DoC 2000/1, 2000.

11. Lobo, J., R. Bhatia, and S.A. Naqvi. *A Policy Description Language*. in *Proceedings of the 6th National Conference on Artificial Intelligence*. 1999. California, USA.

12. IBM, *Autonomic Computing Policy Language*. 2005, IBM.

13. Damianou, N., et al. *A Survey of Policy Specification Approaches*. 2002 [cited.

14. Naqvi, S., P. Massonet, and A. Arenas, *A Study of Languages for the Specification of Grid Security Policies*, in *CoreGRID Technical Report Number TR-0037*. 2006, Institute on Knowledge and Data Management.

15. Westerinen, A. and J. Schott. *Implementation of the CIM Policy Model Using PONDER*. in *POLICY*. 2004: IEEE Computer Society.

16. IETF, *Policy Core Information Model (PCIM) Extensions RFC 3460*. 2003, IETF.

17. DMTF, *CIM Policy Model White Paper (CIM Version 2.7)*. 2003, DMTF.

18. Farhang, B. and R. Kopeikin, *Policy-Based Quality of Service in 3G Networks.* Bell Labs Technical Journal 2004. **9(1)**.

19. Richard B. Hull and e. al., *Policy Enabling the Services Layer.* Bell Labs Technical Journal, 2004. **9(1)**.

20. Davy, S., et al. *Policy-based architecture to enable autonomic communications - a position paper*. in *Consumer Communications and Networking Conference, 2006. CCNC 2006. 2006 3rd IEEE*. 2006.

21. Lupu, E.C. and M. Sloman, *Conflicts in Policy-Based Distributed Systems Management.* IEEE Transactions on Software Engineering, 1999. **25**(6): p. 852-869.

22. Baliosian, J. and e. al. *Self-configuration for Radio Access Networks*. in *Seventh IEEE International Workshop on Policies for Distributed Systems and Networks*. 2006. Ontario.

23. Bandara, A.K., et al. *A Goal-based Approach to Policy Refinement*. in *POLICY*. 2004: IEEE Computer Society.

24. Bandara, A., *A Formal Approach to Analysis and Refinement of Policies (PhD Thesis)*, in *Faculty of Engineering*. 2005, University of London: London.

25. Lymberopoulos, L., E. Lupu, and M. Sloman, *An Adaptive Policy-based Framework for Network Services Management.* J. Network Syst. Manage, 2003. **11**(3).

26. Horn, P., *The IBM Vision for Autonomic Computing*. 2001, IBM: www.research.ibm.com/autonomic/manifesto.

27. Murch, R., *Autonomic Computing*. 2004: IBM Press.

28. Ganek, A.G. and T.A. Corbi, *The dawning of the autonomic computing era.* IBM Systems Journal, 2003. **42**(1).

29. IBM, *An architectural blueprint for autonomic computing.* 2004, IBM.

30. Strassner, J. *Using Autonomic Principles to Manage Converged Services in Next Generation Networks*. in *IEEE International Conference on Autonomic Computing*. 2006. Dublin, Ireland.

31. Parashar and Hariri. *Autonomic Computing: An Overview*. in *International Workshop on Unconventional Programming Paradigms (UPP), LNCS*. 2004.

32. IBM, *Autonomic Computing*. 2006, IBM: http://www-128.ibm.com/developerworks/autonomic.

33. Kephart, J.O. and W.E. Walsh. *An Artificial Intelligence Perspective on Autonomic Computing Policies*. in *POLICY*. 2004: IEEE Computer Society.

34. Norvig, P. and S.J. Russell, *Artificial Intelligence: A Modern Approach (2nd Edition)*. 2002: Prentice Hall.

35. Woodridge, *An Introduction to MultiAgent Systems*. 2002: Wiley.

36. Singh, M.P. and M.N. Huhns, *Service-Oriented Computing - Semantics, Processes, Agents*. 2005: Wiley.

37. FIPA, *FIPA Agent Management Specification*. 2001, FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS: http://www.fipa.org/repository/standardspecs.html.

38. Luck, M., P. McBurney, and C. Preist, *A Manifesto for Agent Technology: Towards Next Generation Computing.* Autonomous Agents and Multi-Agent Sytems,, 2004(9): p. 203-252.

39. Tsai, W.T., X. Liu, and Y. Chen. *Distributed Policy Specification and Enforcement in Service-Oriented Business Systems*. in *icebe*. 2005. Los Alamitos, CA, USA: IEEE Computer Society.

40. Chadha, R. and e. al. *Policy-Based Mobile Ad Hoc Network Management*. in *IEEE International Workshop on Policies for Distributed Systems and Networks*. 2004. New York.

41. IETF, *RFC 2753 - A Framework for Policy-based Admission Control*. 2000: http://www.packetizer.com/rfc/rfc2753.

42. *Madeira: A peer-to-peer approach to network management.* Wireless World Research Forum, 2006.

43. Granville, L., D.M.d. Rosa, and e. al, *Managing Computer Networks Using Peer-to-Peer Technologies.* IEEE Communications Magazine, 2005. **October**.

44. Carey, K., D. Lewis, and V. Wade, *A Policy-based Approach to Composite Service Assurance for Ubiquitous Computing.* 2003.

45. Chadwick, D., et al. *Coordination between Distributed PDPs*. in *IEEE International Workshop on Policies for Distributed Systems and Networks*. 2006. Ontario.

46. Rawls, J., *A Theory of Justice*. 1971: Belknap. 560.

47. ISO, *Information Technology—Open Distributed Processing—Reference Model—Enterprise Language (ISO/IEC 15414)*, ISO, Editor. 2004.

48. Stergiou, C. and G. Arys. *A Policy Based Framework for Software Agents*. in *IEA/AIE*. 2003: Springer.

49. FIPA, *FIPA Abstract Architecture Specification*. 2002: http://www.fipa.org/repository/standardspecs.html.

50. Bellifemine F., P.A., Rimassa G. *JADE: a FIPA2000 compliant agent development environment*. in *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*. 2001. Montreal, Quebec, Canada: ACM Press.

51. Bellifemine, F., Caire, G., Trucco, T., Rimassa, G, *JADE Programmer's Guide (JADE 3.3)*. 2005: http://agentcities.cs.bath.ac.uk/docs/jade/.

52. Burbeck, K., D. Garpe, and S. Nadjm-Tehrani. *Scale-up and performance studies of three agent platforms*. in *2004 IEEE International Conference on Performance, Computing, and Communications*. 2004: IEEE.

53. FIPA, *Publically Available Agent Platform Implementations*. 2006: http://www.fipa.org/resources/livesystems.html.

54. FIPA, *Agents and Web Services Interoperability Working Group (AWSI WG)* 2005, FIPA: http://www.fipa.org/subgroups/AWSI-WG.html.

55. Cook, W.R. and J. Barfield. *Web Services versus Distributed Objects: A Case Study of Performance and Interface Design*. in *Proc. of the IEEE International Conference on Web Services (ICWS) 2006.* 2006. Chicago.

56. JINI, O., *The Community Resource for Jini Technology*. 2006: http://www.jini.org/wiki/Main_Page.

57. IBM, *Policy Management for Autonomic Computing*. 2005: http://www.alphaworks.ibm.com/tech/pmac.

58. Group, I.C.P.R., *Ponder Toolkit*. 2006: http://www-dse.doc.ic.ac.uk/Research/policies/index.shtml.

59. Codehaus, *Groovy Scripting Language*. 2006: http://groovy.codehaus.org/.

60. Nimis, J. and P.C. Lockemann, *Robust Multi-Agent Systems: The Transactional Conversation Approach.* 2002.

61. Pavlos Moraïtis, E.P., Nikolaos I. Spanoudakis. *Engineering {JADE} Agents with the Gaia Methodology*. in *Agent Technologies, Infrastructures, Tools, and Applications for E-Services (Lecture Notes in Computer Science)*. 2002: Springer.

62. Lab), M.N.e.a.T.I., *A Methodology for the Analysis and Design of Multi-agent Systems using JADE.* Due in International Journal of Computer Systems Science & Engineering, 2006.

63. FIPA, *FIPA Communicative Act Library Specification (SC00037)*. 2002: http://www.fipa.org/specs/fipa00037/index.html.

64. FIPA, *FIPA Request Interaction Protocol Specification*. 2002, FIPA: http://www.fipa.org/repository/standardspecs.html.

65. FIPA, *FIPA ACL Message Structure Specification*. 2002, FIPA: http://www.fipa.org/repository/standardspecs.html.

66. Taha, H.A., *Operations Research, An Introduction*. 5th ed. 1992: MacMillan.

67. McCann, J.A. and M.C. Huebscher. *Evaluation Issues in Autonomic Computing*. in *Grid and Cooperative Computing - {GCC} 2004 Workshops*. 2004. Wuhan, China.

68. Lim, H.-J., et al. *An Analysis and Evaluation of Policy-Based Network Management Approaches*. in *Networking and Mobile Computing, Third International Conference, (ICCNMC)*. 2005. Zhangjiajie, China: Springer.

69. Wooldridge, *An Introduction to MultiAgent Systems*. 2002: Wiley.

70. Birnam, K.P., *Reliable Distributed Systems*. 2005: Springer.

71. Aib, I., et al., *Analysis of Policy Management Models and Specification Languages*, in *Net-Con*. 2003, Kluwer. p. 26-50.

72. Bellifemine, F. and e. al., *JADE Administrator's Guide*, G. Caire, Editor. 2006: http://jade.tilab.com/doc/index.html.

73. Bircher, E. and T. Braun. *An Agent-Based Architecture for Service Discovery and Negotiation in Wireless Networks*. in *Wired/Wireless Internet Communications, Second International Conference, (WWIC) 2004,* . 2004. Frankfurt/Oder, Germany: Springer.

74. Various, *Network and Distributed Systems Management*. 1994: Addison-Wesley.

75. Dunlop, N., J. Indulska, and K. Raymond, *Methods for Conflict Resolution in Policy-Based Management Systems*. 2003.