An Investigation into Autonomic Middleware Control Services to Support Distributed Self-Adaptive Software

Nagwa Lotfy Badr

A thesis submitted in partial fulfilment of the requirements

of Liverpool John Moores University for the degree of

Doctor of Philosophy

of the

Liverpool John Moores University

School of Computing and Mathematical Science

November 2003

To my Parents

Abstract

Over recent years, many researchers have advocated the vision of a new generation of *smart computing* including networks services, which can function and/or manage its and other systems' operation independently of human intervention or control. Such a vision has presented many challenges to a range of research communities including; intelligent systems, cybernetics and AI communities. Such new research aspects and issues taking further to develop a system that has the ability to adapt and changes its behaviour dynamically at run time considering the users' requirements and environments.

DARPA has initiated a research program on self-adaptive software, which provides an application level with self-adaptation. Such a body of work is more concerned with system-level self-adaptation and less focused on the development of generative, programming models and/or software engineering for developing autonomic software and normative structures. More recently IBM has concentrated its efforts on supporting autonomic concepts for developing and deploying enterprise server-level solutions with self-managing, self-healing and self-protecting capabilities.

Extending existing work from the self-adaptive software and reflective middleware communities, our research, grounded in distributed software engineering proposes and develops a meta-control service and associated middleware services with its design model and architecture for deliberative middleware and application services. This contributes to the design of self-adaptive software and computing services with the ability to coordinate and control systems adaptation in response to either conflicts or inconsistencies.

Consequently, this approach acts as reference model or baseline architecture to facilitate a normative self-governance model that supports the safe self-adaptation of distributed applications for lifetime management. Based on the sequence model of monitoring, classifying, repairing, and adapting components, the proposed architecture hierarchy encompasses a number of components that include; monitoring, inconsistency, mismatch or conflict detection and diagnosis, solution selection, solution checking, enactment and system reconfigurations In addition, this work defines a

software control (meta-control) service, which acts as a middleware service to support self-management (i.e. autonomic) software with respect to the coordination issues between the interacting applications services. In addition, because this control service is itself distributed as mentioned earlier, it provides an immediate sharing of the information, resources, selected tasks and *system coordination* by using the concept of distributed shared memory (provided by the Jini middleware). The main three services in the proposed software control service or baseline architecture is the service manager, system controller and JavaSpace. An extension to the Beliefs, Desires and Intension (BDI) model referred there to Extendible BDI (EBDI), is also proposed and provides the means and mechanism to underpin the software control of self-governing systems where during system coordination processes, *system controlle*r controls and coordinated its system by exchanging constraints about their goals, norms, actions and predefined with respect to its beliefs that reported to the distributed shared space as well by the service manager to facilitate the development of autonomic control middleware services.

Acknowledgements

Praise to ALLAH; The Cherisher and Sustainer of the Worlds; Most Gracious; Most Merciful. Thanks to God who gave me the ability to do this work.

My deepest thanks and gratitude are to Professor A. Taleb-Bendiab of the School of Computing and Mathematical Sciences, Liverpool John Moores University for his valuable supervision, continuous help, significant feedback and criticism. His extensive effort enabled me to complete this work successfully and collectively shaped this thesis.

I would like to express my appreciation and thanks to Professor Madjid Merabti, Director of the School of Computing and Mathematical Sciences, Liverpool John Moores University for making this work possible by introducing me to the department and offering me all the necessary to complete this work.

My thanks as well to Dr. Carl Bamford, Mr. Andy Laws and Mr. Denis Reilly for their crucial technical assistance, useful guidelines and numerous discussion that clarified many aspects of this work.

I would like also to express many thanks to all colleagues, academic staff, administration staff, technicians and research students in the School of Computing and Mathematical Science, Liverpool John Moores University for their support.

I am also grateful to my father who has been a timeless source of inspiration and influence throughout my life and to my mother for her unfailing support, constantly prompting and motivating over years.

I would like to acknowledge my husband, Mostafa for his strong encouragement, as this work would never be possible without his constructive assistance, helpful support, and guidelines. My deepest thanks go to my children Mohamed and Hanna, who have been great motivators by their sacrifices, patient, constant love and indirect encouragement.

Last but not least I would like to express my heartfelt thanks to all my family, Nahala, Nevien, and Ahmed for their support in different ways during my studies.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations: Software Autonomy

Over the years, Information and Communication Technology (ICT) has gradually become an integral part of our economic and social fabric, and their design and management complexity has grown as rapidly as our requirements and dependence on the systems. The prevailing design of most current large-scale distributed systems can be characterised as *reactive and centralised* in nature, in that they are centrally managed and controlled, and their functions are essentially dependant on direct end-user interventions. There is a need in software design to shift from a centralised client/server model to building n-tier decentralised systems, and hence developing systems that are more dependable, scalable, robust and amenable to changing their own behaviour with minimal intervention from users.

Recently, many researchers have advocated the vision of a new generation of *smart computing* including networks services that should function and manage their systems' operation independently of human intervention [1]. Such an aspiration has provided many challenges to a range of research communities including; intelligent systems, cybernetics and AI communities. Now, there is a renewed interest by both academic and commercial communities in developing systems that adapt autonomously to their users' requirements and environments; for instance, to recover from an encountered/anticipated system failure, tune performance to fulfil a quality guarantee, and/or accommodate changes with respect to the number of participants and/or integration of new services. We are now at the point of the emergence of a new class of large-scale *decentralised* and *autonomic* applications that can operate independently or with minimal direct human control.

Taking this vision even further, IBM has characterized this as *autonomic computing*, and is now actively promoting, developing and deploying enterprise server-level solutions with self-managing, self-healing and self-protecting capabilities [1]. Prior to IBM's *autonomic computing* initiative, DARPA funded a research program on self-adaptive software [2], which applied control theory [3], AI planning and/or software reflection techniques to provide application-level self-adaptation mechanisms and/or heuristics [4]. Though, this body of work was more concerned with a system's self-adaptation level and focused on the development of generative, programming models and/or software engineering supports for fine-grain, software dynamic and predictable adaptation. Some of these issues are currently addressed within the DARPA-funded Dynamic Assembly for System Adaptability, Dependability, and Assurance (DASADA) initiative [5]. Here, work is underway to develop software engineering tools and techniques to support the design of software assured dynamic adaptation primarily using an architecture-driven approach coupled with probes and gauges to enable software to interact with an executing system to collect a range of measurement data. This is then translated into suitable metrics for system performance tuning and/or error recovery through adaptation.

Consequently, there is an increasing research trend in the development and/or application of self-adaptive software, autonomic software and reflective middleware for adaptive software. Also, there is still a lack of fundamental understanding of adjustable control models for autonomic behaviour to ensure and facilitate safe, predictable and software self-adaptation that is crucial of the distributed self-adaptive application's environment. However our research is build on the previous related work and on ongoing work in software management autonomic computing. But rather than simply looking at the static management at design time or complex dynamic management that is embedded in the functionality of the base-level of the applications, it provides a baseline architecture and a software meta-control model to define the requirements for developing an autonomic meta-base/middleware-based control service at runtime.

1.2 Challenges

While software autonomy represents an essential approach to delegating much of the software maintenance and/or management activities to the software itself, it

engenders a range of technical challenges to be addressed and requires the development of;

1. Reference models: software design patterns, baseline architecture and/or middleware for developers to design, deploy and/or manage self-adaptive software, thus enabling systems to monitor their behaviour and performance, to reconfigure when required and determine that any proposed software composition is compliant with its design, requirements and guarantees. Therefore we need to take into account and cope with the inherent uncertainty, complexity and scalability issues related to such systems.

2. Mechanisms: which can be used for runtime software component and service assembly to ensure safe and predictable software transformation that guarantees the required and desired properties. To this end, other facilities and utilities need to be developed and include;

    - How to access and reason about data coming from a variety of software instrumentation for monitoring and analyzing targets including the environment before deciding how or whether to react or not.
    - How to support predictable and conditional triggers to facilitate software change management based on detecting, filtering, and prioritizing system events and generating and coordinating self-repair change plans.
    - How to support negotiation to resolve conflicts emerging from a given proposed self-adaptation plan prior to its enactment.
    - How to reconfigure distributed systems by dynamically enabling a control service to adjust and control using its repair tactics and strategies for customisations and adaptation of the system itself.
    - What normative models can be used for instance to specify management policies, enforce and adapt to support software self-governance that may or may not permit any intended changes.

3. Experiment/benchmarks: demonstrating that complex systems (and sub-systems) can practically monitor and validate their runtime behaviour with respect to critical system properties, requirements and intended goals.

1.3 Approach

In this thesis, we aim to develop a meta-control model with baseline architecture and associated autonomic control middleware services to support the development and lifetime management of *deliberative software.* Such software could be composed (assembled) of networked software services and provided with a range of deliberative capabilities, such as self-governance, self-monitoring and self-repair to enable safe predictable self-adaptation and guarantees the required functional and non-functional properties are within specified tolerances.

Therefore the establishment of self-management, self-repair and self-adaptation, requires the uses of flexible infrastructure to support a full range of adaptation services, such as:

- The service manager service: which contain service control processes such as the service monitoring model, diagnosis model, repair model, and adaptation strategies.
- The system controller service: which include system control processes such as the system monitoring model, system repair strategies model and system reconfiguration model.
- The control rule base: which is accessible by the manager and controller to provide the domain and boundaries for specific application control strategies, repair plans, etc.

For theoretical support, this work draws a number of research results emerging from related fields including;

- Self-adaptive systems: using proposed models, requirements and theories to enable software to use feedback and feedforward, real-time monitoring and model-based control, such control theories are intended to provide software with the ability to evaluate its behaviour and environment against a given goal and revise behaviour in response to the evaluation [2, 4]. Further details will be provided in Chapter 2.
- Advanced software engineering: using middleware services to bridge the gap between the network layer and application layer, and using the event notification concept to enhance the level of communication between a server and its clients, using an exception handling model for

the safe termination of the application in the event of failure as well as classification exception type, and using the concepts of distributed shared memory (e.g. JavaSpace service) for both remote system coordination at run-time and for storage of the required information for system coordination to achieve lifetime management.

- Software agents: the work proposes the extension of the Beliefs, Desires, and Intension (BDI) model is EBDI to underpin the software control of self-governing systems. The system, using BDI concepts compare its current behaviour (i.e. beliefs) against its goals (i.e. desires) generates a decision (i.e. intension) that represents an actual action.

- In addition, this work follows an experimental research approach by aiming to develop, build and test new models of software meta-control and the associated baseline architecture and autonomic middleware control services.

1.4 Contributions

This work makes a number of novel contributions, all of which have been or are being submitted to relevant research publications[6-11]. There are summarised below;

1. Baseline architecture and model: which is influenced by and grounded in a range of current research on high-level software control, coordination, autonomic computing, deliberative systems, normative systems and adjustable autonomy. In particular, this work defines the requirements and a software architectural model for a middleware-based control service, which facilitate adjustable self-governance utility to support the safe self-adaptation of distributed applications for lifetime management. Based on the "monitoring-classifying -governing-adapting" model [12], the proposed architecture encompasses a number of components that include; monitoring, inconsistency, mismatch or conflict detection and diagnosis, solution generation, solution model checking, enactment and system reconfiguration (Chapter 6).

2. Software meta-control model: which represents how distributed applications services interact with their associated middleware service management and control services. The main elements are outlined below:

- The Service *Manager* is concerned with managing its service conflicts. Hence for each service there is a manager that looks after that service. The service manager has a hierarchy of control scripts/tasks that are:

    - The monitoring model uses a set of control rules to check monitored behaviour and architectural configuration and hence detects conflicts.
    - The diagnosis model involves the execution of control rules, activated by conflicts that identify and classify the conflict types to provide the basis for the selection of a conflict resolution operator.
    - The repair model is specified using contract-based assertions, pre-conditions and typical operators to provide operations that resolve a service's conflict. These operations are provided as primitive operations integrated into the service manager. Three key augmentations to a model are required to allow the appropriate decision to be taken for the detected conflicts. These are notification, repair operators, or thrown appropriate exception. Whichever is chosen to resolve the conflict, at the end of conflict resolution stage, the service state is stored in a shared space (JavaSpace Service), which is again monitored by the System Controller
    - Adaptation Engine: in which the service manager has to adapt the service according to proposed changes.

- The System controller is responsible for establishing and managing the coordination of the overall system's services and ensures that the interrelated system services are maintained and coordinated. The controller regularly checks the service state, stored previously by the service manager in the distributed shared space. The system controller applies the appropriate strategy according to the state; and contains three main models for the control of the system. These are:

- A monitoring model that has the ability to collect and store the information that is required to support and guide the resolution strategies within the control process. This model starts by checking the distributed shared space (e.g. JavaSpace, or T-Space). This is a resource or service shared between the distributed systems over the network. Each service manager stores its service state and this is received later by the system monitoring model to check the system's service states and starts the control process sequences

- The system repair strategies model that determines when, where, and how the repair or adaptation is required. The repair strategies must consider the functions of the services/application, the operating environment and its attributes and properties. Our resolution strategies are used to evaluate the effect of various alternative solutions based on the BDI model of deliberative systems [13].

- The system reconfiguration model that applies the required reconfiguration attached to the resolution strategy. This is dynamically interpreted from an XML document to a run-time executable model. For example, if the resolution strategy selects an alternative service to a failed one, the reconfiguration system should establish the required changes that result from the resolution strategy dynamically at runtime. For example, getNewManager(), notifyClient() and newConnect(),.

- The system associated interpreter model that is used to translate the external format (e.g. XML) of repair strategies or operators to a lower-level and executable level that is used in the code, therefore this model allows run-time changes within the code without the need to recode or recompile the system again.

3. An autonomic control middleware service: which includes a programming model to facilitate the development of adjustable-autonomic control middleware services that would facilitate customisation of self-adaptation control strategies and self-governance policies including norms and

authorities. Also, a control strategy markup language and associated interpreter has been developed to achieve a level of software "*separation of concerns*" and the externalisation of control models and knowledge from the core controller logic (Chapter 7-8).

1.5 Scope

In this research we propose a new control service to enable next generation software systems that can manage their own runtime structural or behavioural changes in reaction to, for example, a set of unpredictable hardware and/or software failures. In particular, this work focuses on:

- A new middleware control service, providing systems with a required functionality and knowledge for dynamic autonomy. In Table 1.1, we illustrate two main categories that focus on *either* actions or attributes that could change values, *or* actions and operations related to the services coordination and/or interactions with each other. This also explains the effectiveness of these changes on the system itself and its services.
- A generic capability for developing our control service, for example, the XML format.
- A *runtime* demonstrator to test the theory and illustrate the feasibility of the approach by detecting errors and inconsistencies and taking corrective action without disrupting ongoing processes.

Category	Action/Change	Effect
Service Related	Attributes	Performance/QoS
	Execution	Service Functionality
	Leasing/Contracts	Clients Requests
	Service Availability	Dependence
Operations		
Application	Events Handlers	Performance/QoS
Services	Listeners	Execution Time
Management Process	Execution	Latency

Table 1.1: Management categories of changes and effects

Furthermore, we tested our approach capability on three application examples. These are the GridPC example and the other two from an existing Jini application, namely EmergeITS, which is intended to realize the concept of a 3in1 phone service and web-based information service in Intelligent Networked Vehicles. For example, the 3in1 phone scenario is as follows:

- The demonstration has been applied to a 3in1 phone service that allows a mobile phone or Palm device to be used in one of three different modes, subject to the requirements of the user and service provider availability.
- The service manager service monitors the application service and reports its service status in the JavaSpace for the system controller service.
- The system controller checks the system desires against current beliefs (i.e. service manager report) and uses distributed shared memory (e.g. JavaSpace) as a tool for achieving system coordination.
- The negotiation and coordination of the overall system's services ensures that system sequences are maintained, coordinated and reconfigured using the appropriate strategy provided in an XML document.

1.6 Thesis Organization

The thesis is divided into ten chapters and is organised as follows:

Chapter 1 provides a general introduction to the work, challenges, contribution and a structured thesis outline.

Chapters 2 and 3, introduce the relevant background theories, principles and/or technology that are used or considered important to the understanding and development of our proposed model, outlining the required features of a self-adaptive autonomous based model. The survey covers several areas of distributed systems including, Service-Oriented Programming (SOP), Distributed System Development and Management, Middleware Technology and Computing Autonomy. Finally, the main problem to be addressed is defined.

Chapter 4 provides a literature review of related work drawn from a range of fields including; static distributed system management approaches and the dynamic approaches used so far. The thesis provides a comprehensive survey of related literature and details the various uses and understanding of both static management approaches or computation model management and dynamic management approaches. Static management approaches require human intervention during the management process such as conflicts resolution and coordination approaches and, strategy and plans representation approaches. On the other hand, the dynamic management process requires no user intervention during, for example, policy–based management, event-based management, architecture-based management or autonomic management. Finally, we present the required approach to establish the lifetime management processes of distributed software systems.

Chapter 5 presents the requirements of the proposed control service to support the autonomic management of self-adaptive software.

Chapter 6 describes the architectural design of our approach. A range of related research such as high-level software control, coordination, autonomic computing, deliberative systems and normative systems that influenced this designed are considered.

Chapters 7 and 8 provide an implementation approach using examples drawn from an on-going research project by evolving three application examples, GridPC, a 3in1 phone service and a web-based application service.

Chapter 9 presents an evaluation of our proof of concept, given the assumptions of the previous chapters and using two main applications, which are the 3in1phone application and a sorting algorithms application.

Chapter 10 presents a summary, concluding remarks and proposed future work.

Chapter 2

Background

2.1 Introduction

Prior to the full description of the proposed and developed autonomic control middleware service and associated functionality to support the design, deployment and lifetime management of distributed self-adaptive applications. Therefore, this chapter presents existing systems and techniques that are considered important to the understanding and development of autonomic middleware control service to support distributed self-adaptive software.

2.2 Self-Adaptive Software

R. Laddaga [2] gives a good definition for self-adaptive software as:

> *"... Self-adaptive software evaluates its own behaviour and changes behaviour when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible..."*

As noted by Laws *et al.* [14] the primary ambition of the self-adaptive software approach is

> *"... to devolve some of the responsibility for evolutionary activity to the software itself. Essentially, this requires embedding equivalent elements of the human software evolution process in the software itself, thus allowing autonomous adaptation to local conditions during runtime. Effectively, such software must be capable of detecting the need for change, either to address changing external conditions or for internal performance-related reasons, determine which elements to change and how they should be changed,*

planning and enacting the change and finally verifying the effectiveness and robustness of the resulting solution."

Most notable progress towards achieving such a vision has followed three guiding directions namely; control systems theory, dynamic planning systems and self-aware systems.

The main three elements considered by control theory are (1) the external environment, (2) the productive element of the system consisting of physical objects that are viewed as a factory or plant interacting with and providing products or services for that environment and, (3) A model-based self-control unit that ensures the plant meets the policies and norms of the environment, as the main characteristic that can distinct the structure of the plant is its physical makeup object is alive paralleled with the lifetime of the control system. Therefore, research has concentrated on the flexibility of the system's control element, in addition to the development of a hierarchy of even more complex control models, each designed to provide increasing adaptive capability to the control unit.

Consequently, plant management is achieved by integrating the control unit with both the system's goal and plant's model. The feedback control process is used here to monitor and evaluate the system performance in respect to its goals, and then the control unit according to its control goal selects appropriate control actions. Such approaches rely to some extend upon the plant's constancy and consistency and the stability of both the goal and the environment.

Parameterisation of both the controller and the plant model in an adaptive control approach have been employed to adopt adapting changing goals caused by unexpected environmental disturbances, and estimating mechanisms are also provided to address uncertainty in the plant model and subsequent control action. The integration of databases of plant models, approaches and associated system controllers could increase the flexibility of the changes in the system's structure allowing the appropriate model and controller for any particular situation to be selected, thus facilitating a reconfigurability stage of the whole system.

Such reconfigurability is directed towards the control elements of the system. Kokar *et al.* [15] have addressed the adaptation of both the control elements and software plant by incorporating both a high-level specification database and component database that

could make the reconfiguration of the control elements and software plant achievable [15]. However, such an approach requires that the software be provided with a degree of awareness of its goals, thus for evaluating the configuration of the system requires both the current intention and the external environment [16, 2]. However, the problem of evaluating multi-part systems is still not yet fully solved and remains challenging as there is a need to measure system components in order to determine normal and abnormal behaviour and therefore by identify either a solution or a replacement for such components.

Hence, the software system should be provided models of both its internal specification and capability and the external environment. In addition, these should have a self-adaptive capability thereby providing a degree of self-awareness to the system and thereby allowing changes raised in either internal or external circumstances to be identified [17]. Hence the system can perform the deliberative processes of performance evaluation, reconfiguration and subsequent adaptation. In general, these require systems to have the ability to decide which alternative is appropriate in response to a negative evaluation [2]. Such a view is integrated with the notion of dynamic planning and with that of the self-adaptive software area. Here, the system plans and possibly the system itself could be require changes in cases, for example, where no longer their plans are not suitable for new changes or external environments [16].

Of courses the attachment of such control systems to the operational software will add much overhead and complexity to the executing software, as Robertson *et al* [2] note:

> *"Managing complexity is a key goal of self-adaptive software. If a program must match the complexity of the environment in its own structure, it will be very complex indeed! Somehow we need to be able to write software that is less complex than the environment in which it is operating yet operate robustly.".*

Avoidance of such complexity and overhead in the self-adaptive software system may be addressed by presenting the system in a supporting architecture that is responsible for model maintenance, performance monitoring and evaluation and action adaptation [18, 19], thereby freeing the operational units to pursue their respective objectives [2].

2.2.1 Self-Adaptive Software Compared to Control System

The task of developing self-adaptive software is comparable to a control system and concerns new techniques of building a robust program with a control system in its structure. Such ideas are borrowed from control system thereby and adapted to self adaptive software [2].

An important problem for self-adaptive software is evaluation. Osterweil and Clarke [20] see this as a continuous measurement of the gap between the software system operation and its requirements, thereby providing the basis for self-improvement efforts. They describe the transition of the responsibility for the testing and evaluation of software from humans into automated tools and processes and suggest a process of automated continuous self-evaluation.

Meng [3] makes explicit the relationship between control system theory and self-adaptive software by developing a descriptive model of self-adaptation based on control systems and borrows the feedforward and feedback control paradigm from control theory. This considers that self-adaptive software consists of two components: the feed forward element that provides the specification of the software and its predictability and the feedback component that receives runtime feedback from its environment. However, the general model of self-adaptive software can be viewed from many different aspects, for example, a new programming paradigm, new architecture style, new modelling paradigm, and new software engineering principle. Meng [3] addressed the evaluation of self-adaptive software based on their different aspects, as follows:

- As a new programming paradigm, reflection programs could be modify themselves at run time and change their behaviours and as such are close to the concepts of self-adaptation. However, such programs cannot determine when and what the program needs to modify itself at runtime. Self-adaptive software generates evaluators at run time to check the deviation of the state of the program from its goal, then the control regime computes the distance between the current state and its goal state and adapts to maintain its stability and robustness.
- As a new architectural style, self-adaptive software needs to formalize the feed forward and feedback controller concepts and configuration, which are system structural components that are part of the Architecture Description Language (ACL). The "configuration" and "controller" architecture

description vocabulary in self-adaptive software maintains stability and settling time when the system transfer from one configuration to another.

- As a new modelling paradigm, reconfiguration in terms of self-adaptive software concepts uses adaptive control to allow the system to switch a control regime based on the runtime situation. Self-adaptive software transfers the feed forward process from the model to the executable and synthesises, however the feedback process transfers from execution to reconfiguration and hence to runtime re-synthesis.
- As a software engineering principle, a software system can be seen as a control system incorporating adaptation and reconfiguration based on adaptive control theory and, generalizes the control model as a concept of algorithm selection in software engineering. The system is provided with different algorithms and software adaptation becomes choosing a suitable algorithm for a particular environment.

2.2.2 Self-Adaptive Software Research Directions

There are further research directions for self-adaptive software. The *first* technology here is transient management, where self-adaptive software reconfigures itself to ensure robust performance of the program. Reconfiguration means any modification or change in systems parameters, however, although modifications may be more robust they may be also demonstrate undesirable transient effects. G. Simon, T. Kovacshazy and G Peceli [21] investigated the two important issues in transient management, which are (a) management that depends on the suitable selection of a structure that features of the transient properties. A more formal definition of reconfiguration transients is "*the difference between the measured value in the reconfigured system, and the idea value in reference system. The reference system is a hypothetical system, which operates in the new mode for a long time* " [15], and (b) the runtime control to support the transients management. The *second* technology is a model-based generative technology called 'Model-Integrated Computing' [22], and is applied to self-adaptive software. Transferring/Porting Model-integrated computing from design time to runtime helps in the design and implementation of dynamic embedded systems and these embedded models are independent and applicable to a reconfigurable architecture, which is an essential constraint for self-adaptation.

2.2.3 Example of Self-Adaptive Software Application

The aim of self-adaptive software research is the provision of an integrated software environment in which runtime data sets drive the organization and control of behaviour of software systems. Realizing such an aim depends on developing technologies that produce robust, fault-tolerant systems easily.

The Containment Unit supervisor "... *which is an active entity designed to represent a family of optimal contingency plans behaviour programs B-Pgms* ..." [23] for tracking a human subject was able to handle individual sensor faults as faults in the runtime [23]. This work provides a Port-Based Adaptive Agents Architecture (PB3A) [23] to facilitate the development and deployment of self-adaptive, distributed, multi-agent systems. The architecture specifies an agent-to-agent communication scheme using input and output ports and includes the necessary code for migration. It also provides specifications of how a Port-Based Agent (PBA) should be structured to allow agents to become self-adaptive. A PB3A module allows agent-to-agent communication testing to be done in isolation, allowing the system to built rapidly.

2.2.4 Self-Adaptive Software and Reflection

Reflective architectures enable programs to access thier own structural, behavioural and computational state to compensate for any changes in resources, context or environment. Reflection provides the tools for writing such a program but does not describe how it is done [4]. Self-adaptive software has a computation model to compare what the current program with the intended program and modifies the semantics of the current program to correct it. Shaul [24] argued for and described the use of reflection to support self-adaptive software for a network computing environment: The quality of self-adaptation, the degree of adaptation, robustness and the availability were explored by developing two frameworks using Java reflection, namely the HADAS and FARGO projects. These focused on intra and inter component adaptation mechanisms [24].

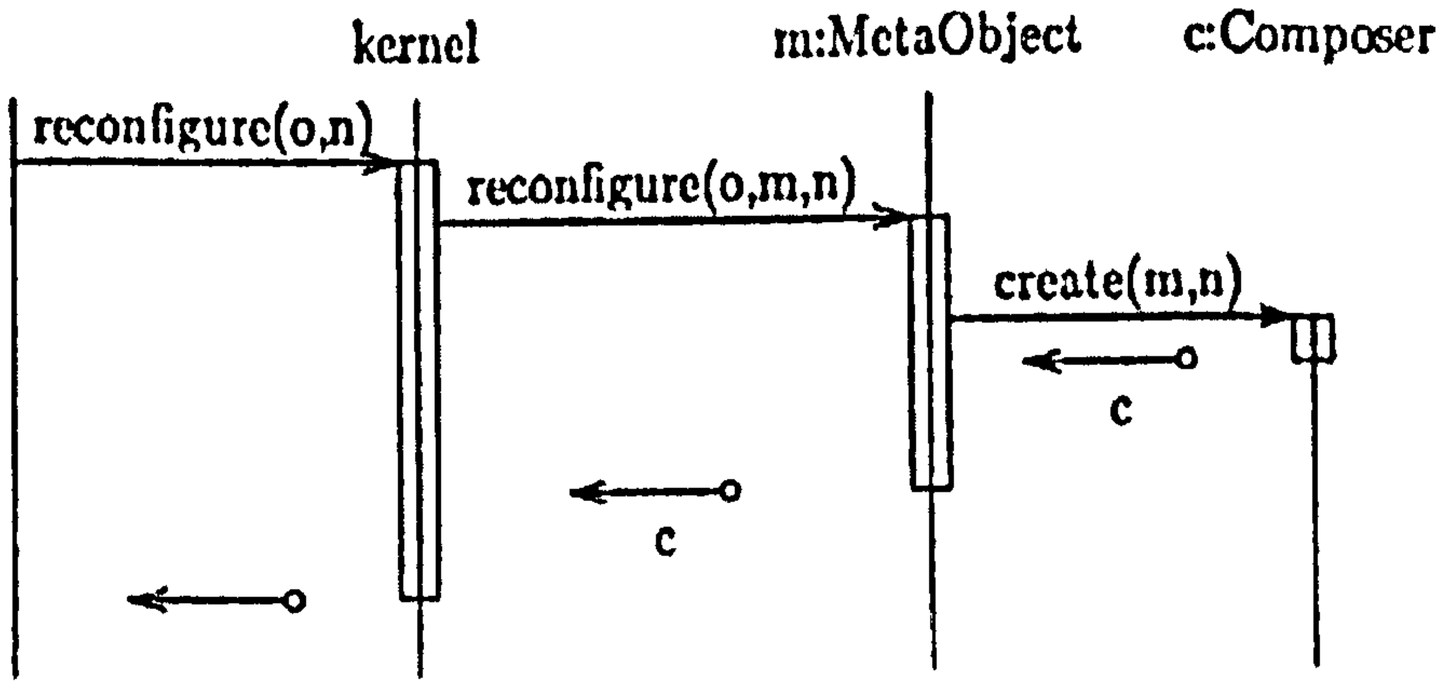


Figure 2.1: The Primary Meta-Object of Object [25].

Figure 2.1, illustrates a sequence diagram for a required reconfiguration that is based on computation reflection principles of the meta-level objects to undertake the monitoring and managing of an objects base set. In this case, the meta-object enables the base object to request a composer method for reconfiguration [25] , so a reflective software architecture and meta-level protocol represent a partial solution for creating self-adaptive software.

2.3 Autonomic Computing

Autonomic computing is an approach to self-managing computing systems with minimum interference; these control key functions without conscious awareness or involvement and increase productivity while hiding the complexity from users. Autonomic computing systems have the ability to manage themselves; they dynamically adapt to change in accordance with system policies and objectives. Self-managing systems can perform management activities based on situations they observe or sense in the environment [26]. The main characteristics of an autonomic system are detailed in [26] , as follows:

- An autonomic computing system needs to "know itself"; its components must also possess a system identity. It also detailed requires knowledge of its components, current status, ultimate capacity and all connections to other systems, to govern itself.
- An autonomic computing system must configure and reconfigure itself under varying (and in the future, even unpredictable) conditions.

- An autonomic computing system never settles for the status, it always looks for ways to optimise its workings. It monitors its constituent parts and fine-tunes workflow to achieve predetermined system goals
- An autonomic computing system must be able to recover from routine and unusual events that might cause some of its parts to malfunction, and discover problems, then find an alternate way of using resources or reconfiguring the system to keep functioning smoothly.
- An autonomic computing system must be an expert in self-protection. Also it must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity.
- An autonomic computing system must know its surrounding environment and act accordingly. It will find and generate rules for how best to interact with neighbouring systems. It should negotiate the use by other systems of its utilized elements, changing both itself and its environment in the process of adapting. While an autonomic computing system independent in its ability to manage itself, it must function in a heterogeneous world as well.
- An autonomic computing system anticipates the optimised resources needed while keeping its complexity hidden without involving the user in that implementation.

So autonomic computing will lead to automated management of computing systems. But that capability will provide the basis for much more, such as seamless e-sourcing and grid computing [27] to dynamic e-business and the ability to translate business decisions that managers make and policies that make those decisions a reality.

2.3.1 Autonomic Computing Architecture Concepts

A standard set of functions and interactions govern the management of the computing system and its resources, including client, server, database manager or Web application server. This is represented by a control loop (Fig. 2.2) that acts as a manager of the resource through monitoring, analysis and taking action based on a set of policies and rules [26].

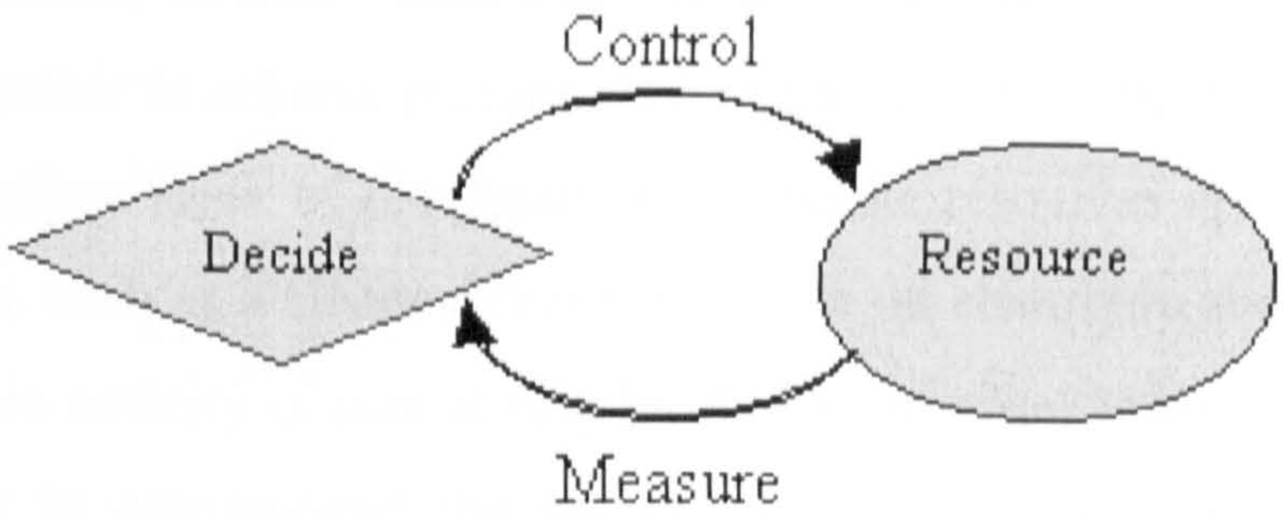


Figure 2.2: The control loop architecture.

These control loops, or managers can communicate with each other in a peer-to-peer context and with higher-level managers. For example, a database system needs to work with the server, storage subsystem, storage management software, the Web server and other system elements to achieve a self-managing IT environment.

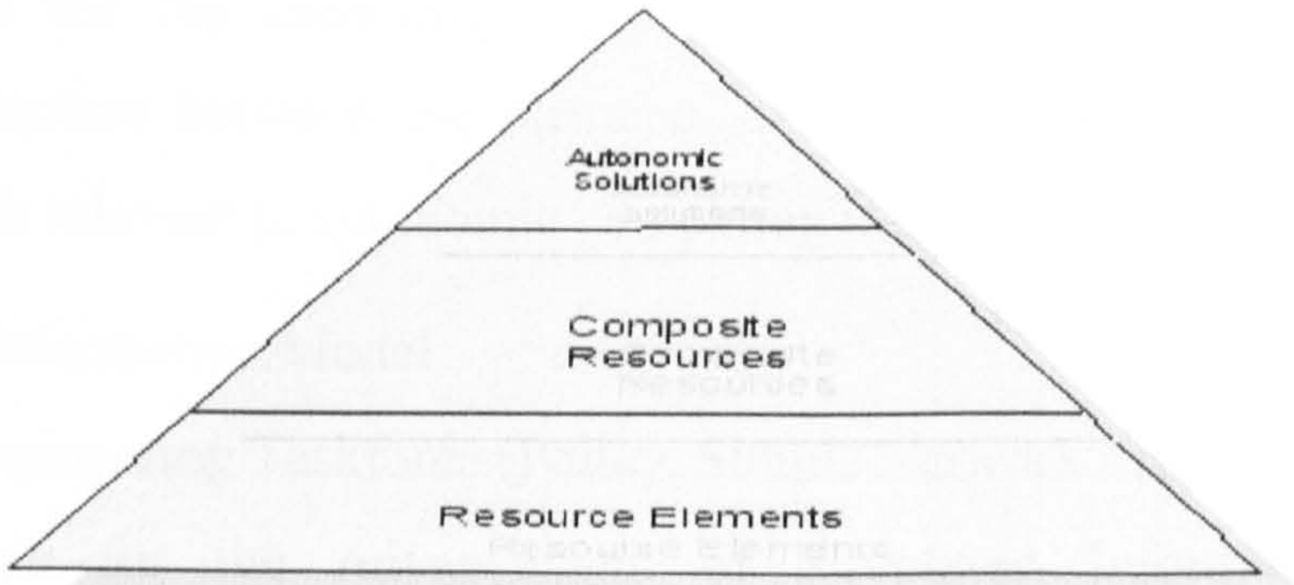


Figure 2.3: The hierarchy pyramid of the autonomic computing technologies [26].

The previous pyramid (Fig. 2.3) represents the hierarchy in which autonomic computing technologies will operate and is explained as follows:

- In the *lower layer* of the pyramid consists of the *resource elements* of an enterprise networks, servers, storage devices, applications, middleware and personal computers. Autonomic computing begins in the resource element layer, by enhancing individual components to configure, optimise, heal and protect themselves.
- In the *middle layer* of the pyramid, the resource elements are grouped into composite resources, which begin to communicate with each other to create self-managing systems. A pool of servers that work together to dynamically adjust workload and configuration to meet certain performance and availability thresholds can represent this. It can also be represented by a combination of

heterogeneous devices (databases, Web servers and storage subsystems) that work together to achieve performance and availability targets.

- In the *highest layer* of the pyramid, composite resources are tied to autonomic solutions, such as a customer care system or an electronic auction system. True autonomic activity occurs at this level. The solution layer requires autonomic solutions to comprehend the optimal state of business processes—based on policies, schedules, service levels and so on, and drive the consequences of process optimisation back down to the composite resources and even to individual elements.

2.3.2 Relevance to Autonomic Computing

Autonomic computing requires some open standards for the managed elements' sensors and effectors and for the knowledge to share between autonomic managers that describe the interaction between the elements of an IT system. Some existing and emerging standards relevant to autonomic computing include:

- Common Information Model
- Internet Engineering Taskforce (Policy, Simple Network Management Protocol)
- Organization for the Advancement of Structured Information Standards (OASIS)
- Java™ Management Extensions Storage Networking Industry Association
- Open--Grid Service Architecture and Infrastructure (OGSA&I)
- Web Services Security

2.4 Summary

In the last decade, managing and changing systems required human support, but in order to be practical, distributed applications must be able to adapt automatically, with minimal human intervention. Adaptation in most applications so far has been implemented in a fairly ad-hoc manner. The code that deals with adaptation is typically embedded in the application/program code. While this may be suitable for local adaptation, it is not viable/possible for distributed networked systems, which may require change in the application structure or need some kind of coordination, which is still complicated from the implementation view of running systems, and new problem solving tools and functionality.

The management of distributed software components within a heterogeneous, multi-organisational environment will be important for future distributed systems. However, current management techniques and standards have emerged from the communications world and are still focused towards the management of every hardware devices or connections. For example, SNMP provides a very low-level, variable oriented approach to management, which is analogous to remote debugging.

In this chapter we reviewed the background of the principles and definitions required in our research beginning with the notion of self-adaptive software and its categories, and its feature that can help us to develop our approach in the distributed system area. The following section provides an overview of autonomic computing concepts and requirements. Having identified the need for computing systems with self-management and control, we then presented an overview of autonomic computing and its main characteristics as a basis for such efforts as e-science and grid technology. This chapter addressed the main relevant background theories and principles used in this work.

Chapter 3

Distributed Computing Management

3.1 Introduction

Chapter Two presented an overview of the fundamental requirements to support runtime self-management of distributed applications especially in an unpredictable environment. This highlighted the increasing demand to understand the research and technological aspects relevant to supporting distributed application management and self-management in particular. In addition, a brief discussion of how to use these technologies as a platform for our study was presented and what types of extensions are required to facilitate safe and dependable applications self-management.

This chapter provides an overview of the Service-Oriented Development and Programming (SOP) approach, followed by a brief description of object-oriented middleware technology with a focus on Jini middleware services. This has been used in our proof of concept implementation of the proposed autonomic control middleware services. This is followed by a discussion of the limitations of current middleware technology specifically with regards to runtime change control for the safe and predicable self-adaptation of distributed applications.

3.2 Service-Oriented Development

Many technologists contend that SOP is the next programming revolution since Object-Oriented Programming (OOP) [28]. Since the introduction of Java™-based middleware - Jini technologies, SOP has attracted increased popularity within the web services community [29]. Based on the SOP model, the Openwings™ architecture has been refining the SOP model, which enables a new generation of service-oriented computing applications and adds the idea and the concept of modelling the programming problems and how to solve these problems.

The Openwings™ SOP model [28] shares a number of architectural elements with component-based software see Figure 3.1 below, namely:

- Contracts: a contractual interface that defines interactions between the components, the syntax and semantics of a single behaviour.
- Components: 3rd party reusable, deployable computing elements, which are independent of protocols, platforms, and environments. There are several architectural aspects important to service oriented computing.
- Connectors: encapsulates the details for a specified contract. It is an individually deployed element that contains a user proxy and a provider proxy.
- Container: the service that enables components execution, in the mean time managing their availability and the security of the code. Examples of containers in other architectures include EJB Containers (e.g. Enterprise Java Beans), Web Servers (e.g. contain servlets) and Web Browsers (e.g. contain applets).
- Contexts: that enables a deployable plug and play component that describes the details of installation, security and lookup.

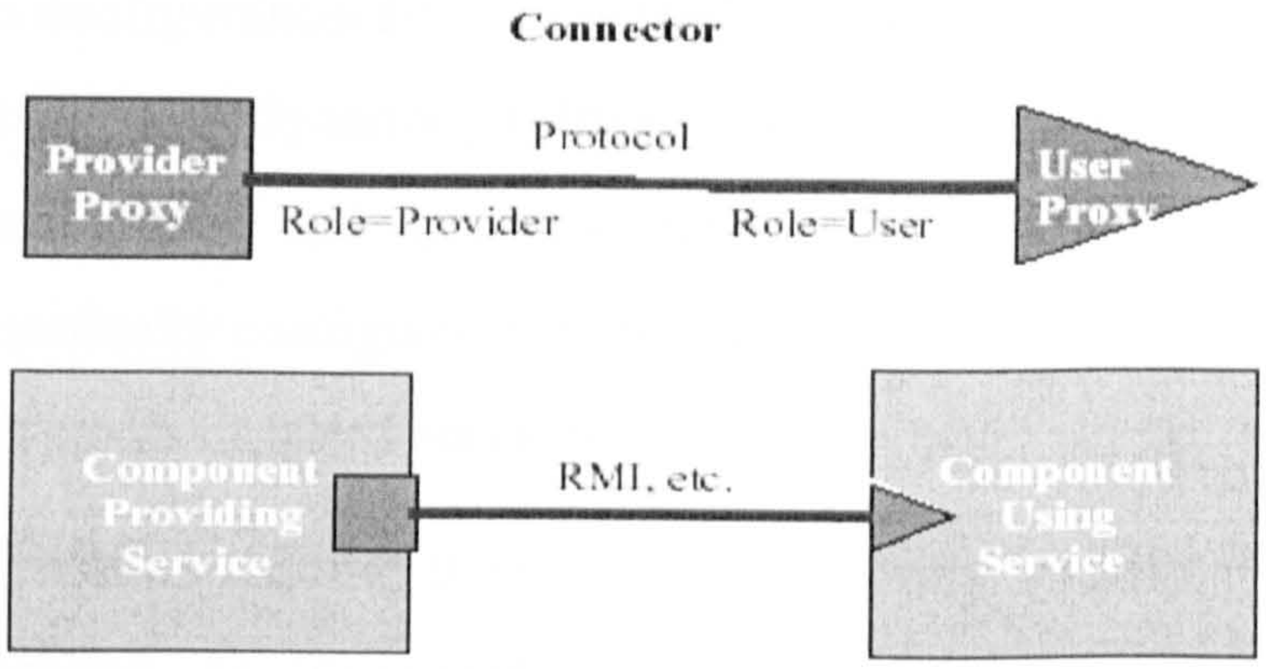


Figure 3.1: ADL[1] Concepts Architecture [28].

- Service: are component capabilities to other components and is defined by an interface that provides a semantic behavioural specification and a specific syntax definition. Examples of Interface Definition Languages (IDL) include Common Object Request Broker Architecture IDL and Microsoft IDL (MIDL).

[1] Architecture Description Language ADL is service-oriented programming modelling language, which contains all the aspects of components, connectors, roles, and ports.

The first three fully functional frameworks for SOP are Java™, Jini™, and Openwings™ (Fig. 3.2). Some of the pattern for describing SOP may be supported in a Java™ programming environment only and it will be very difficult to implement Service-Oriented Programming in other languages until these capabilities are provided to other programming languages[2]

Service-Oriented Programming Patterns		
Java™	**Jini™**	**Openwings™**
Contracts	Lease	Component
Mobility	Discovery	Connector
Code Security	Lookup	Container
	Service Security	Context
	Service User	Policy
	Interface	Proxy
	Transaction	Installer
	Coordinators	Management

Figure 3.2: Frameworks comparison for SOP support [28].

An essential aspect in a service-oriented architecture is how components locate services. Service location information is normally hard coded in software components, or saved in a configuration file that is read on start up by the components. In reality, a networked system is dynamic; software and hardware components are replaced or upgraded, nodes enter and leave the network thereby creating a large management problem for statically configured systems. These systems are simply not built to recover from network errors or failed services.

Solving this problem requires the use of two concepts, namely: i) discovery service and ii) service lookup, as components "discover" the environment in which they are deployed and dynamically "lookup" the services they need.

3.3 Distributed Middleware

Over the past decade, the adoption of Commercial-Off-The-Shelf (COTS) middleware products across the software industry has gained huge attention. The two key reasons

---

[2] The middleware technologies are supported for service-oriented architectures, including Sun's Jini Technology, the CORBA Trader Service, and Microsoft's Universal Plug'n'Play and the Ninja research project under development at UC Berkeley.

for this growth are Internet usages and the need to integrate heterogeneous legacy systems to streamline business processes.

Distributed middleware plays a very important role by providing functions and API that effectively bridge the gap between the network operating system and distributed application components and services. Middleware is defined as a set of services required for providing connectivity and management services in a distributed computing environment. These services include database connectivity, messaging, remote procedure calls, object request brokers, transaction services, timing services, and naming services.

3.3.1 Categories of Middleware

The main categories of middleware are described as follows:

- Distributed Tuples middleware, distributed relational databases offers the abstraction of distributed tuples, as its Structured Query Language (SQL) allows programmers to manipulate sets of these tuples (a database) with intuitive semantics and rigorous mathematical foundations based on set theory and predicate calculus. Linda is a framework offering a distributed tuple abstraction called Tuple Space (TS). Linda's API provides associative access to TS, but without any relational semantics. It offers spatial decoupling by allowing deposit and withdrawal processes to be unaware of each other's identities. Temporal decoupling is also offered by allowing them to have non-overlapping lifetimes.
- Remote Procedure Call middleware extends the familiar procedure to offer the abstraction of being able to invoke a procedure whose body is elsewhere on a network.
- Object-Oriented middleware provides the abstraction of an object that is remote yet whose methods can be invoked just like those of an object in the same address space as the caller. Distributed objects incorporate all of the software engineering benefits of normal object-oriented techniques such as encapsulation, inheritance, and polymorphism and make these available to the distributed application developer.
- Message-Oriented middleware (MOM) provides an abstraction of a message queue that can be accessed across a network and a generalization of well-known

operating system constructs such as the mailbox. Many MOM products offer queues with persistence, replication or real-time performance. MOM offers the same kind of spatial and temporal decoupling that Linda does.

Web Service Middleware: that is a middleware that enables and simplifies web application-to-application connectivity. Web services differ from other forms of middleware as it is based on XML standards, a user understandable form.

This section provides an overview of some of the main object-oriented middleware and web service middleware.

3.3.1.1 CORBA Middleware Technology

The Common Object Request Broker Architecture (CORBA), developed by OMG, is an open and vendor-independent solution to enable distributed application networking. Here, using the Internet Inter-Orb Protocol (IIOP), a CORBA-based program running on distributed heterogeneous[3] hosts can interoperate with a CORBA-based program from the same or another computer on almost any other computer, operating system, programming language and network. CORBA protects applications from heterogeneous platform dependencies. CORBA defines interfaces, not implementations. It simplifies the development of automated distributed applications, by encapsulating the following [30]:

- Object location.
- Connection and memory management.
- Parameter de/marshalling.
- Error handling and fault tolerance.
- Object/server activation.
- Concurrency.
- High confidence.
- CORBA Architecture.

As shown in Figure 3.3 below, CORBA specifications group objects into four categories, namely [30]:

[3] Running on any computer on almost any platform, operating system, programming language, and network.

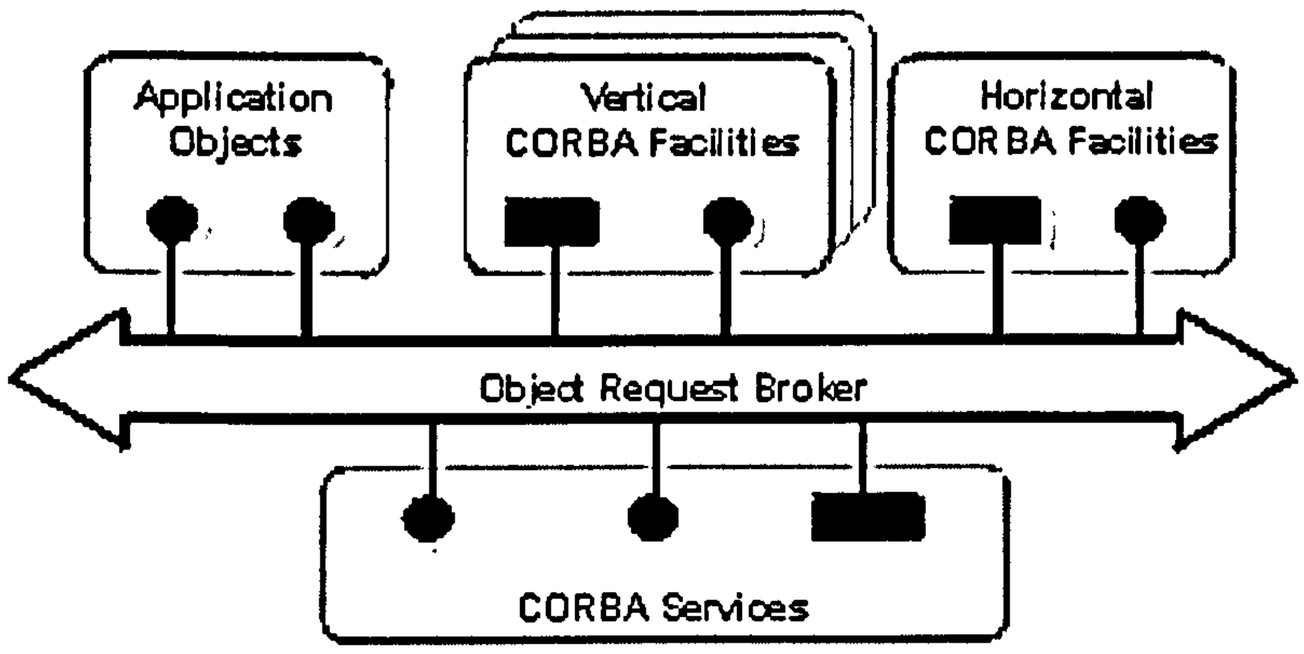


Figure 3.3: The OMG-CORBA Architecture [30].

- CORBA services™: which provide basic performing functionality for distributed object applications. Such services may provide system library calls, for example, larger services, transactions and security. This is included in the naming service, object trader service and new persistent state service.
- The Horizontal CORBA facilities sit between the CORBA services and the Application Objects (see below) and are facilities that are potentially useful across business domains. There are four horizontal CORBA facilities: The Printing Facility, the Secure Time Facility, the Internationalization Facility and the Mobile Agent.
- The domain CORBA facilities: IDL allows a standard interface to be defined for standard objects that every company in an industry can share.
- Application objects are provided in most parts of the CORBA architecture. Since they are typically customized for each individual application and do not require formalization, this category identifies objects that are not affected by OMG standardization efforts.

CORBA applications are composed of *objects*, which are individual pieces of running software that combine functionality and data. For each object, an interface OMG IDL [31] is defined. The interface is the syntax part of a contract that clients can invoke when offered by the server object. Any client can invoke an operation it wants to perform and to marshal the arguments that it sends. When the invocation reaches the specified object, the same interface definition is used to unmarshall the arguments so that the object can repeat the requested operation with them. The interface definition is then used to marshal the return results, and to unmarshall them when they reach their target [31].

The IDL interface definition is independent of any programming language, thus OMG has standardized mappings from IDL to a range of programming languages including; C, C++, Java, Smalltalk, COBOL, Ada, Lisp, Python and IDL script [31]. Clients can access objects only through their IDL interface, invoking only those operations that the object exposes through its IDL interface along with input and outputs parameters, which are included in the invocation.

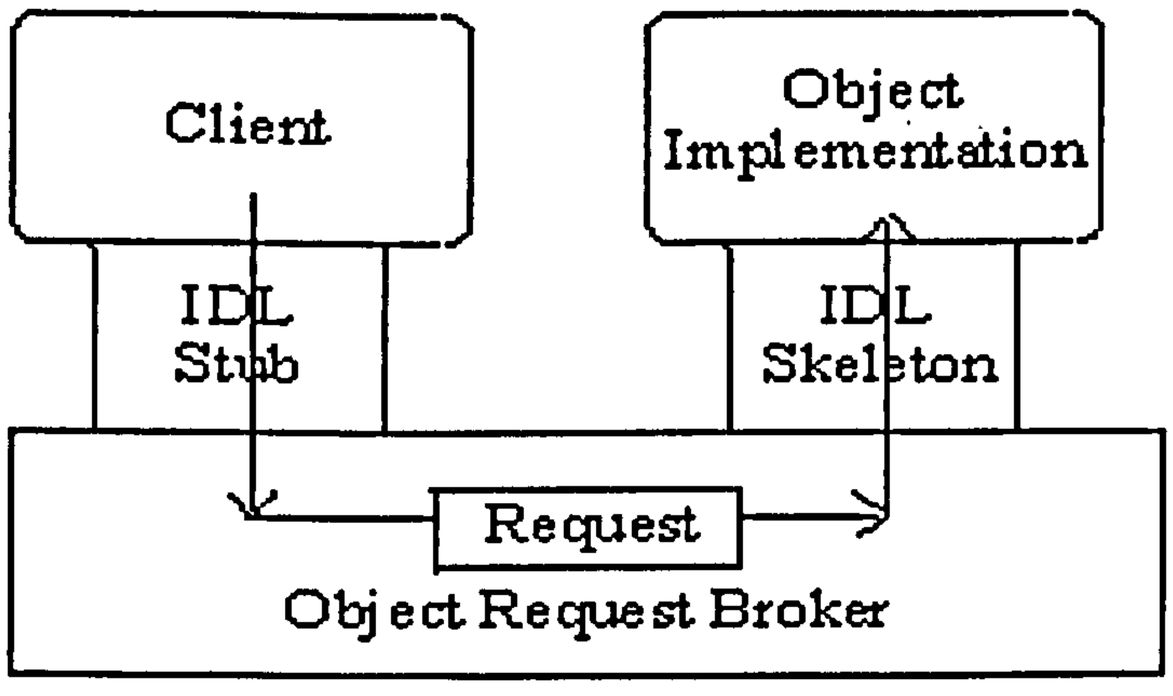


Figure 3.4: A request passing from a client to an object implementation [31].

When the IDL is compiled, both client stubs and object skeletons are automatically generated. Because IDL defines interfaces, the stub on the client side operates with the skeleton on the server side, even if the two are running on different ORBs or are compiled from different programming languages (see Fig. 3.4 above). Every object instance has one unique object reference that is used from the client side to direct their invocations and identifying to the ORB the exact instance for invocation [32].

CORBA also supports a remote invocation protocol to facilitate remote object invocation. As illustrated in Figure 3.5 below, this process works on two levels [31]:

1. The client knows the type of object it is invoking and the client stub and object skeleton are generated from the same IDL. This means that the client should know exactly which operations are required for invocations, what the input parameters are, and where they have to go in the invocation. As soon the invocation reaches the target, all the required data is to hand.
2. The client ORB and object ORB must agree on the same protocol to specify a target object, operation, input and output parameters of every type that they will use and how these are represented over the wire. OMG has defined the standard protocol IIOP.

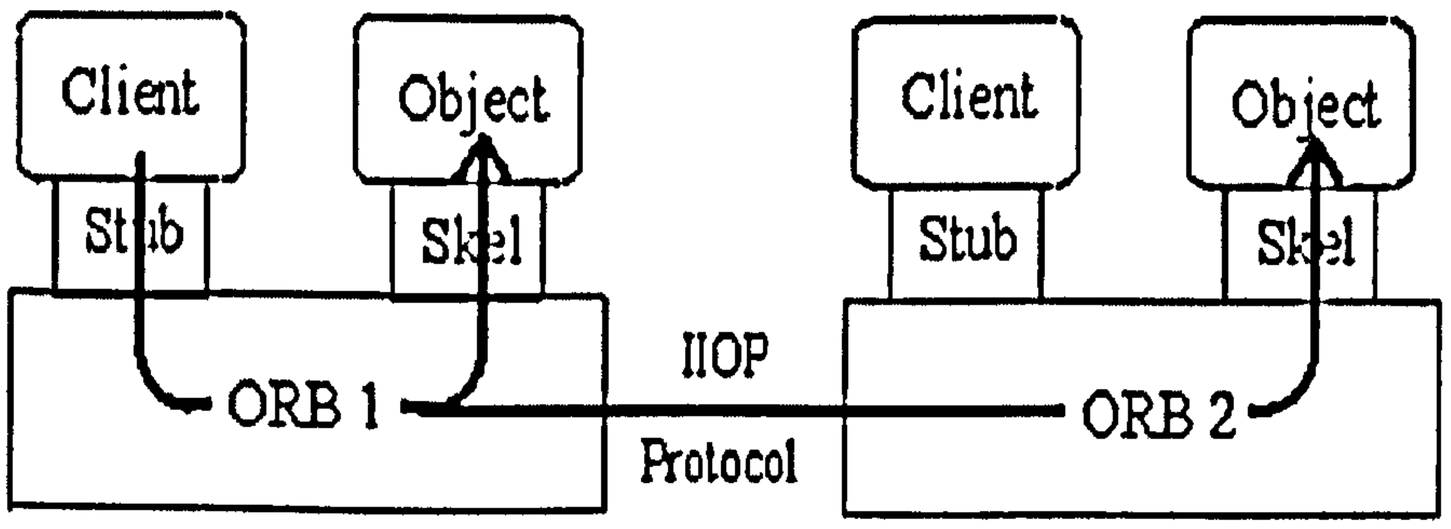


Figure 3.5: Interoperability using ORB-to-ORB Communications [31].

3.3.1.2 Universal Plug and Play

Universal Plug and Play (UPnP) is an architecture for peer-to-peer network connectivity for PCs, intelligent devices and other networked devices. UPnP has been generally targeted at home networking and is built on Internet protocols including; TCP/IP, HTTP and XML, enabling devices to automatically connect to each other and work together. UPnP is independent of any particular operating system, programming language or physical medium [33].

Network Address Translation (NAT) is an Internet Engineering Task Force (IETF) standard used to allow multiple PCs or devices on a private LAN. This translates a private IP address and port number to a public IP address and port number, tracking those translations to keep individual sessions unimpaired.

Each application must use a public address and a unique port number for each session. Large organizations have professional staff to handle their corporation applications with NAT, unlike smaller organizations or people at home. So UPnP-NAT can automatically solve many of the problems that occur with applications, making this an ideal solution for small businesses and home appliances. Therefore, UPnP mainly supports home appliance consumers and small businesses and organizations [33].

UPnP has common protocols and procedures to guarantee interoperability among network-enabled PCs, appliances, and wireless devices [34].

UPnP defines a set of common protocols/services that devices can use to join a network and describe themselves and their capabilities, enabling other devices and users to use it without a complex set up or configuration. The six main layers of UPnP architecture, as shown in Figure 3.6 below are [33].

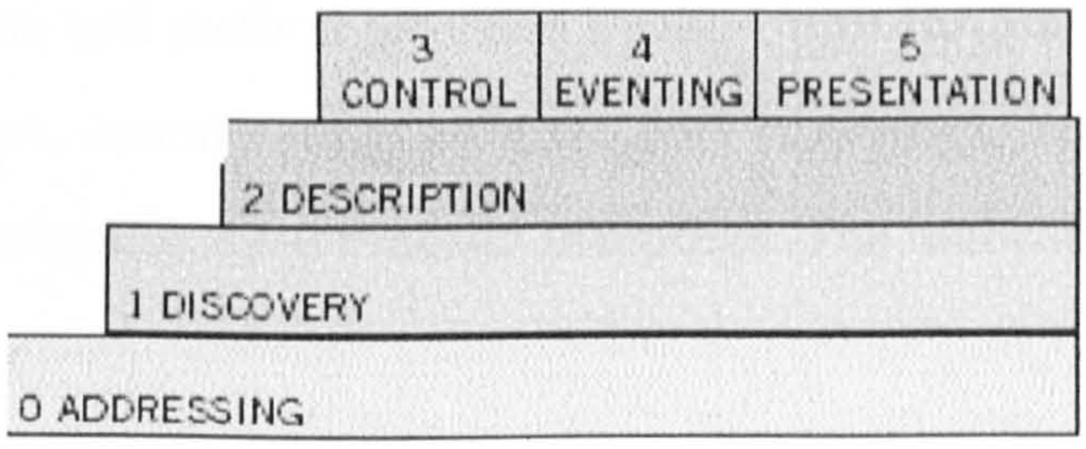


Figure 3.6: UPnP Function's Layer [35].

- Device addressing.
- Device discovery.
- Device description.
- Action invocation.
- Event messaging.
- Presentation, or human interface.

For data transmission, UPnP does not move byte codes or use ActiveX controls. It has an independent operating system but is based on various network standards, in particular, peer-to-peer or ad-hoc networking. Devices can use a Dynamic Host Configuration Protocol (DHCP) server or Auto IP (Internet Protocol) to automatically select an IP address from a range of other addresses [33].

As soon as the device connects to the network or becomes online, the device describes itself using TCP/IP for network control point communication (the *control point could be an operator's station or just another device on the network.*). Control points can discover devices by searching through the entire network. The device, using XML, describes itself and its service and can be initiated by receiving messages from the control points. Devices send events to control points to subscribe to a device's event. This is called "*event messaging*" and holds a report of the status of each device. Also, it

sends the presentation or Web home page to the control point as a part of the device description.

The main function and method provided by the UpnP-NAT are, for example, learning a public IP address, enumerating existing port mappings, adding and removing port mappings and assigning lease times to mappings. The real cases that UPnP Nat can use include [35]:

- Multi-player gaming
- Peer-to-peer connections
- Real time communications
- Remote Assistance (a feature in Windows XP)

3.3.1.3 Web Services Middleware

Web service is middleware that enables and simplifies web application-to-application connectivity. Web services differ from other forms of middleware by being based on XML standards. In principle, these standards will create hub-and-spoke configurations, rather than the so-called spaghetti code that results from point-to-point connectivity. In addition, the computing services offer enabled web-service through the web and are accessible from any enabled machine with Internet access.

Web services have very important characteristics that are essential to an e-business environment such as:

- Enable interoperability through a set of XML-based open standards.
- Enable self-contained business functions that are written to strict specifications to work with other similar kinds of components and with each other. Most of the established functions at this stage are messaging, directories of business capabilities and descriptions of technical services but there are other functions as well.
- Enable systems in different companies to interact easily with each other. In businesses companies, close cooperation with suppliers and customers is required, engaging in more joint risk and short-term marketing alliances, pursuing opportunities of business, and facing the prospect of more mergers. Companies need the capability to link up their systems quickly with other companies.

- Enable companies with the capability to do more business electronically, with more potential business partners, in different ways and at suitable cost.

For example, a company can allow their supplier the ability to see all the products levels the supplier provides. Consequently, the supplier can re-supply without the need for separate purchase orders. Suppliers could build on the basic features provided by web services messaging and service-description functions for this kind of electronic relationship and hence provide better services to the customer. Companies could extend these capabilities to other trading partners.

The foundation of the Web services standard is XML. There are three critical legs of Web services, all of which are derived from XML namely; Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL) and Universal Description, Discovery and Integration (UDDI) [36] although all are still in development.

Businesses use the XML-based Web Services Description Language (WSDL) and Web Services Flow Language (WSFL) to describe their Web services on the Internet and list them in an XML-based registry such as the Universal Description, Discovery, and Integration (UDDI). This allows available Web services to be found (see Fig. 3.7 below). A client sends a request for a service to the registry, which tells the client about the registered services that suit the client request. The Simple Object Access Protocol (SOAP) is then used to communicate (using HTTP and XML as the exchange mechanism) between the applications running on different platforms [36].

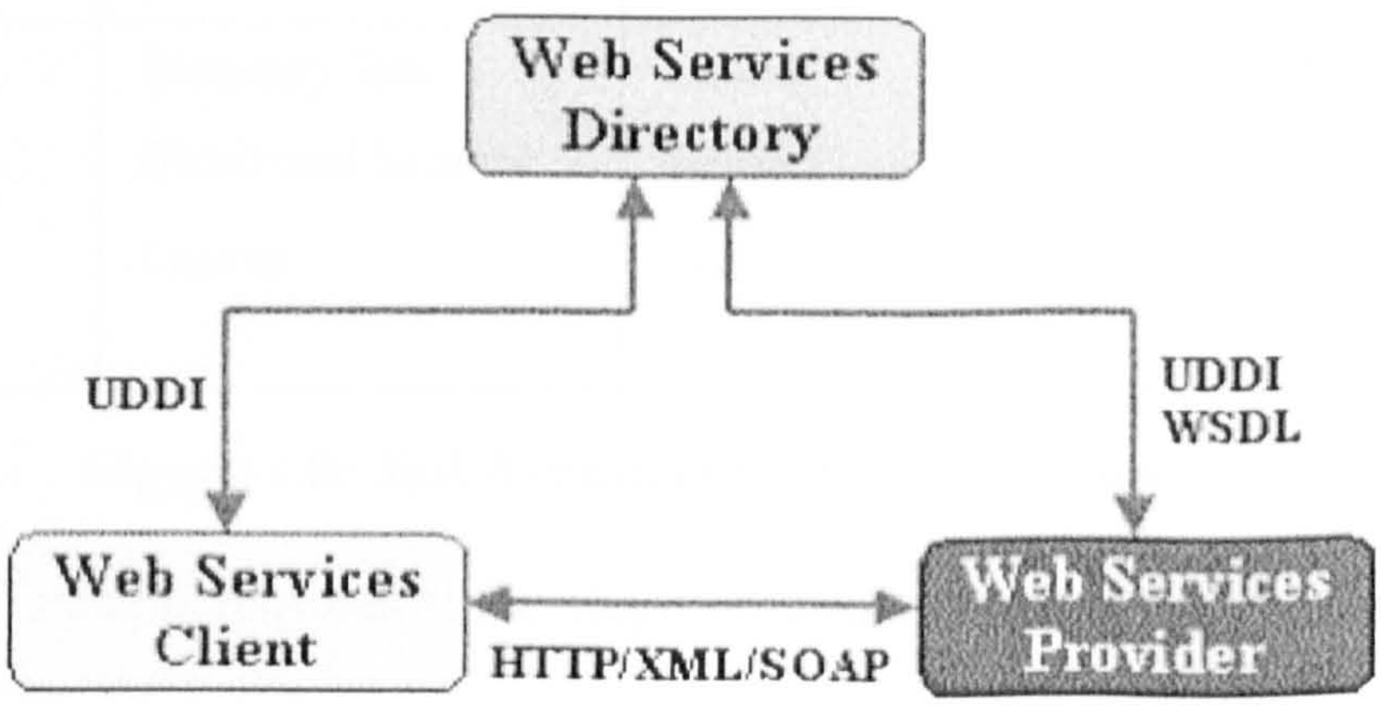


Figure 3.7: Web services protocols [37].

The relevance and importance of each of the following technical benefits will vary greatly from company to company, application to application and implementation to

implementation. If all of these factors are considered together, good results may be obtained from using the Web Services. The benefits may be considered as [36]:

- Software development automation.
- Streamlining middleware technology.
- Use of standards-based integration.
- Integration with applications and business process management.
- End of duplication of software code, leading to reusability.

3.3.1.4 Jini Middleware Technology

Jini is a Java-based middleware that provides an Application Programmer's Interface (API) and programmers may write services and components that make use of its core middleware services. A Jini system or federation is a collection of clients and services that communicate using Jini protocols [38], where Jini applications are often written in Java and communicate using the Java Remote Method Invocation (RMI) protocol. Although Jini is written in pure Java, neither clients nor services are constrained to be pure Java. They may include native code methods, act as wrappers around non-Java objects or even be written in some other language altogether.

	Infrastructure	Programming Model	Services
Base Java	Java VM RMI Java Security	Java APIs JavaBeans ...	JNDI Enterprise Beans JTS ...
Java + Jini	Discovery/Join Distributed Security Lookup	Leasing Transactions Events	Printing Transaction Manager JavaSpaces Service ...

Figure 3.8: Jini Architecture Segmentation [39].

Running a Jini system requires three main components (see Fig. 3.8 below), namely [38]:

- *A service*, such as a computation or storage service, etc,
- *A client* that would like to make use of this service and,

- *A lookup service* (service locator) that acts as a broker/locator between services and clients [38].

A further major component is the network connecting all three of these and this is generally implemented using TCP/IP. The dynamic nature of a Jini system enables services to be added or removed from a federation at anytime according to demand, need, or the changing requirements of the distributed applications. In addition, as shown in Figure 3.8 above, components of the Jini architecture may be considered in three further categories [39] namely;

1. Infrastructure: The Jini infrastructure is the set of components that enable the building of a Jini federation system and defines the minimal Jini core. The infrastructure is composed of the Java RMI protocol [40], which enables objects to communicate through Java RMI and a lookup service (i.e. lookup provides a central registry for services). The local representative of the remote entities that are involved in the invocation process is a java RMI concept based *proxy*. *Stubs* are the client-side proxies, while *skeletons* are the server-side proxies. A stub implements the same remote interfaces as the remote object it is representing and forwards the received method invocations from clients to the suitable skeleton. Skeletons wait until they receive the remote method invocations and then dispatch them to their remote objects. Stubs and skeletons look after all the low-level details of communication between clients and servers. Services are discovered and registered through the lookup service, which enables the registration of proxies for them. In particular, a lookup service maps the interfaces that indicate the functionality provided by a service to sets of objects that implement that service. In addition, descriptive attributes associated with a service allow a convenient selection of services based on humanly understandable properties.
2. Programming model: as defined in the Jini specification, this is a set of interfaces that enable reliable service construction, including services that are part of the infrastructure (e.g. the lookup service) and those that join the federation. The programming model is based on three distinct paradigms for distributed computing. These are event notification,

leases and transactions as detailed below [38]. The Event Notification, which allows clients to register interest in being notified of particular messages (i.e. the notification interface) and supports asynchronous, one-way delivery of such notifications. If a particular service wishes to support subscription on a notification event, it must support the notification interface to manage these subscriptions. A service that wishes to receive notification messages must implement the notification interface, which is used to deliver a notification event. To start notification from a particular service, one invokes the subscribe operation on the notification source interface. The Lease Interface extends the Java programming model by adding the notion of time for holding a reference to a resource, enabling references to be reclaimed safely in the face of partitions. For example, as registrations in the lookup service are leased, a service must periodically renew its leasing registration, otherwise its proxy is removed when the lease expires. The Transaction interface introduces a simple object-oriented protocol enabling Jini services to coordinate its state changes. The Jini transaction protocol differs from existing transaction interfaces. The Jini transaction specification has identified the basic components of a transaction, such as transaction clients, transaction managers, and "*participants*". All interactions between clients, transaction managers and participants are based on the Java RMI protocol. Transaction clients start a transaction by contacting a transaction manager through a proxy. The proxy is obtained by requesting the lookup service for a service that implements the manager interface. The transaction manager responds with a transaction object, which will represent the transaction in subsequent communications and contains information such as an identifier for the transaction and a proxy for the transaction manager. Clients then start to interact with participants by communicating with the object and the operation requested such as commit/abort. The participants use the semantic object to communicate with the transaction manager. The transaction manager is responsible for the consistent execution of the operations and ensures that all participants subsequently know if they should commit or abort them.

3. Services: A Jini system consists of services that can be collected together to perform a particular task. Services may allow the use of other services and a client of one service may itself be the service of other clients. The dynamic features of a Jini system enable services to be added or removed from a federation at any time according to demand. Jini systems provide mechanisms for service construction, lookup, communication and use in a distributed system. Services in a Jini system communicate with each other by using a service protocol, which is a set of interfaces written in the Java programming language. In addition, Jini-based systems define a small number of such protocols that consequently define essential service interactions. We will focus on two critical categories, identified by Newmarch [38] and used in this research, Lookup and Discovery Services: The heart of the Jini system is a group of three protocols called discovery, join and lookup (see Fig. 3.9 below). The first two of these, discovery and join, occur when a device is plugged in. Discovery occurs when a service searches for a lookup service to register. Join occurs when a service has located a lookup service and wishes to join it. Lookup then occurs when a client or user needs to locate and invoke a service described by its interface or attribute type.

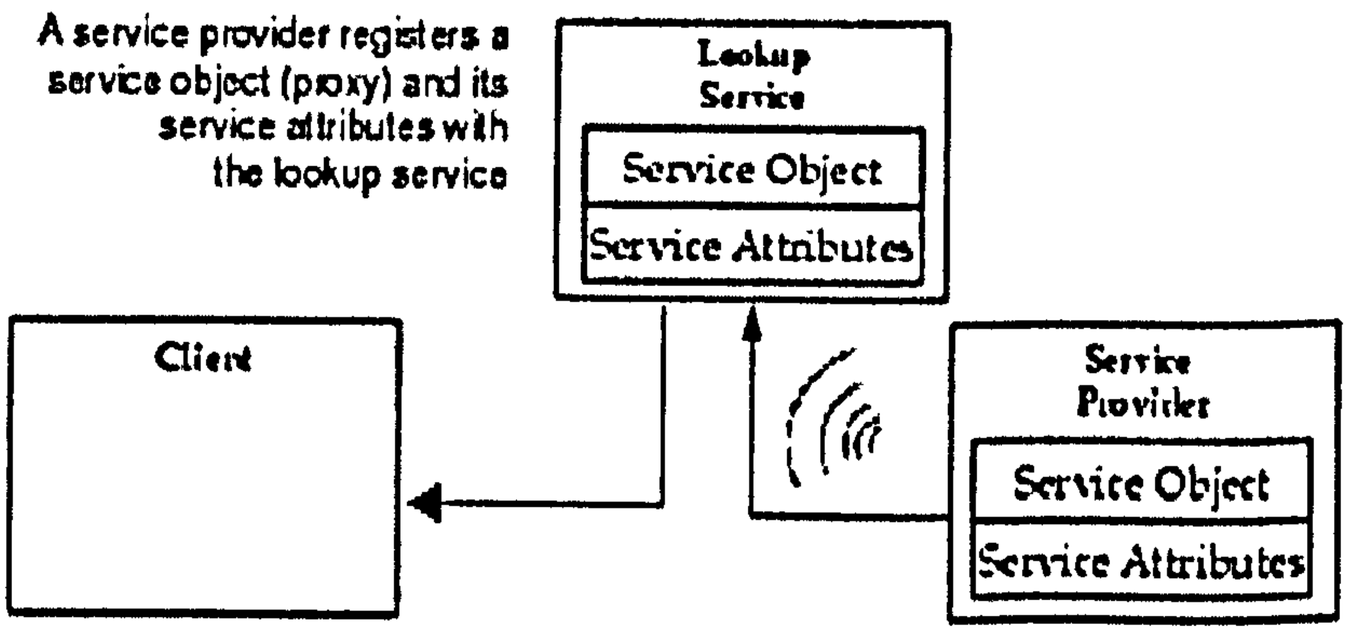


Figure 3.9: Lookup, Discovery and Join [40].

From the service's client point of view, there is no distinction between services that are implemented by objects on different distributed machines or on one machine, as services are downloaded into the local address space. All of these services should appear to be available on the Jini network as objects written in the Java programming

language and another could replace one kind of implementation could be replaced by another without change or intervention by the client.

3.3.1.4.1 JavaSpace Services

JavaSpace services are a platform for the design and implementation of distributed systems and provide tools for building distributed protocols. They are designed to work with applications that can be modelled as flows of objects through one or more servers, which can provide a reliable distributed storage system for the objects, as it is the abstraction of distributed tuples called Tuple Space (TS). Linda's API provides associative access to TS, but without any relational semantics. Linda offers spatial decoupling by allowing depositing and withdrawing processes to be unaware of each other's identities. It offers temporal decoupling by allowing them to have non-overlapping lifetimes (see Appendix A).

JavaSpaces technology provides many benefits. If an application can be modelled in this way, it provides a reliable storage strategy that guarantees that entries written to the server will not be irrecoverably lost and could retrieved again. An implementation of the distributed transaction mechanism is also provided.

A JavaSpace service holds entries; an entry is a typed group of objects expressed in Java. There are a number of operations, namely: *Write, Take, Read,* and *Notify*, These are detailed as follows (see Fig. 3.10 below) [40]:

- Write: A write request creates a copy of that entry in the JavaSpace that can be used in future lookup operations. Looking up an entry requires the use of *templates*, which are entry objects that have some or all of the fields set to specified values that must be matched exactly. Remaining fields are left as wildcards. i.e. fields that are not used in the lookup.
- Read: A read request to a space returns either an entry that matches the template or an indication that no match was found.
- Take: A take request operates like a read, but if a match is found, the matching entry is removed from the space.
- Notify: A JavaSpace Services notification occurs when an entry that matches a specified template is written, this is done using the distributed event model.

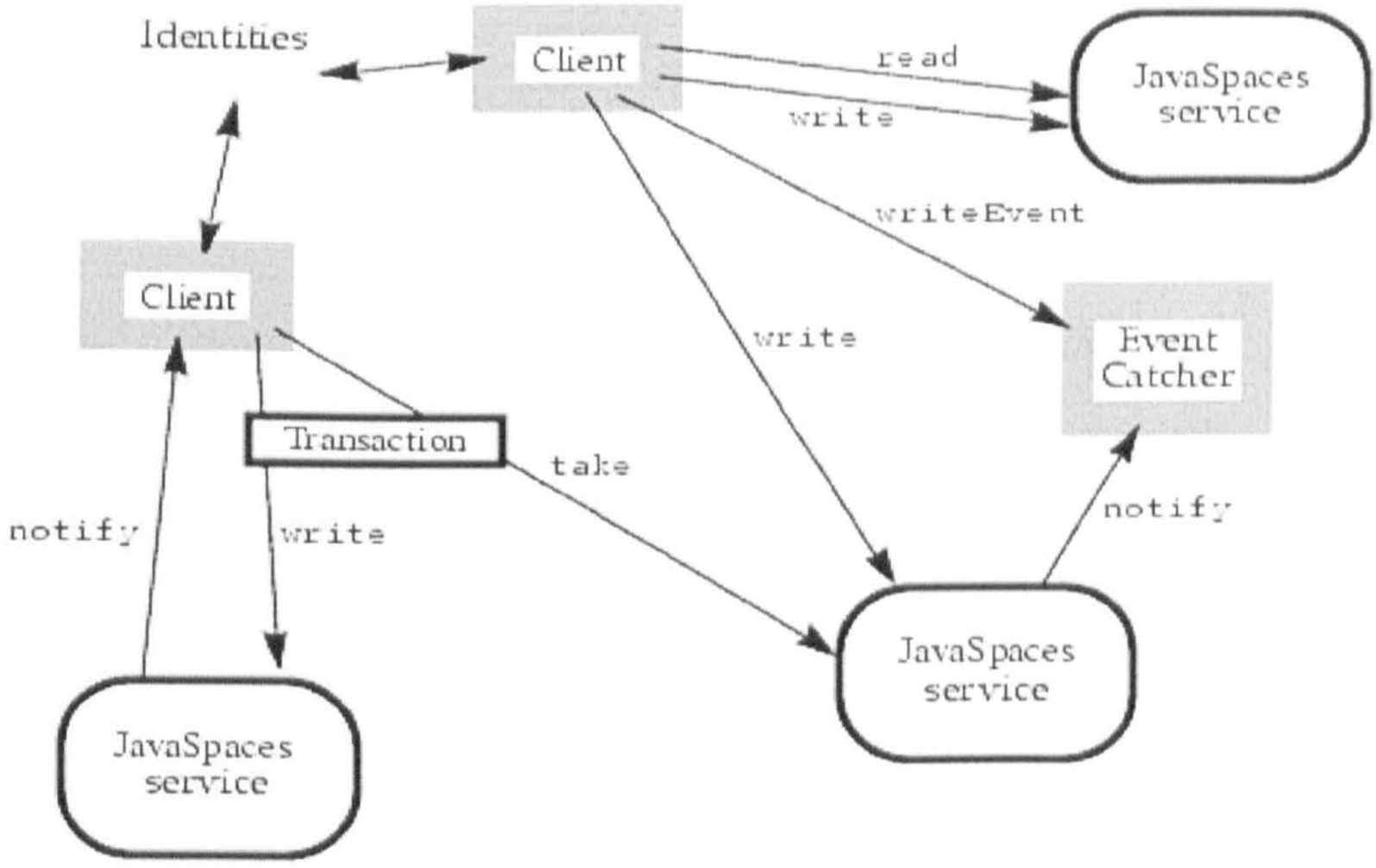


Figure 3.10: A JavaSpace Application [40].

Clients perform operations that map entries or templates onto JavaSpaces services, as operations take place. A client can interact with as many spaces as it needs and identities are accessed from the security subsystem and passed as parameters to method invocations. Event Notifications are sent to event catchers, which may be clients themselves or proxies for a client.

3.4 Self-Management Requirements for Distributed Applications

Modern distributed systems are becoming increasingly complex; containing a large number of remote objects and must be capable of monitoring, faults self-detection, self-evolution and self-management to match changing requirements without shutting down the complete system. However, middleware technologies used for remote object invocation simplify the runtime management of distributed application by bridging the gap between the network operating system and distributed components and services. The runtime management of distributed applications is however difficult because of the

many different possibilities that may arise from the runtime environment, leading to inconsistencies, conflicts and exceptions.

Addressing the problems of distributed application management requires enhancing existing management approaches to enable the distributed application itself to find an appropriate solution strategy and deploy the change description to the running system. Some of prevalent needs of such distributed systems are listed below:

- Suitability for both distributed operation within an application and the use of generic services across applications, by adding the capability to support software developers, systems integrators and coordinators.
- Agreement and/or conformity, i.e. should consider the existing Internet infrastructure as much as possible.
- Tolerance of failures, networks where nodes are very tightly coupled often suffer catastrophic failure when one node goes down. This is a serious problem for heterogeneous networks.
- Strong support for general software lifetime management tools for users.

3.5 Summary

Distributed applications development, deployment and management are taking place in enterprises. Middleware products are developing robust feature sets and object-oriented technologies are exhibiting increasing standards and features. As a result of these changes, effective distributed applications management and control becomes ever more. However such control should provide the distributed application with robust performance and efficiency without disturbing running system

This chapter has addressed the major aspects that are required for the management of distributed computing; in particular the aspects that are used as a basis for our approach has been considered. Object-oriented middleware and Service-Oriented Programming (SOP) have been introduced in this chapter as a foundation for robust and dependable distributed application development and management. Although software patterns availability and object mobility is reduced management time however, this may not be the case in the future, there are many factors that cause problems for the management of distributed application and must be taken in account. In such a networked system environment, dynamic software and hardware components are replaced or upgraded, nodes enter or leave the network that could cause run time problems. This generates the

need for new concepts that provide the system with two main facilities, i.e. i) discovery service and ii) service lookup. So components could "*discover*" Their environment and dynamically "lookup" the services they need. Distributed middleware technology has added such features to aid the management of distributed computing as, CORBA, UpnP, Web services, and Jini middleware technology. Jini™, our chosen middleware provides the additional advantage of the distributed middleware. Distributed shared spaces (i.e. JavaSpace service), JavaSpace is a composite of a synchronization construct and an object database, which allows transactions to be written, taken and read from a shared space by the objects.

Although The technologies reviewed in this chapter are already in existence, a new approach capable of integrating self-monitoring, self-detection, self-evaluation and self-management is required to fully realize the autonomous management of distributed computing, i.e. to match changing requirement without human intervention and without at shutting down the system.. Such an approach should serve to minimize system complexity for users as systems correct themselves in the event of failure by autonomic management of computing systems.

Chapter 4

Literature Review

4.1 Introduction

Traditional software lifetime management, including software evolution activities have been mostly conducted[4] at maintenance-time requiring the shutdown of a software system (or a large part of it) to enable software engineers to undertake the required maintenance, update and/or evolution of the considered software system.

Much research work exists that focuses and/or contribute to enhanced methods and techniques to support the lifetime management and self-governance of distributed applications, including; system monitoring, conflict detection, conflict identification, solution generation and evolution, system adaptation and reconfiguration and coordinating and managing the interdependencies among services.

This chapter provides a literature review of such work together with related research work. The review will be structured into two main parts covering research relevant to; (i) the static management of distributed systems, and (ii) the dynamic management of distributed systems.

4.2 The Static Management of Distributed Systems

This section considers model-based approaches to systems management, namely: conflict resolution and coordination approaches, strategy and plan-based representation approaches and exception-handling approaches.

4.2.1 Conflict Resolution and Coordination Approaches

Conflict resolution can be characterised as an event-driven process, which is activated as soon a conflict, error and/or a system's model inconsistency is detected. Where a

[4] And remain so far a large proportion of today's complex software systems.

system's model inconsistency can be described as "... *the situation in which two descriptions do not satisfy some relationship that should hold between them...*" [41].

From a requirements engineering perspective, Bashar *et al.* [42] described the constituent processes of inconsistency management as encompassing four major steps: *monitoring* for inconsistency, *diagnosis*, *handling* and *monitoring the outcome*. These processes provide great flexibility for selecting appropriate inconsistency handling actions and system coordination. In all this, coordination is used to ensure that every actor [29] acts in accordance to a defined plan of action, thus preventing any delay, deadlock and/or any other system performance degradation [43].

Other work focusing on high-level strategies for conflict resolution presented a set of strategies such as; *negotiation, arbitration, voting*, and *independence*, each of which is outlined in the following sub-sections.

4.2.1.1 The Negotiation Method

Negotiation is commonly used in multi-agent or any distributed systems to support conflicts resolution processes and can be defined as;

> *"...the term used in distributed problem solving research to denote the process by which autonomous nodes coordinate their views of the world and act and interact to achieve their goal ..."[44]*

During such a process, agents exchange constraints associated to their own intent and/or performance requirements. For example, agents send their requests to other agents and receive either confirmation or explanation of the constraints, including their goals and intents, which can be used by the requestors agents to relax their constraints. The process continues until both parties converge on a mutually acceptable request. Cooper and Taleb-Bendiab [45] grouped negotiation mechanisms as follow:

- Mathematical model-based systems: these are generally based on game theory and economic behaviour. The majority of developed systems have used quantitative models including; multi-criteria decision making, conflict analysis, group decision theory, multi-objective linear programming and fuzzy arithmetic in searching for an optimal solution based on the negotiation criteria and preferences provided by the user [45].

- Heuristics model-based systems: that have been developed using AI techniques to support multi-agent negotiation behaviour. For instance, Sycara et al. [46] reported on their proposed hybrid negotiation model, which combined case-based reasoning and multi-attribute utility theory to generate solutions for a given negotiation process. However, this approach did not consider the potential impact of change and more specifically, subsequent conflicts emerging as a consequence of the conflict resolution process.

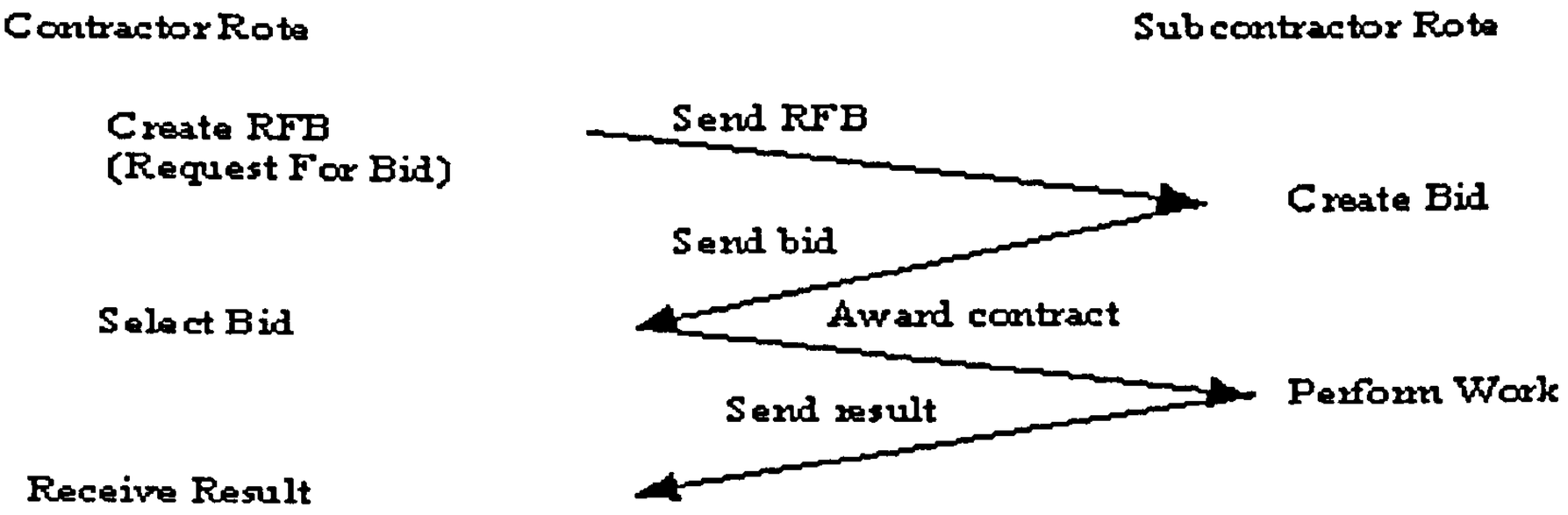


Figure 4.1: A simple version of the Contract-Net protocol [47].

The negotiation approach has also been used to underpin the search process and process delegation. For instance, Lander and Lesser [48] proposed a "negotiated search" algorithm for multi-agent systems, which is, for instance, triggered by agents search for a given resource, service provider and/or task brokerage. Klein *el al.* [47] described their use of the Contract-NET protocol to support the negotiation process for distributed control architecture. The Contract-NET protocol has often been used as a high-level coordination protocol for multi-agent task brokerage and delegation. For instance, as shown in Figure 4.1 above, an agent (contractor) identifies a task that it cannot do itself and tries to search for another agent (sub-contractor) that can, For this, the agent initiates a Request For Bids (RFB) describing its intended task. Sub-contractors respond by sending a Bid and then the contractor selects the "best" bid and contracts the task to the selected sub-contractor. The latter will undertake the task and return the results in accordance with the contract specifications (Fig. 4.1). Different implementations of the protocols exist either using a blackboard architecture [47] or distributed shared space such as; Linda or JavaSpace (Sec. 3.3.1.4.1, [49]).

Other descriptive negotiation models have been proposed, including the Rosenschein and Zlotkin [50] model, which provides a taxonomy of negotiation processes required to support conflict resolution, namely [50]:

- Detect conflict: by using a service's constraints for comparing its goals (i.e. what should be done) with what is actually done.
- Identification of exceptions and select the appropriate handler, in order to handle exceptions that have been thrown.
- Negotiation team formation/reformation, to identify an agent/service required for executing a selected plan.
- Solution Generation, which generates an ideal solution for arisen conflicts.
- Solution Evaluation, which evaluates a previously generated solution.
- Negotiation Monitor, used to gather the required information to support the decision made in the negotiation control phase.
- Negotiation Management/Control is a management mechanism to resolve the detected conflict.

Based on the previously required taxonomy of the negotiation process, Cooper and Taleb [51] proposed and developed a computational framework to support a high-level control mechanism for multi-agent systems, which was applied to supporting distributed design negotiation. The computational framework used a mixed-mode initiative approach to enable the delegation and adjustment of control authority between agencies and human users -thus providing a model for adjustable autonomy. As shown in Figure 4.2 below, a brief description of the control mechanism is as follow [52]:

- *Human and computer interface:* The interface provided for a user to input control constraints and required information easily.
- *Control Profiles:* A system library of control information containing three sub-libraries:
    - Conflict resolution strategies: Which a set of strategies and plans for resolution the conflict.
    - Control gate: This observes and alters the plan when necessary.
    - Control Preferences: Used to adjust the control profiles behaviour.
- *Planning system:* The most important module in the mechanism that executes all the tasks in the control mode and contains a script interpreter, matching plan list, plan selection and control resources.
- *Control resources:* where the plans/strategy are arranged as a system call library, plan library, strategy library and profile library.

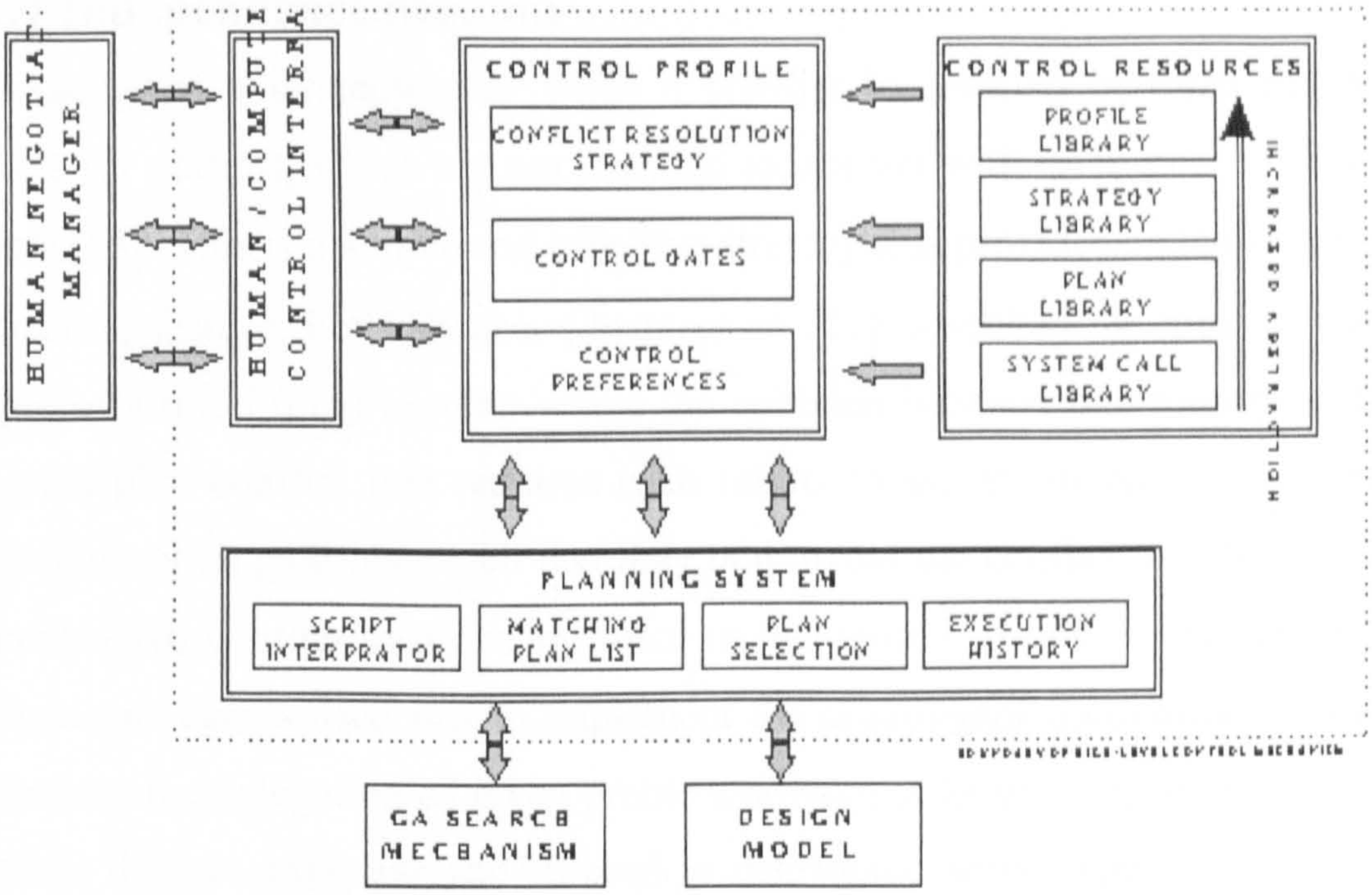


Figure 4.2: A high-level control mechanism architecture [52].

4.2.1.2 The Arbitration Method

Rosenschein and Zlotkin [50] have defined the arbitration method as a process in which conflicts are arbitrated by a third party (arbitrator), who has no direct power to modify the conflict behaviour. The arbitration decision must be agreed by the conflicting service/agents. However the arbitrator being equipped with the authority and solution-search capabilities has more knowledge than any other agents involved in the conflict. This authorization should permitted by the agents in the organization structure [50].

4.2.1.3 The Voting Method

Voting is a conflict resolution method adapted from the human organization research field, and views authority as being distributed in a society. Group Membership Problems (GMP) applied this approach in a distributed system environment to elect the new agent/service to assume managerial duties when the previous manager loses its functions and cause conflicts [53] . Ephrati and Rosenschein [54] used voting to develop maximum social welfare, where a voting for preferences and maximum candidates serves to maximize total satisfaction. This approach was used as a basis for incrementally constructing a plan that brings the society of agents to achieve a maximal social welfare states [55].

4.2.1.4 Independence Method

Independence is a strategy used when a conflict is detected between independent members (or parties), which are not required to interact with each other to solve their conflicts [56]. This is a simple and effective strategy that provides systems with a self-modification feature. For instance, Chang *et al.* [57] described the method using the example of a multi-robot system, where the collision between two robots results in a robot path plan conflict that requires each robots to initiate independently, the self-modification of its path plan such that they both avoid the conflict -- collision. Lander [58] considered conflict as a driving force in the control of distributed-search among heterogeneous agents used this to implement the multi-agent framework TEAM [58]. This enables the delegation of given problem-solving tasks to independent agents with individual domain expertise and through a distributed search approach, enables the generation of individual solutions for each of the assigned agents' tasks. These are then integrated into an overall solution to the considered problem. In addition, TEAM enables agents to communicate and coordinate with each other through a shared memory. The idea of the shared memory is similar to our distributed shared space and used for the same purpose, namely, the coordination process.

4.2.2 Strategy and Plans Representation Approaches

As defined by Barber et al. "*A strategy is an abstraction that the agent can use to encapsulate the coordination mechanism used for any of the core problem-solving tasks* ..." [59]. For instance, a strategy can be employed by an agent to select the "*best*" actions for solving a given problem, and/or achieving its desired goals. Barber *et al.* [59] proposed a representation model for conflict resolution strategies. Their representation model is informed by the general conflict resolution requirements [60] , including;

- The strategy requirements: " *strategies may make use of different parts, or may also place constraints on the reasoning capabilities*".
- The strategy execution cost: the execution of each strategy differs according to difference resources that have been used. For example, "*some strategies may require a larger number of messages or a longer time. It is important to consider this factor when dealing with deadlines or limited resources*".
- The solution quality: possibility of multiple solutions and therefore each solution has different qualities depending on the use of different strategies.

- Domain requirements: applications domains should also be considered and satisfied by the proposed solution strategy.

4.2.3 The Exception-Handling Approaches

Klein *el al.* [61] have defined an exception as:

> *"... any deviation from an ideal collaborative process that uses the available resources to achieve the task requirements in an optimal way."*

Static exception-handling constructs such as; catch, try and dispatch mechanism have long been developed and are adopted in many high-level programming languages [61]. However, over the last decade there has been renewed interest in exception-handling to support open systems, workflow management systems and self-adaptive software with modifying code behaviour. Recent work has focused on the development of dynamic dispatch as well as the use of externalised dynamic exception-handling via external exception-handling repositories or knowledge-base systems [61]. For instance, Visser [62] proposed an exception-handling framework, which adopts a meta-level system design to monitor a system' execution information model and trigger an exception-handling process when a model inconsistency is detected. In such a framework, the exception-handling component makes use of a number of models namely [62] ;

- Monitoring model: which represents any mismatch between the system's nominal or required system state with the current system state.
- Classification model: which contains exceptions signature with their associated class or type.
- Verification and recovery model: which is inspection about a certain feature to recover the system again.

Klein *et al.* [63] addressed the management of the exception-handling process, by providing a category of exceptions and a mapping to remedial actions (i.e.how to handle such an exception). This process was called "*exception management meta-process*" anf shown in Figure 4.3: and 4.4 below [64].

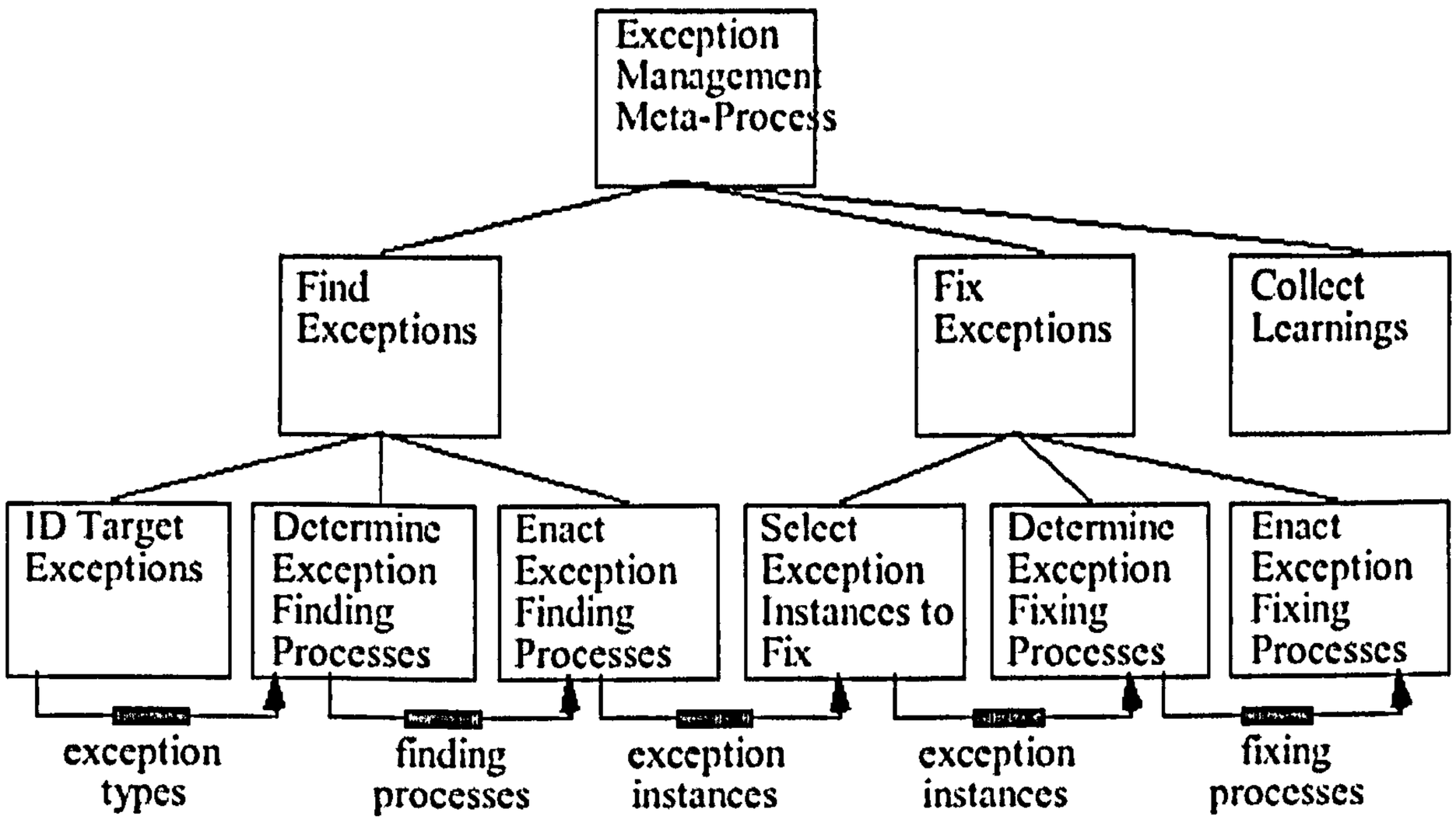


Figure 4.3: The decomposition of the generic exception management meta-process[64].

Figure 4.3 above, outlines the "*exception management meta-process*". This is a high-level exception-handling framework applied to workflow processes, where the workflow process model is checked at design-time using process taxonomy exception information. When an exception is detected then the workflow designer, checks with "*sentinels*" for actual manifestations of the detected exceptions [61]. The notified workflow participant then classifies and handles the exception type using a handbook's knowledge base and associated exception handler processes respectively to uncover the cause of the problem and hence select the appropriate handler. Then the user starts the workflow process modification as proposed by the handler, allowing the process to continue again. A summary of the exception handling process is shown in Figure 4.4 below.

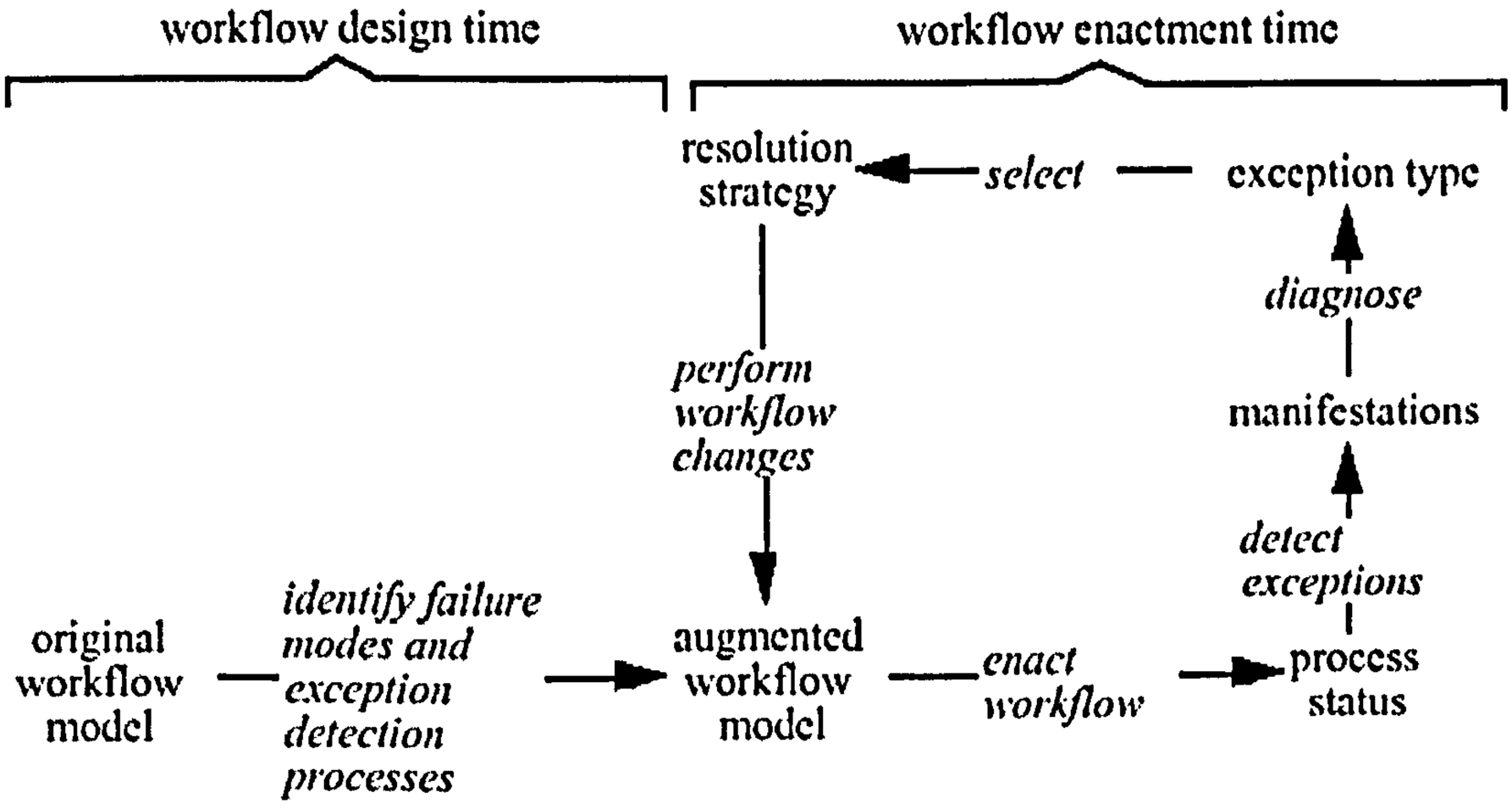


Figure 4.4: Summary of exception management approach [63].

4.3 The Dynamic Management of Distributed System

The autonomic computing vision is based around the notion of systems self-governance and management and requires individual applications and/or infrastructures including middleware, to support runtime and on-demand self-monitoring, self-diagnosing, self-managing and self-adaptation.

Much research in adaptive systems and reflective middleware has generally addressed many aspects pertaining to runtime management, adaptation and reconfiguration [65]. However, many of these did not consider the assurance, predictability and/or the impact of any proposed system adaptation on the system as a whole. This could potentially lead to undesirable systems' states such as runtime errors and inconsistencies.

Over recent years, many researchers have addressed approaches to support the dynamic management of distributed systems, which include policy-based management, event-based management, architecture-based management and autonomic-based management. A review of such approaches is presented in the following sub-sections.

4.3.1 The Deliberated Normative Model

Recently, an extensive literature relating to theoretical and empirical models of software agent and multi-agent systems has been developed[13, 66]. One of the most notable models for a deliberative agent architecture is the Beliefs, Desires and

Intentions (BDI) [67] model, first proposed by Bratman et al. [68], which demonstrate the representations of aims, beliefs and desire about its environment to give suitable responses to be autonomously determined, planned and executed [69]. In addition, the BDI model incorporates a number of essential data structures to determine a plan library and a subgroup of the agent's beliefs, desires and intentions, where each of which can be used as constraints and/or pre-condition for an associated set of intentions. As shown, in Figure 4.5 below, "... *the belief operator B denotes possible plans, the goal operator G denotes relevant plans, and the intention operator I denotes plans the agent is committed to.*" [66].

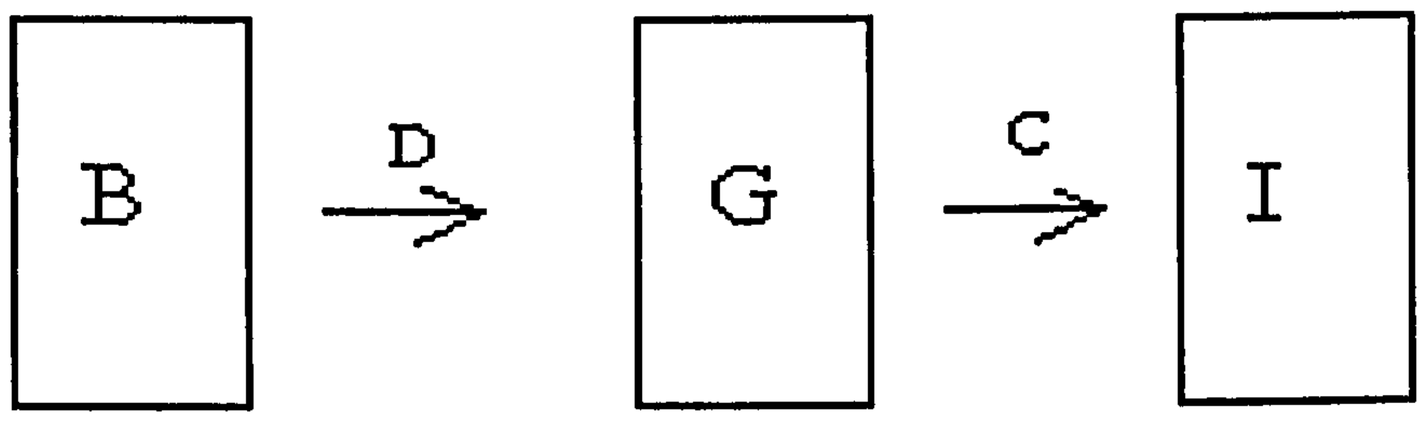


Figure 4.5: Relations between beliefs, goals and intentions [66].

4.3.1.1 BDI Extensions

Many extensions to the BDI model were proposed. For instance, Bratman *et al* [69], proposed an Intelligent Resource-Bounded Machine Architecture (IRMA), which is an extension of the BDI model [13]. IRMA incorporates a plan library addressing a composition of the agent's beliefs, desires and intentions. In addition, the architecture includes a process for reasoning about the environment; a means-ends analysis process to decide which strategy may be used to achieve the agent's intentions. *An analysis process* represents the environment and offers options in response to make any changes. *A filtering process* responsible for determining the subset of means-ends analysis and analysis proposals that are consistent with certain plans; and *a deliberation process* that considers the recommended options that survive the filtering process and produces intentions that are incorporated into the agent's plans.

4.3.1.2 The Epistemic Deontic Axiologic Model

However, the BDI model has many shortcomings, including; the lack of support for revised beliefs, desires and intention sets [70], and the lack of support for situated and normative intentions [71]. In the latter case, the BDI model does not represent any

specific norms, policies or rules for system attitudes or reasoning associated with a set of intentions, which have to be considered to select the correct decision. This is particularly crucial in an autonomic distributed computing environment or open system , which in the worst case, can be characterised by evolving and unpredicted behaviour [71].

Laws et al [14], proposed a model architecture for self-adaptive software, based on both the IRMA [69] and the cybernetic Viable System Model (VSM) [72], where a normative system component has been proposed to compensate for one of the BDI deficiencies, i.e. the "*blind intention*". Many improvements to the BDI model have been proposed, including the Epistemic Deontic Axiologic (EDA) model, which is a normative-based model for multi-agents systems that incorporate agents' social activities including; associated obligations and norms[70] (i.e. Norms " ... *are simply built-in constraints in the agents architecture or rules and protocols the agents necessarily applies* ..." [71]). The EDA agent consists of three main components [70]:

- A cognitive, *epistemic* component to adopt the degrees of belief or disbeliefs, formalising their plans and procedural abilities.
- A behavioural, *deontic* component that predisposes the agent to respond with actions and plans to social obligations.
- An evaluative *axiologic* component that contains the order of the agents' preferences when considering its norms.

The main feature of the EDA model is that the deontic component is represented as generalised goals and plans and is viewed as either social obligations or (self-obligations) individual goals. So, deontic norms determine and evaluate the agent's behaviour based on the epistemic state by considering axiologic norms for resolving arisen conflicts.

4.3.2 Policy-Based Management

For the dynamic management of large distributed systems, much research has addressed the use of policies [73]. Moffett *et al.* [73] stated the necessity of representing and manipulating the policy management of distributed systems, where policy can be defined as "... *the plans of an organization to achieve its objectives* ..."[73]. In policy management, action policies are represented as a policy hierarchy, where each policy in the hierarchy represents a plan that meet a specified objective.

However, the separation of policy management from the policy interpreter (i.e. managers) facilitates both the dynamic change in the distributed system management process and the reuse of managers in different processes [74].

However, there is a need for approaches to program network components and policies specifications. Sloman and Lupu [75] studied authorisation and obligation policies' specification for programmable networks, which, for example, can respectively specify those "*...permitted to access programmable network resources or services*" [75], and "*... event triggered rules which can perform actions on network components ...*" [28]. Such policies are interpreted to facilitate runtime activation, de-activation and/or their modification without having to shut down the network node or system. This can lead to possible conflicts and/or system's inconsistency. For example, "*... an obligation policy may define an activity which is forbidden by a negative authorization policy; there may be two authorization policies which permit and forbid an activity or two policies permitting the same manager to sign checks and approve payments*" [76]. Conceptually, policies conflicts management is an essential concern requiring rigorous policy specifications and conflict resolution mechanisms [77]. Policies could be grouped to refer to the same subject domain and to propagate to the assigned managers [78]. It is reported in [79] that the Ponder language [79] provides a utility for specifying a set of policy types including; authentication policy, obligation policy, policy groups, roles and domain [79].

Other approaches to policy-based management have used condition-action rules to support a static policy configuration-based solution, in which human intervention is required for system reconfiguration and policy deployment. However, Moffett *et al.* [80] have proposed a framework for supporting automated policy deployment and flexible event triggers to permit dynamic policy configuration, focusing on solutions for dynamic adaptation of policy in response to changes within the managed environment. They are concerned with two types of policy adaptation namely;

1. Dynamically changing policy parameters to set new attribute values of the managed object at runtime and,
2. Reconfiguring the policy objects, by selecting, enabling, disabling or load predefined QoS policy that is stored on a simple policy database. An administrator has the flexibility to add or change the database for dynamic adaptation at run time without recompiling or recoding the application again.

Furthermore, other research efforts are focusing on policy specification and enforcement for dynamic service management. For instance, the IETF Policy Working Group [66] is developing a QoS network management framework using the X.500 directory schema[81]. IETF policies are encoded as If-Then conditions and stored in directories. So, the component polices mapping is done by interface roles. However also the IETF policy group has focused on the network layer and not the application layer. Verma and Jennings [82] have proposed a policy-based management system for Service Level Agreements (SLA) management in DiffServ networks by using a tabular specification. Here, a table contains entries for mapping traffic aggregates into classes of service. Hence, the proposed system is not capable of dynamic reconfiguration at runtime. However, policy failures could be possible and should be expected in any programmable environment, so a validation process is still required to support such adaptive management frameworks.

Yoshihara *et al.* [83] have proposed another framework that adapts policy parameters for network monitoring. Here, a management script is expressed using the IETF Policy Working Group proposed representation encompassing; policy specifications, policy management life-cycle, system's notification related to QoS threshold violations and prototyped using the differentiated services. Similarly, Brunner *et al.* [84] have addressed system design for managing QoS in Multi-Protocol Label Switching (MPLS) networks by extending the Common Information Model (CIM) policy model with MPLS specific classes. In this work, Bearden *et al.* [85] have extended the IETF's Policy Core Information Model (PCIM) for supporting goal by using monitoring data to evaluate whether the specified goals are satisfied or not. Such goals are specified as in [80] in terms of obligation policy rules.

4.3.3 Event-Based Management

Large-scale distributed systems many benefit from event-based publish/subscribe interaction protocol as a new scalable communication mechanism, which either publish new events or subscribe to events. There are a number of developed publish/subscribe or event-based systems [86-89], in which events are the basic communication mechanism. It starts by event subscribers, where a client is registering an interest of receiving particular event 's notification, then event publishers responding to the previous subscription by publishing those subscribed events to all event subscribers.

Therefore this model solves the communication problem between publishers and subscribers.

There are a number of ongoing research efforts related to event-based and publish/subscribe management such as Cambridge Event Architecture (CEA) [90, 91] . This provides an event source that publishes events; event sinks that subscribe to particular events. and event mediators who distinguish sources from sinks.

A publish/subscribe system advanced by type-based publish/subscribe is described in [92, 93] .Here an integration of the type model of an object-oriented programming language deals with the event as first-class objects and events are classified according to their type. Subscribers specify their interest by subscribing to the type of object classes they interested in. This approach allows holding arbitrary methods for calling the event object to provide a filtering condition, but on the other hand, it is hard to optimise or distribute since it requires efficient implementation. Siena [94, 95] is a distributed publish/subscribe content-based system, consisting of network event brokers and focuses on a global broadcast operation for advertising and disseminating through the network. However, it does not support event type notions and is static so it cannot deal with failures. JEDI [86, 96, 89] is another Java-based event service framework composed of active objects, since these objects have similar behaviour to event sources and sinks, dispatcher and broker. However its routing algorithm acts as a dynamic event dissemination tree after electing a group leader, who is responsible for performing a global broadcast to all event dispatchers that have previous knowledge of the group leader. JEDI focuses on disconnect and re-connect operation only and does not consider other middleware services like fault-tolerance. Herald [97] is a global, event-based, notification framework for a publish/subscribe service architecture using publishers, subscribers and rendezvous points on the Internet. This gives the framework the scalability to deal with more client than normal event-based notification (i.e. publish/subscribe) and the ability to explore dynamic system reconfiguration. Another two projects, Bayeux [98] and Scribe [99], are topic-based event using rendezvous nodes that are created by routing a message to the identified topic. However, rendezvous nodes can be replicated for fault-tolerance. and all the events must still be sent via the rendezvous node which can become a limitation. Hermes [100] is a distributed event-based middleware that to address most of shortcomings attributes to type-based and attribute-based publish/subscribe model by focusing on event types and

then filtering within the event attributes. Pietzuch *et al.* [101] introduced a framework based on Hermes [100] for general composite event detection(i.e. composite events represent complex models of actions from distributed sources.), to be added to the existing features of the existing middleware architectures. That framework, based on a finite state extended with a model, could decompose the composite event detection and be distributed for system integration. Another feature that could be added to the event by using event calculus is the fluent [102],. A fluent is a situation within the system that is initiated, i.e. triggered by the event with time duration. Friday *el al.* [103] have proposed system-wide flexible adaptation polices using an event calculus for system adaptation and coordination according to client requests [103]. Furthermore, generic models (i.e. not human-generated model) are applied to publish-subscribe systems as a semi-automatic method to get input from the user and reason about the components and the runtime environment [104].

4.3.4 Architecture-Based Management

Recent research in software architecture and description languages[5] has provided a foundation for research funded by DARPA under the DASADA initiative. This is centred around work in software architecture modelling and analysis and reasoning, to support runtime software management including; adaptation and evolution [105]. In this sub-section, the review is focused on research concerns relevant to runtime/dynamic architecture-based management for unplanned architectural model changes of runtime systems. Changing the system runtime architecture could also play a crucial role in the system runtime reconfiguration, since it enables the developer/designer to design the most appropriate changes in an application at runtime, based on the application specific policies and requirements as addressed by [65]. Therefore, architecture-based management requires allowing an external, reusable mechanism to be added to the system infrastructure.

Software architecture could be described as a graph for software architecture formalization [106]. In graph term, nodes represent the individual agents while edges define their interconnection. Individual agents can communicate only along the links specified by the architecture.

---

[5] Since software architecture can be seen from various ways, which defines as components composed structure and rules characterizing the interaction of these components.

The independent dynamic evolution of architecture is defined by a "*coordinator*". For each architectural style, there is a class of architectures specified by a graph grammar. The class formalizes a set of architectures sharing common communication models and rules, the rules of the coordinator are checked to ensure that constraints are preserved by the architectural style. Modelling architectures through categorical diagrams and dynamic reconfiguration could be realized by algebraic graph rewriting [107] and describing architectures and operating changes over a configuration, such as adding, removing or substituting components or interconnections. Moreover, it emphasises obtaining an integrated language that covers the main three-architecture aspects description, constraints and modification.

Garlan *at al.* [108], generalize the previous approaches by making the architectural style and its supporting infrastructure a parameter in the control/repair framework to issue both the interested characteristics (i.e. performance or QoS), and the available operators for runtime adaptation [12, 104]. RAACR [18] extends the capability of adaptive software to the architecture domain; in which the architecture of the adaptive program is modelled toward the architecture of an adaptive controller by adapting the software requirements changes through feedback integration. Oriezy [19] extends the previous adaptive software architecture by addressing the fundamental role of self-adaptive software architecture and the required technology to achieve these fundamentals. These include adaptation management, monitor observation, plan changes and deploy change descriptions to provide a separation of software management concerns. Thereby, achieving separation between computation and coordination concerns as a way of providing a higher level of self-adaptability through system reconfiguration [109]. Similarly, an architecture-based approach to resource management will play a crucial role in the eventual success of the grid. There are three different models of an architecture-based management for grid computing, namely, the hierarchical model, the abstract owner model and the computational economy model [110]. The use of grid computing in mission-critical situations can also benefits from the application of autonomic computing [111].

4.3.5 Autonomic-Based Management

The distributed system community need to design and build computing systems capable of running themselves, adjusting to unpredictable changes and handling resources efficiently [112]. The two main elements of autonomic management are the functional

unit that performs the main operation and is provided by elements such as web services or databases, etc and the management unit responsible for system resources and operational performance and hence the reconfiguration of resources according to adaptive changes [113] . Autonomic systems have been defined by IBM [26] as system that have:

> *"...The ability to manage themselves and dynamically adapts to change in accordance with policies and objectives, which are self-diagnosing and self-healing system, so these systems have the abilities to detect QoS and performance [1], and allow users to accomplish what they request rather than try to handle and repair these computing systems..."* .

Intel research explores new computing system that provides the issue like the autonomic computing called proactive computing, but may be the difference is that autonomic computing focus on managing the computing system complexity but the proactive computing adds the need to monitor and build complex real-world interactions [114]. Since autonomic computing systems have the ability to monitor, diagnose and healing themselves, this requires that the system has the ability to dynamically insert and remove code in real-time systems, a technique known as "hot swap". Hot swapping is proposed as a means to [115] enable the autonomic software systems. This is performed by either *interposition* or *replacement* of the code. Interposition involves inserting a new component between two existing ones. For example, inserting one monitoring component when a failure is detected at run time that reduces the performance of the running system. However while replacement allows an old component "swapped" with a different implementation while the system is running. The new component then continues the management of resources. The interposition and replacement methods of the hot swapping technique effectively support autonomic computing as it becomes possible to monitor, diagnose and manage the computing system. This is especially of an object-oriented system where each resource is a different instance of an object [115]. Nevertheless, successful autonomic systems not only need to self-detect, self-diagnose, self-heal, but also to self-protect to allow autonomic management in a secure environment [113].

4.4 Summary

In this chapter, we describe related research in terms of two research categories; the static management of distributed systems, and the dynamic management of distributed systems. We discussed related research and how this attempts to achieve control and management of the system. The approaches are now initially considered. .

Considerable research has been undertaken in the area of dynamic control and adaptation in the running system. However, those systems have focused on user-mediated management, whereas our research is mainly concerned with automated/self management and adaptation/reconfiguration [15]. There has been some closely related work on self-adaptive software [19] and reflective middleware [116] but these give less emphasis to the coordination and control of systems adaptation to respond to either failure and/or other inconsistencies. Our research extends this by showing how to turn "control at design time" into "control at runtime" [15]. Some other efforts in this area have investigated formal foundations for this in terms of protocols, but have not carried the results through to implementation.

To accomplish this, we must propose some new mechanisms to allow external, reusable, namely, a autonomy, deliberative, life-time middleware control service that can be added to the management of distributed systems in a flexible manner. Our contribution extends the ideas described previously in these developments by integrating the autonomic control service within a middleware service. To address these issues, we must determine the requirements and design of the architecture of an autonomic middleware control service. These are which are fully explained in Chapter 5 and Chapter 6 respectively. The architecture and design are then tested by implementation and evaluated by case studies, the details of which are found in Chapters 7-9 respectively.

Chapter 5

Requirements

5.1 Introduction

From a service-oriented architecture viewpoint, a distributed software application can be seen as a federation of distributed software services, interacting over the network through well-defined interaction and control models. Over the past two decades, much research has focussed on a range of concerns related to adaptive systems design, development and management, including; reflective middleware, conflict detection and resolution and system coordination. However, as described in Chapter 4, much work is still required to study and develop a middleware governance model and associated control services to support the development, deployment and lifetime management of self-adaptive applications.

In this chapter, we present the essential and crucial requirements only that are required for evolving the meta-control software model, which are responsible for developing autonomic lifetime management of application services. The requirements addressed here are categorised along different autonomic concerns, including the following:

1. Conflict self-detection, where the monitoring model in our control service is responsible for conflict detection in either a decentralized fashion as in a *service*-monitoring model, or centralized as in a *system*-monitoring model.
2. Conflict self-diagnosis, which includes; (a) conflict identification and, (b) a conflict classification that provides the basis for the solution and repair strategies that trigger the appropriate action for such conflicts.
3. Conflict resolution and self-repair strategies, which address dynamic and generic selection, execution and evaluation phases.
4. System dynamic adaptation, reconfiguration and coordination among the services in the distributed computing environment.

A detailed description of the requirements is detailed in the following sections.

5.2 The Autonomic Middleware Control Service's Requirements

The requirements for managing distributed computing start by providing the system the with ability to detect, classify, fix and reconfigure itself at runtime without any disturbance and/or system shutdown. Runtime conflicts and systems errors are typically difficult to predict, detect and rectify at runtime without introducing a new approach that dynamically rectifies the inconsistencies or failures.

Inconsistency can be denoted as "... *any situation in which two descriptions do not obey some relationship that is prescribed to hold between them*" [41], in other words the system consistency rules are broken or unsatisfied. Consistency or control rules we expressed to describe the relationship against which checking can occur (i.e. the generation of control rules is not our concern in this research). Examples of such inconsistencies/failures are runtime faults and stakeholder conflicts. Figure 5.1, shows an example of consistency rules.

Example Consistency Rules	
Rule 1	In a data flow diagram, if a process is decomposed in a separate diagram, then the input flows into the parent process must be the same as the input flows into child data flow diagram.
Rule 2	For a particular Library System, the concept of operations document states that "User" and "Borrower" are synonyms. Hence, the list of user actions described in the help manuals must correspond to the list of borrower actions in the requirements specification.
Rule 3	Coding should not begin until the Systems Requirement Specification has been signed off by the Project Review Board. Hence, the program code repository should be empty until the SRS has the status 'approved by PRB'.

Figure 5.1: Informal examples of consistency rules [41].

Additionally, system monitoring is used to detect any conflict and so helps to focus attention on problem areas. This and may be used as tool for learning or as Validation and Verification (V&V) tools for analysis and diagnosis and therefore facilitate the generation of appropriate solutions.

Figure 5.2 below, illustrates the general building blocks, processes and sequence required during a conflict/failure resolution. Here the control service analyses information provided by the monitoring model, either generated from checking against the control rules or from a feedback process and notifies the diagnosis model. Then the diagnosis model notifies the repair model with the conflict type to facilitate the

selection, execution and evaluation of a repair strategy, as the repair strategies invokes the required reconfiguration operators. In turn, the reconfiguration model establishes the required changes to adapt and coordinate the system in consideration of the new solution (i.e. changes), which may cause further conflicts that activate a feedback process [41] to start the control sequence again and establish the appropriate decision.

Essentially, the autonomic middleware control service allows a distributed system to act as a self-detecting, self-diagnosing, self-repairing, and self-reconfiguration system to facilitate the prescribed operation of autonomic behaviour.

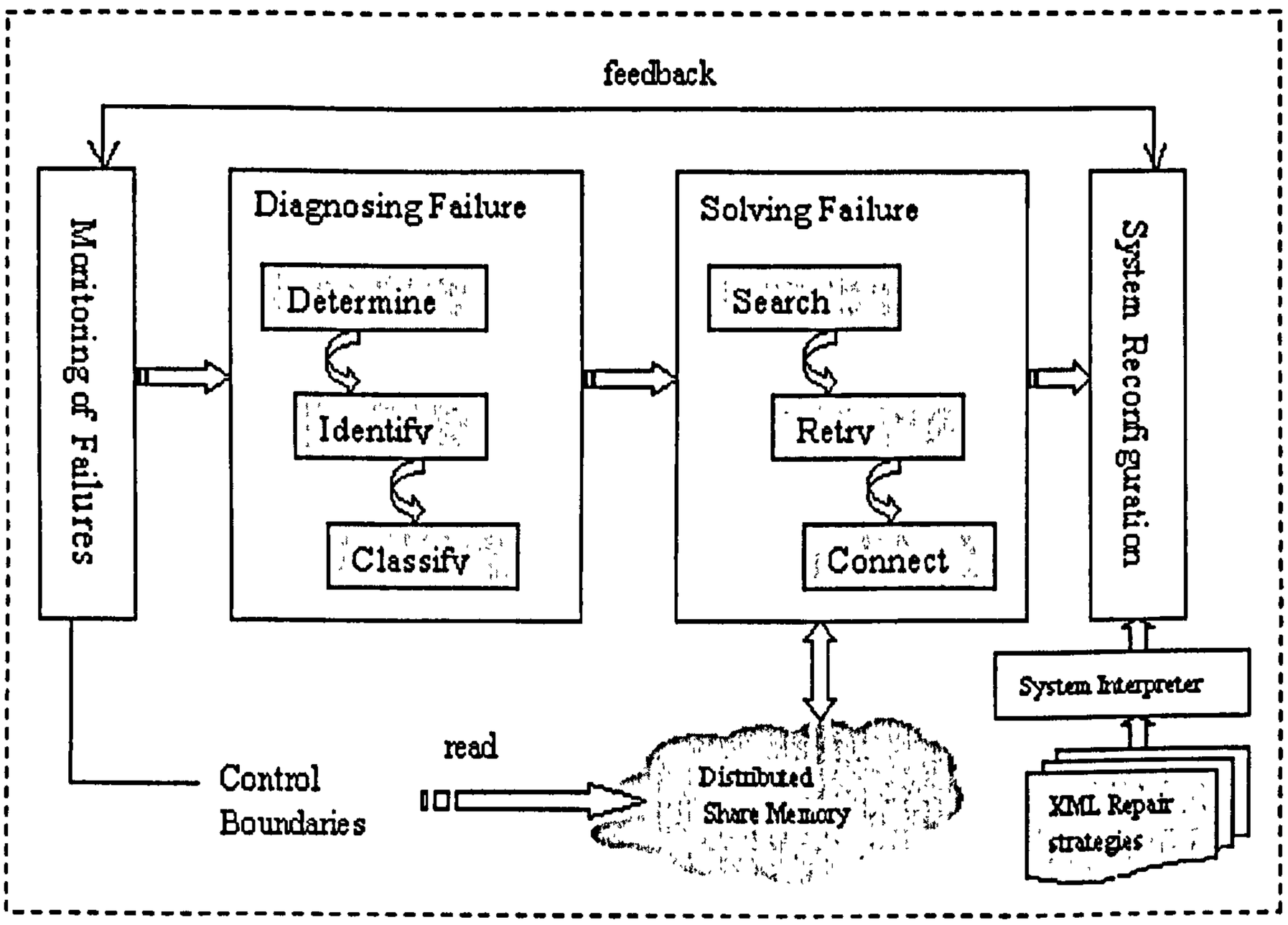


Figure 5.2: The autonomic management process's requirements

The details of each model shown in Figure 5.2 are covered in the follwing sections.

5.2.1 Conflict Detection

Our conflict resolution management process starts with the detection of conflicts or inconsistencies, or any abnormal behaviour of the monitored system. This is achieved by utilizing a set of heuristic *rules* to self-detect and self-identify the conflicts that arise at run-time, giving the rules a crucial mission in the management process. Thus, there is a need for a precise definition and specification of such rules prior to starting the consistency monitoring system.

Rules could be derived from a number of sources such as; useful information relating to performance, Quality of Service (QoS), behaviour of application and domain-specific constraints. Our rules require a definition of scope, so action/behaviour can be checked against those rules [41]. In a distributed system, the services and its rules are distributed over the network among several servers, so the use of a service-oriented middleware invocation methods is recommended for retrieving the *rules* and a querying service to evaluate their executed action. The sources of consistency rules are many, including [41]:

- The notation's definition that exist in the development process. For example, each variable in a programming language should be consistent with its declaration.
- The method's development that consisting of a set of notations with some guidelines. Such as a method for designing distributed systems to communicate, two subsystems are defined consistently within each interface of those subsystems.
- The model of the development procedure that defines evolutionary steps, the entry and exit conditions for those steps and the product's constraints of each step.
- The possible occurrence of a consistency relationship between instances of two objects, even if this has not been determined before on either notation or method model.
- The domain of the problem and the specific domain for the constraints that are not related to either the notation or method model. Such as discovering, specifying and refining rules in the development procedure.

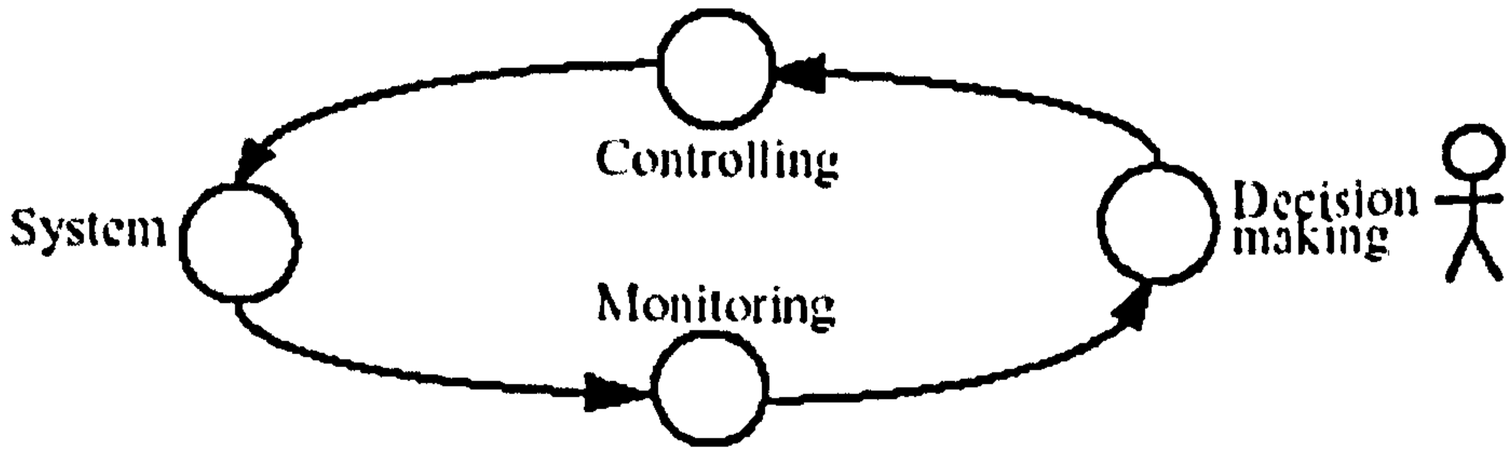


Figure 5.3: The relationship between management and monitoring [41].

In this section, the monitoring process is considered to gather information for post-mission performance characterisation. If the observed information is defined within

specified bounds, this should facilities the process of conflict detection is facilitated by determining the specific part that should be monitored, in order to start the management process. The control decision depends on such information gathered from the monitoring process as shown below in (Fig. 5.4), i.e. the relationship between the monitoring process and the management process.

Whenever inconsistencies are detected, a diagnosis process is initiated to locate and identify the conflict type. As the pre-determination of a full set of rules covering all possible actions on the large project and at runtime is not possible, it is necessary to develop a dynamic, external document format such as XML documents for specifying rules that are proper for each individual behaviour, where they can be matched correctly.

START_ENTITY Monitor_task

// Monitoring of the services operation by checking if the services attributes is within its bounds and notify the associated system controller. //

- Monitor_rule: list of detect_condition;
- System_notify: notification;
- System_control: control_action

END;

START_ENTITY Detect_condition

// A condition applied for measuring service parameters and attributes at runtime. //

- Service_parameters: parameters_initialization;
- Measurment_interval: [nominal_values-non_nominal_values];
- Duration: execution/runtime;

START_ENTITY Notification

// An immediate notification in the case of the occurrence of a problem detected. //

END;

START_ENTITY Boundary

- Acceptable Range of each service parameters.
- Operator: for example<=, >, equals () or! =

END;

Figure 5.4: Task monitor script's description.

Conflict detection using management rules can be performed dynamically for a set of policies, boundaries or rules at runtime. A runtime detection mechanism acts as filter preventing the activities that must not be performed or are not permitted [73], on the other hard it is required to detect actual conflicts rather than potential conflicts, may be performed statically (compile-time) [77].

The definition of bounds for the measured feature will create two 1-dimensional feature spaces, one defining the *nominal* feedback, the other the *non-nominal* feedback. After the detection of an exception, the monitor system has to know how to notify the rest of the system about this situation (`notification` description). Figure 5.4 above illustrates the *`monitor_task`* actions to provide the system with the monitoring utility.

5.2.2 Conflict Identification

The identification process starts whenever an inconsistency is detected. The diagnosis process includes the following [41]:

- Locating the failure, by identifying an inconsistency event and notifying the controller by sending a triple containing the event description, source system and ID of the violated consistency rule (e, s, rid).
- Determine the cause of failure by analysing the triple (e, s, rid); the failure event has broken a rule like, missing information, time out or security failure, etc. The order or history of such events aids in identifing to identify the root cause of the failure. For example, failure detection instrumentation may follow a timing failure event, which indicates that the server has recovered and the root of that failure is a timing failure. Alternatively, a timing failure may occur first, followed by several subsequent omission failures, which represents an indefinite timing failure. So, in this case we deduce that the root failure is an omission failure. The ordering of these messages is significant and it represents a failure pattern that can be used to classify the type of the failure.
- Classification of the type of failure; by providing the basis for the selection of a conflict resolution strategy. For example, the failure from the type of rule broken, or type of action that causes the failure.

5.2.3 Failure Classification

In this section, we define a classification of system's failures that provides an understanding of the effect of the failure. and aids in the selection of appropriate resolution strategies. As shown in Figure 5.5 below, the main types of failures as classified by Hadzilacos and Toueg [117] are:

1. **Omission failures:** These refer to cases where, for instance, a component or a communication channel fails to perform actions that it is supposed to do. Typically, a component omission failure occurs when a component crashes or fails to respond (fail-stop) and a channel omission failure may occur when a message is sent, but never received. In a distributed application, omission failures occur when either a client or server component crashes or fails to respond because it is busy or has stopped. For instance, to monitor a server's availability, a heartbeat instrument (monitor) is attached, which send a heartbeat pulse – message at regular intervals. If the heartbeat monitor failed to receive some of the pulses (messages), the failure detection monitor assumes that the client-server communication channel has suffered an omission failure and notifies the resolution strategy unit to select a suitable solution for that omission failure.

2. **Timing failures:** may occur in synchronous communications when components fail to execute steps within specified time limits or when messages arrive too early or too late because of problems in components or their communication channels. The timing failure may be regarded as a special case of an omission failure, namely, one for which a message is delayed indefinitely. Therefore, the treatment of timing failures is based on similar ideas to those used for omission failures. So the approach used for timing failures may be regarded as a super class to that used for omission failures, which encapsulates the omission failure semantics. Timing failures normally occur in synchronous communications and are particularly relevant to multimedia applications that may involve audio or video streaming. They may also occur due to clock drifts in either the client or server or congestion in communication channels. The detection of timing failures does place some essential requirements on client and server components, in that they are required to be equipped with capabilities to acknowledge that a message has not been sent or received within some time-interval. To this end, we assume that either a client or server will throw some form of timeout exception when the time-interval is exceeded. As mentioned earlier, timing failure detection is an extension of omission failure detection; its entry point is that of a "supposed" omission failure. When an omission type failure occurs, a failure detection checks for client or server exceptions. If the

client/server is still alive, it means it is late in responding, so the failure detection assumes either the server or the client-server communication channel has suffered a timing failure.

3. **Byzantine failures:** represent the most complex failure semantics in which any type of error may occur. Typically, a Byzantine component failure is one in which the component arbitrarily omits to send a message(s) or sends an unintended message(s). Along similar lines, Byzantine channel failures may occur when messages become corrupted, or arbitrary messages are sent or arbitrary results received. Such failures are rare, as software techniques may be used to detect them, however, they cannot be discounted. The Byzantine failures that can be detected are those due to messages sent or results received out, of sequence, unexpected or repeated messages and corrupted messages. The detection of Byzantine failures is based on the introduction of a new detection message (dm_n), which records a summary of application messages sent and results received since a previous detection message (dm_{n-1}). The failure detection appends all application-level messages with the timestamp of the previous detection message dm_{n-1}. The detection message dm_n has the same fields as an application-level message except for the data contents field, which stores the timestamps of the messages sent since the previous detection message dm_{n-1}. The failure detection then checks the contents of dm_n and dm_{n-1}, before the application-level message is used in a method invocation and again after a result is returned. Comparisons between these checks may then be used to find any breaks in the time sequences between dm_n and dm_{n-1}, for messages sent by clients and results returned by servers. If any of the above checks are positive, the failure detection monitor recognizes that either the client or server or client-server communication channel has suffered a Byzantine failure that may be identified by the nature in which the check proved positive.

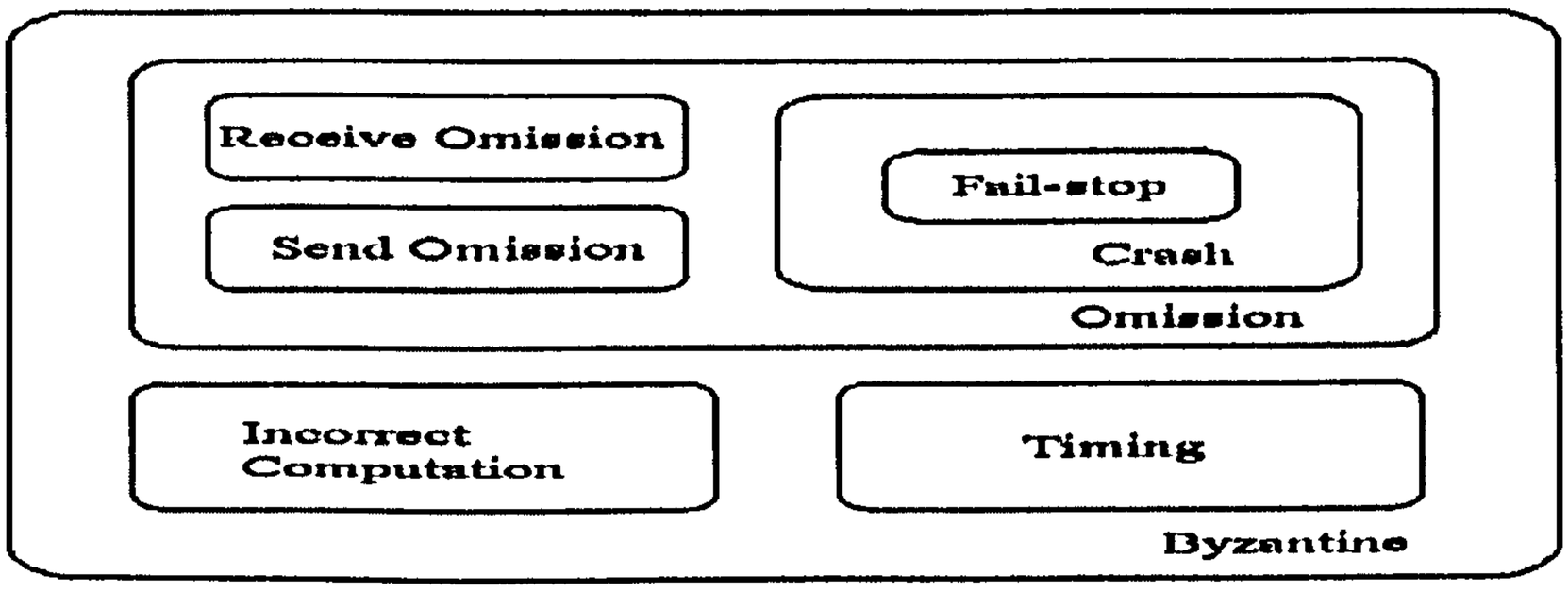


Figure 5.5: Failure's classification in terms of failure types and their scope.

5.2.4 Conflict Resolution Strategies

The resolution strategy area distinction is made between identification and classification of the failure and the process of updating and verifying the environment model. Also the strategies set for the system recovery stage is a mix of heuristics, planning functions and activities, where a repair strategy can help to fix the environment by selecting and triggering selected recovery actions (see Fig. 5.6 below).

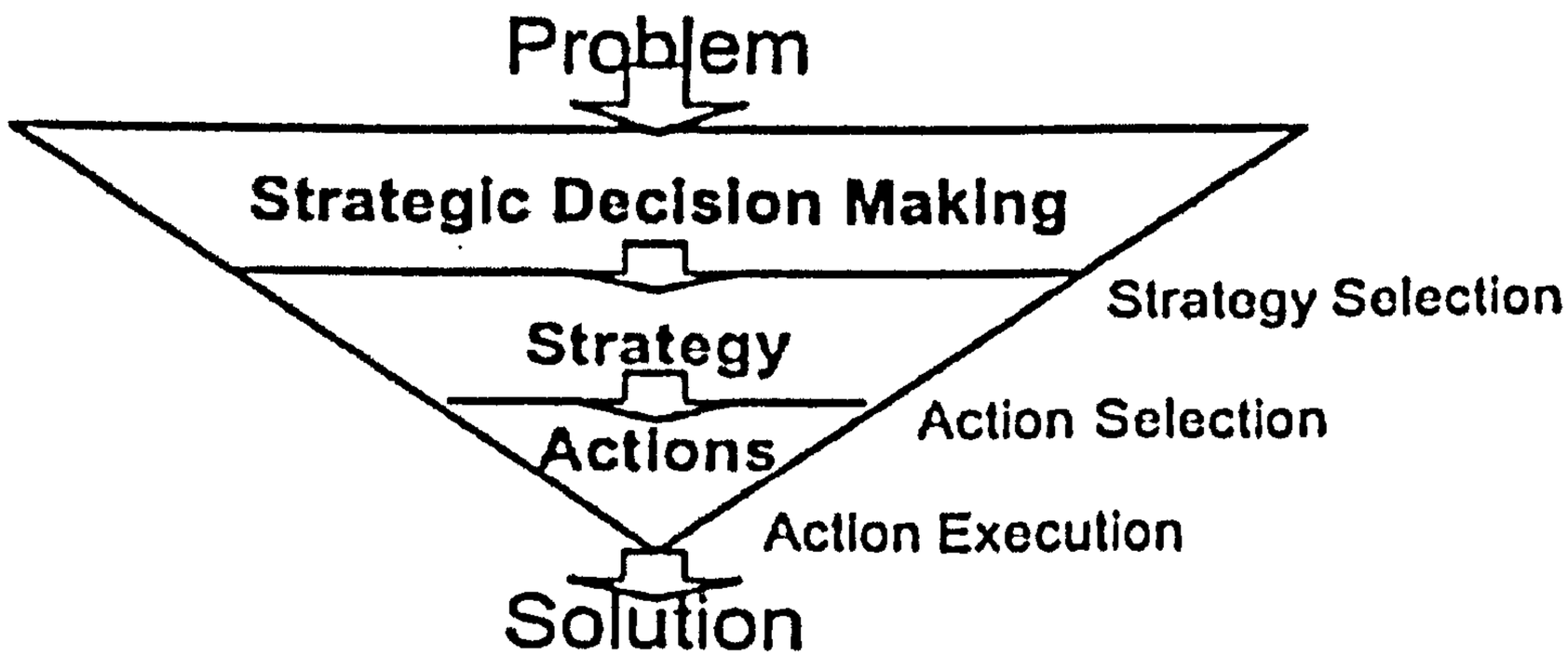


Figure 5.6: Relation between Actions, Strategies, and Strategic Decision Making [118].

The rigorous selection of recovery strategies is vital for achieving a reasonable success rate and avoiding further system's inconsistencies. Such a selection process takes into account a number of considerations including;

- The classification of conflicts such as conflicts that may occur during the plan generation phase, or in any of the other phases as indicated above.

- The maintenance of the application, application services and their dependencies, including; software components' dependency, execution and interaction.
- The maintenance of a service's preferences including; QoS and service level guarantees.

Following the script-based knowledge representation approach, repair strategies are defined and structured as a sequence of plans that are valid for a specified scenario with a defined scope (context) of applicability and/or influence. In other words, the strategies are specified as a situated and bounded plan set – when and where a plan is applicable. For instance, the evaluation of the plan's bounds will usually involve the examination of various properties of the service to detect, identify any plan execution conflicts and determine the capability of it being applied. If the applicability exists, the plan executes a repair plan or the task that is written as an encoded program using specific, designed operators.

Evidently, the precision and robustness of conflicts identification has a vital effect on the selection of an appropriate resolution strategy and the outcome of a considered failure recovery process. In addition, when selecting a repair strategy there is often more than one matching strategy, thus requiring the inclusion of a utility-based rule selection mechanism [52] (see Fig. 5.7 below).

START_ENTITY Control_action

- Operator: for example: add ( ), remove ( ), or connect ( );
- Exception: represents the information necessary to classify the abnormal situation or the unexpected events with the service.

END;

Figure 5.7: Control action operators.

5.2.5 Control Rules

A control rule is generated based on the comparison of two main parts. These are the expected beliefs (i.e. current states) and the desired action. For example, if a client cannot discover and invoke its desired service, the control rule will lead to requesting another service lookup process and notification of the client with the alternative service. There are two types of control rules required for establishing any control service or process, namely:

1. External control rules: which are not attached to any particular strategy, but are used for continuous monitoring purposes such as; repair task/execution progress monitoring and coordination. In the event of a conflict during the execution of control process itself, then a conflict is detected and by using the feedback process, the system will start another control thread (process). For example if the average latency of the control process is taken over the maximum allowed latency, then the control rule detects and fires an execution failure.
2. Internal control rules: which are embedded within a resolution strategy to monitor the recursion of a strategy's actions or attributes (i.e. partials solution). For example, check that the number of connected users does not exceed the maximum number permitted by the service provider.

5.2.6 System Reconfiguration

This section describes the requirements of a reasoning mechanism for selecting repair strategies and the control rules to coordinate and validate an automated application reconfiguration. Consequently, a coordination component is required to manage the various forms of task coordination that naturally occur whenever the services have inter-linked objectives or share a common environment and/or when they negotiate their intents and/or constraints to achieve their common goals.

A negotiation utility is required to facilitate the control and coordination of distributed computational aspects of distributed applications to enable distributed actors and services to cooperate through their services brokerage as in the Contract-Net model[47] (Sec.4.2.1.1). Therefore, there is a requirement in our study to incorporate a coordination utility that should be;

- Shared and accessible by all services or parties.
- A centralised service to send/receive or request/respond to satisfy the coordination aspect.
- Able to provide a reliable distributed database or storage for storing their coordination requests, strategies or history.
- Able to communicate between itself and system services (e.g. communication by using notification aspect).
- Persistent and exist at all times.

One of the main requirements for reconfiguration is to establish coordination as a simple unified mechanism providing dynamic communication, management, and the sharing of objects between network resources like clients and servers. In a distributed application, distributed shared memory such as; Linda space, JavaSpace or T-space can be used to act as a virtual space between providers and requestors of network resources or objects.

START_ENTITY Reconfiguration_Strategy

//The strategy guidelines on how conflict has to be solved, considering a possibility of different coordinated services issue//.

- Establish required communication between services
- Apply: conflict_repair_operator (Fig. 5.10).
- Exclude: failure_source (OPTIONAL).

END;

Figure 5.8: The reconfiguration strategy sequence.

Each reconfiguration strategy (see Fig. 5.8 above) is assumed to be effective for some set of conflicts (i.e. the successful execution of a strategy may give facts and ideas about the reason for the conflict, which could be declared by a set of *excluded* failure sources in the future).

A reconfiguration strategy must indicate reasonable actions to take in the event of a specific conflict situation. This means that the strategy has to indicate new choices for the planning functions that are able to generate a new or modified sequence of actions. The initial sequence of system desires should be stored in a distributed shared space, accessible by all that are considered to be in a considered distributed application services' federation and the middleware services associated to the application lifetime management including; instrumentation, service monitors and controllers.

In the case that more than one resolution strategy is suitable for a given intercepted conflict, a unification algorithm with backtracking is required to support strategy/plans selection for a given situation guided by a utility and/or uncertainty measure[6]. For example, in the case where a number of alternative strategies can be applied, the reconfigurator selects one based on the following sequences (see Fig. 5.9 below):

[6] A rescheduling of strategies in this situation may be the right strategy or a heuristics rule base selection such as FIFO.

- Retry: sometimes if a solution fails to solve the failure it is initially better to attempt several connection retries, followed by consecutive pauses. If the retries are unsuccessful, the strategy then searches for another choice for that failure.
- Search: for an alternative that considered as alternative solution of choices if all of the permitted trials fail. This way determine an alternative resolution, which should consider an architectural reconfiguration.
- Connect: the client connects to the desirable action that achieves its request, otherwise connect to the alternative without disturbing the whole system or throwing an exception behaviour.
- Handle exception: that is activated as a consequence of any exceptional behaviour thrown when a control rule executes because of a conflict that cannot be resolved. Exceptions are dealt with according to priority, which may be low, intermediate or high to accommodate varying degrees of fault tolerance. For example, the invocation of the *connect()* method may result in a `RemoteConnectionException` due to the unavailability of a communication service provider.

START_ENTITY Conflict_repair_operator

- A scenario of the configuration that has to be made in the original plan, to reach to the satisfied status by apply one of the following functions for the proper solution:
- connect: connection( );
- retry: retry( );
- search: search( );
- handle: throw_exception()

END;

Figure 5.9: Conflict repair operator.

5.2.7 System Interpreter

To facilitate flexibility and runtime extensibility of the control rules and repair strategies, an externalised rule-based system is required to enable a separation of concerns, including; middleware services logic from middleware meta-control and application control heuristics from application services' management rules. Therefore, there is a need to translate the dynamic strategies, described using a dynamic format such as a markup description language. The system-associated interpreter is one of the

main requirements for developing such an approach as a lightweight interpreter. This is used to parse and map between the markup description language (e.g. XML) format and executable code to facilitate the autonomic, dynamic and executable concepts required in the proposed approach.

5.3 Summary

In the past two decades, distributed software system management can be seen as a group of dependent services distributed, communicating and coordinated over the network. This requires full-time management particularly runtime management.

In this chapter, we have addressed the issues related to an autonomic middleware control service. These are (a) conflict detection, using a set of control rules against which behaviour is monitored to detect conflicts, (b) conflict identification and classification that is activated whenever a conflict is detected as a result of the execution of the control rules which locate, identify, and classify the failure, (c) failure/conflicts resolution strategies, which selects the proper strategy for the identified failure, (d) system reconfiguration provides the capability for our approach to reason about the current state and re/configuration of an application, in order to keep the validity of an coordination/configuration strategy, (e) a system interpreter, responsible for translating the markup system strategy format an the executable format and recoding the control operations at run-time, and (f) the control rules which examine the control services process either internally or externally.

The control service uses its control rules and the information gathered by the service monitor together with services coordination relationships, to detect, identify, classify, interpret and repair/correct any exceptional behaviour or conflict. Then, maintaining system coordination is achieved using a system reconfiguration process for which the application's services coordination and interaction must remain satisfied

Chapter 6

Autonomic Middleware Control Design

6.1 Introduction

An essential feature of any middleware services is its capabilities to dynamically control and adapt in response to runtime changes. The design of such dynamic control capabilities are often derived from the software architecture model that describes the software components and their interactions, the properties and policies that regulate the composition of the components and limit the allowable range of control operations.

Much research undertaken by the self-adaptive software community hinged around adapting mechanisms from control engineering and intelligent systems. For instance, Osterweil and Clarke [20] argued for the need of a continuous self-evaluation mechanism to facilitate the delegation of software's continuous testing and evaluation from humans to the software itself. In support of their claim, they described a proof of concept controller architecture that employs feedforward and feedback loops to monitor and regulate a controlled system's operation in accordance with its specified control model.

Other research in reflective middleware [116] and policy-based distributed systems management [74] has focused on the use of managerial or meta-level protocols to attain reactive adaptive behaviour. However, reflective and policy-based management approaches alone cannot address all of the needs of self-adaptive software because of their inability to maintain a faithful autonomic and self-adaptive runtime model of the system.

This chapter considers the design of the proposed autonomic middleware control service for distributed self-adaptive software that combines three main services namely; the Service Manager, the JavaSpace Service and the System Controller. In particular, this chapter describes the three-layered design model adopted, namely (Fig.

6.1); (i) the middleware core services (Sec. 6.3), (ii) the autonomic middleware control service (Sec. 6.4), (iii) and user applications services (Sec. 6.5). This architecture is based on a control service model that follows a cycle of monitoring the assigned target application, detecting undesirable behaviours, identifying conflicts/errors, prescribing remedial action plans and enact change plans through reconfigurtion.

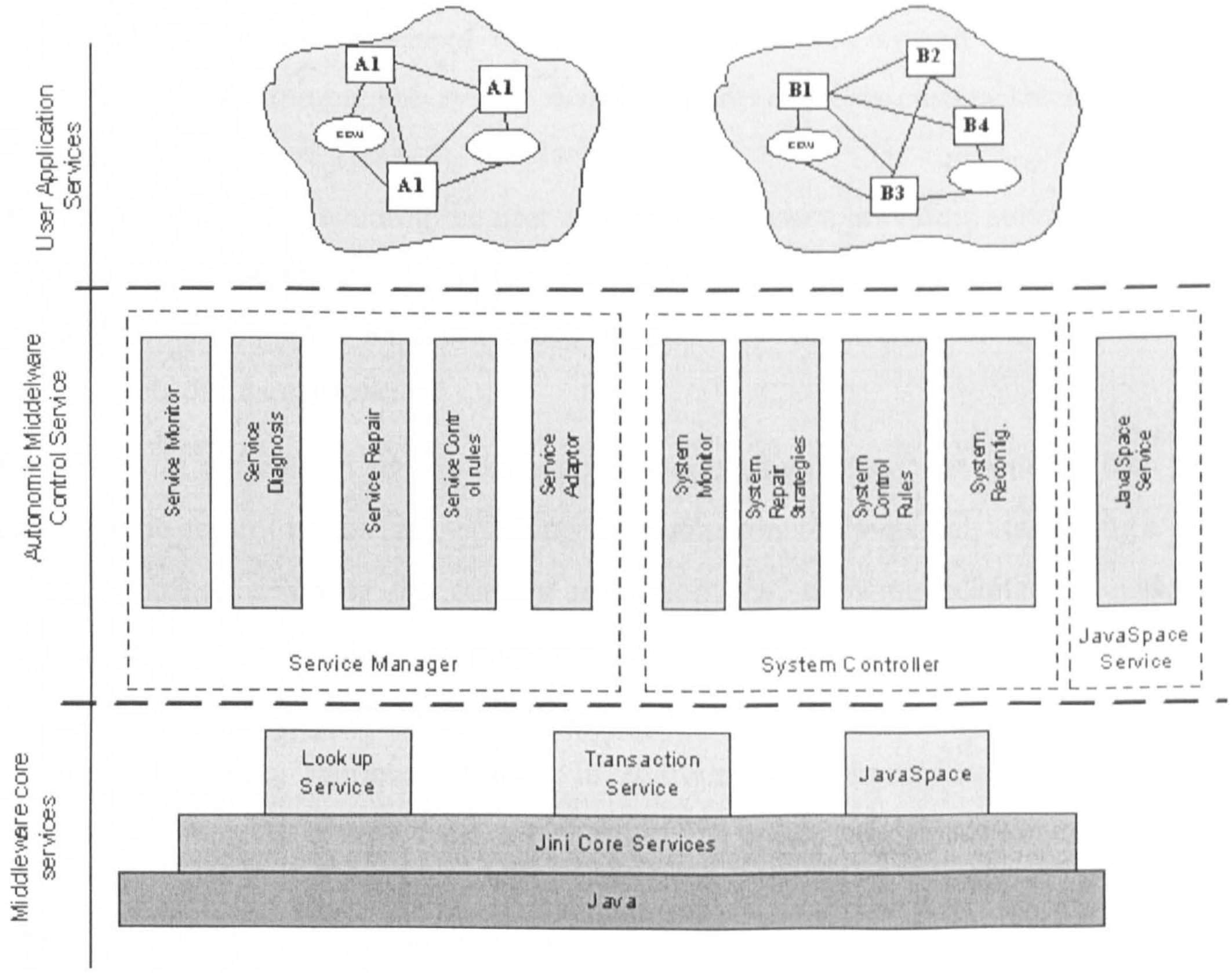


Figure 6.1: The middleware control service architectural layers.

6.2 The Control Service Architecture

This study considers the development of a control service that monitors itself, and maintains a faithful autonomic model of the runtime system. As shown in Figure 6.1 above, the control service layer encompasses:

- The first layer: containing all middleware core services such as; a registration service, discovery service and space service (Sec. 6.3). The middleware control service can establish the required communication via its registration service, as the application services register with a middleware service locator (e.g. in Jini a

service locator is *ServiceRegistrar* and the registration method is *register ()*). Therefore, clients can easily discover any registered service or share services through the JavaSpace service (see Sec. 6.5).

- The second layer: containing the proposed autonomic middleware control services encompassing meta-services such as; the service manager, system controller and the distributed shared space. The latter plays several roles namely; a distributed shared memory and a persistent system environment service for examining the system *beliefs* (i.e. the services current states) and system *desires* (see Sec. 6.4).
- The third layer: containing the user application services providing services that could be requested by clients (see Sec. 6.5), and managed by the autonomic middleware control services, which repairs any occurrence of application conflicts or inconsistencies.

The workflow of our control service begins by a monitoring or feedback process, then determining the control inputs [20], checking the parameters of the model, classifying a discovered conflict, selecting the conflict resolution law, allowing adaptations and system reconfiguration. Thus, the proposed control services architecture achieves autonomic control by a continuous cycle of detection, identification, categorisation and resolution of emerging runtime conflicts. In addition, a feedback loop is used to continuously monitor the control service itself providing middleware fault-tolerance.

6.3 Middleware Core Services Layer

The middleware core services support our approach with the basic level of services. These contain three basic services that enable the registration service (see Sec 6.3.1), service discovery service (see Sec 6.3.2) and distributed shared memory (see Sec 6.3.3). These are described in some detail below.

6.3.1 Registration Service

The registration acts "... *as a proxy object to control the state maintained about the exported service object stored on the lookup service* ..." [38]. This service is responsible for registering the application service's object, which is the visible part of the service and will be downloaded to clients over the network (e.g. Jini lookup's registration) by "listening" on a port for registration requests. When a request is received, a dialogue between the application services is established and a copy of the

service proxy is moved and stored over the specified network's port. For example, in Jini middleware the service's object could be registered with the Jini lookup service (see Fig. 6.2 below), whenever the lookup service receives a request on a port, it sends a lookup service object (i.e. *registrar*) back to the server. This acts as a proxy to a lookup service and executes on the service's Java Virtual Machine (JVM). Any service's request submitted through such a registration uses any suitable protocol to establish this request (i.e. HTTP protocol).

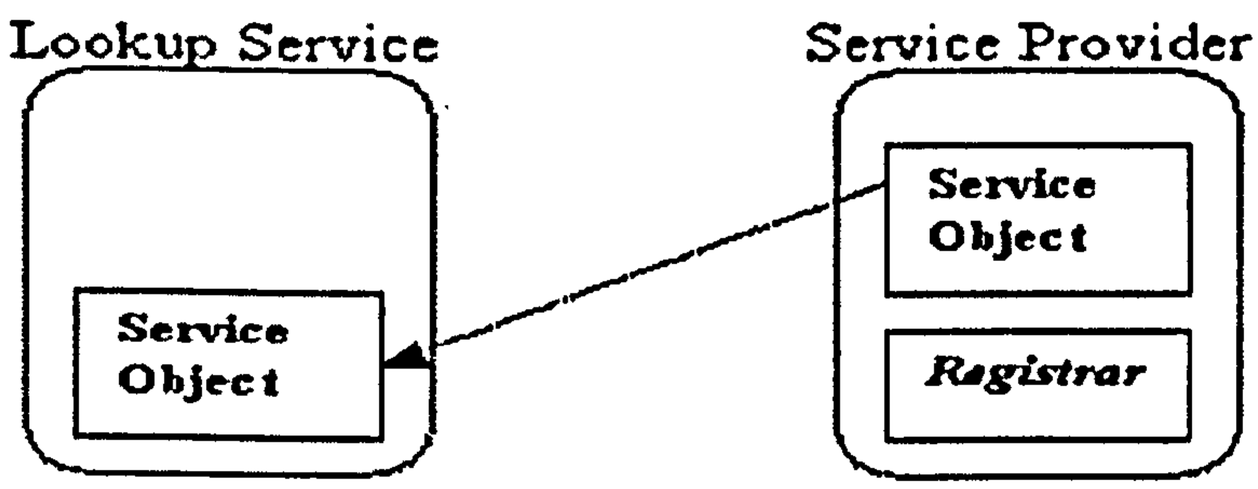


Figure 6.2: Service object lookup registration [38].

6.3.2 Discovery Service

When a service has been published, clients can discover it by querying the lookup service. The clients establish a request using a template, which is compared with the service proxies that are currently stored over the network. If a matching process is found, then a copy of the matched service proxy is moved from the network to the client machine.

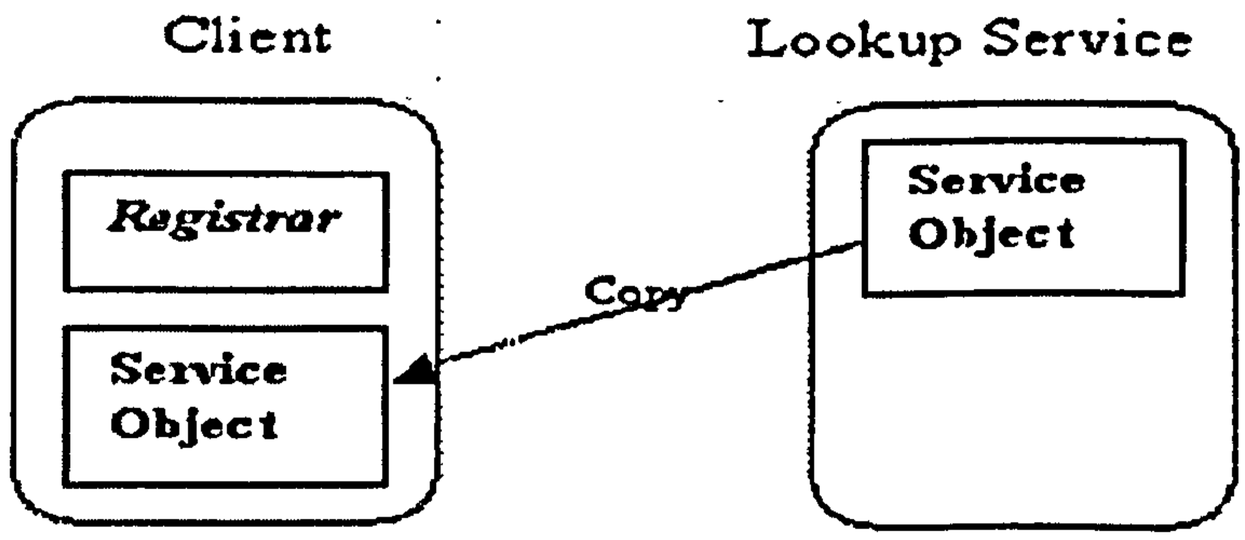


Figure 6.3: Client's discovery of the registered service [38].

For example, a client is trying to get a copy of the service object to its own JVM, so it discovers the *registrar* from the lookup service and requests a service object copy to be delivered to the client machine (see Fig. 6.3 above).

6.3.3 Distributed Shared Memory Service

This is based on a persistent object model used to store, exchange and coordinate the activities of interacting distributed processes. Here processes communicate indirectly by exchanging objects via the shared spaces. In addition, shared spaces have many important properties [49] including (see Fig. 6.4 below):

- Spaces are sharable, where remote processes can interact concurrently.
- Spaces are persistent, where an object stays permanently is and stored in the space untill it is removed.
- Spaces are associative; the object can be accessed with no known name-using *template.*
- Spaces are based on a transaction service to ensure that operations on spaces are atomic and single.
- Spaces take a copy of objects and change their fields and attributes.

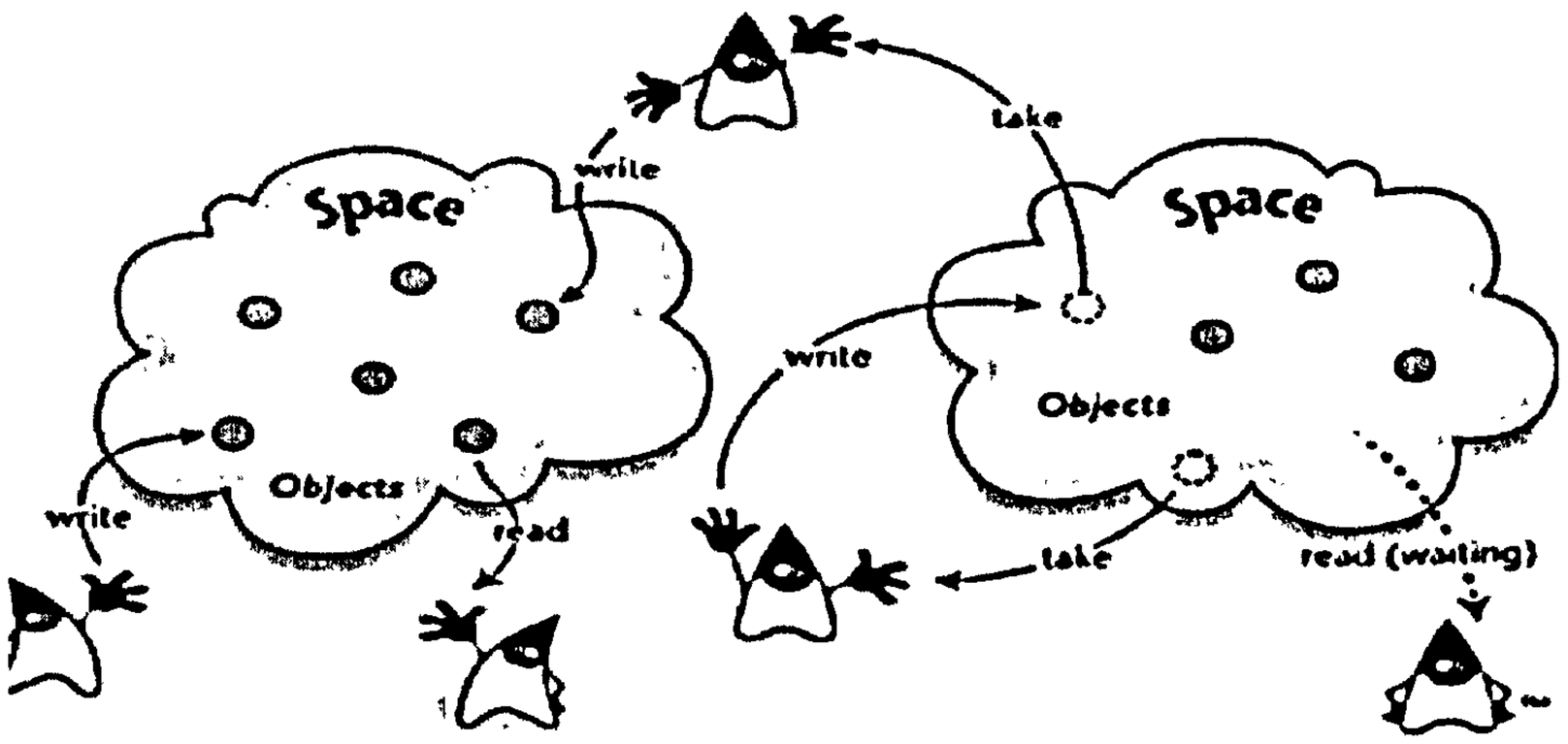


Figure 6.4: Illustration of a distributed shared memory based computation [49].

JavaSpace is an example of a distributed shared memory developed by Sun Microsystems [119] and is included as a core Jini service. JavaSpace provides a high-level means of creating collaborative and distributed applications, the process can *write* new objects into the space, *take* objects from the space or *read* (make a copy of) objects from the space.

6.4 Autonomic Middleware Control Services Layer

As soon as the basic middleware is started and established using the registration service to register services with the same middleware network, and the discovery service for

discovering the registered service. The control service is able to start its autonomic control of environmental behaviour. The control service includes three main parts, these are: the *service manager, JavaSpace* and the *system controller* (see Fig. 6.5 below). Figure 6.5 outlines the sequence of the control process using the design of our autonomic middleware control service, as described below:

1. Each service has an assigned service manager that is activated when the client requests a service; this looks after its service and reports the service status to the system controller either directly (e.g. by remote event) or indirectly (e.g. by the distributed shared space).
2. The service manager posts both a service state and a leasing for its service to the shared space that is used in the case of a "not alive" message being received by the dependent service managers or the system controller; for example, the service status may either be alive or dead, on or off etc.
3. The system controller monitors (observes) the shared space at specified, regular intervals, checking workflow and conflict in service states that is reported by its manager.
4. The system controller detects and repairs conflicts and then reconfigures the system and responds to the service manager with the repair decision.
5. An externalised dynamic document for the repair strategy (e.g. an XML document) is provided to address the intended resolution for a conflict.
6. A system associated interpreter that translates the external format of the strategies document to an executable format.
7. A reconfiguration processes that adapt and reconfigures the whole system according to the new repair strategy.

The next section, will describe each of the three main services (*i.e. service manager. system controller and the JavaSpace service*) of our autonomic middleware control service.

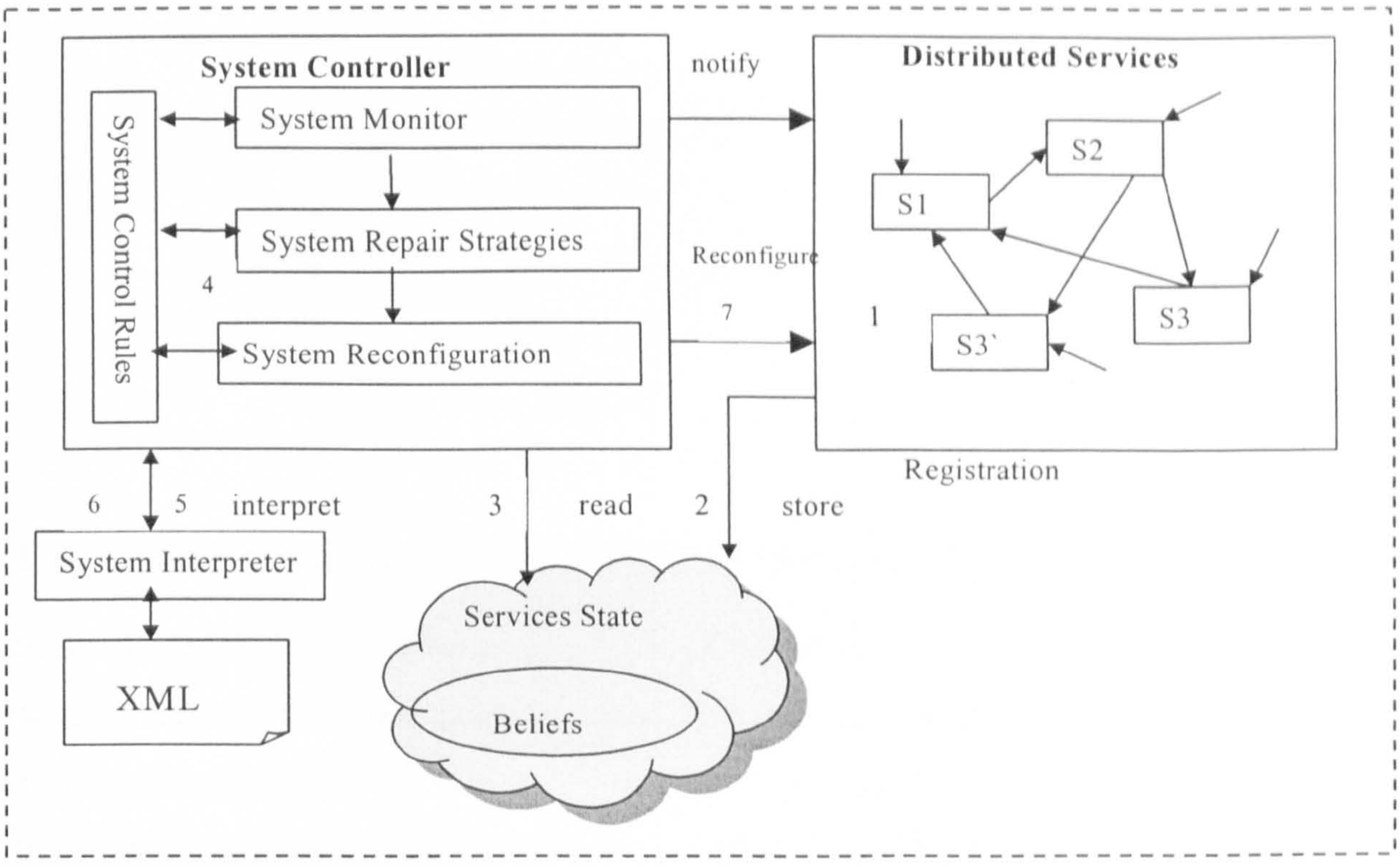


Figure 6.5: The autonomic middleware control service architecture.

6.4.1 Service Manager

Service controller or manager concepts have recently gained popularity amongst the self-adaptive software community as typified by [20]. In this work, the controller is used to adapt the structural components and dynamic behaviour of its service. Structural components can evaluate their behaviour and environment against their specified goals with capabilities to revise their structure and behaviour accordingly [20]. Service managers make use of well-known regulatory models using; *feedback* loops enabling a target application to monitor and regulate its own operation according to its given control model (see Fig. 6.6 below:

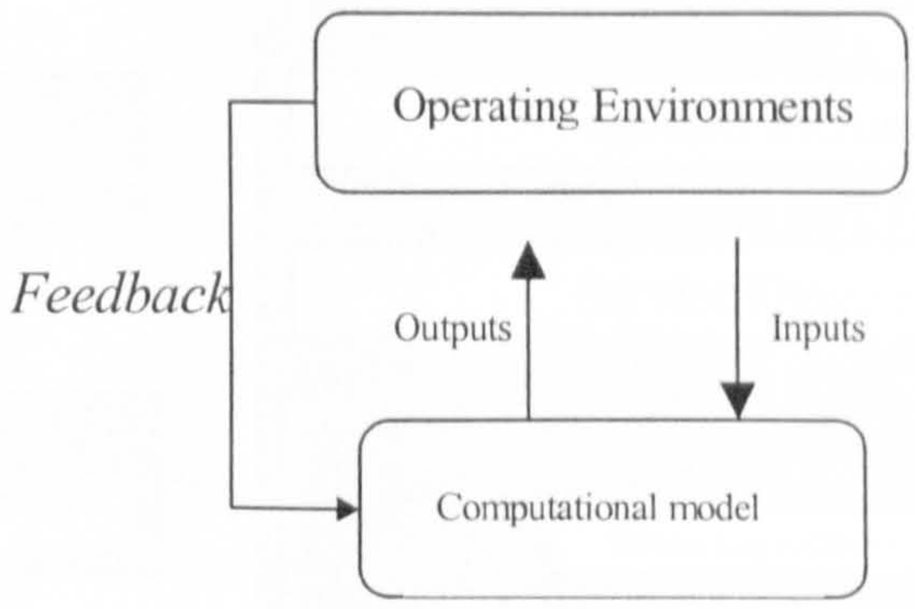


Figure 6.6: Basic computational model with feedback control.

Our choice of using a service manager for each separate service provides the main separation of concerns to our control service by adding this feature of distributed control and management to the service. This separation into distinct service managers eases their management by decentralizing the control of each separate service. Service managers look after their service and monitor its behaviours, via information provided by *instrumentation* services [120].

A service manager is able to *self-detect* and *self-identify* conflicts and *self-repair* its service by formulating remedial action in the form of repair operators. Since conflicts are identified and categorized before repair operators are used to minimize the conflict's resolution time. A monitor element also communicates with the *feedback* process.

A service manager has a sequence of processes (see Fig. 6.7 below), which provide easy and efficient control of the internal workflow of each service manager. The service manager has four main units that are detailed below. These are the service monitor (see Sec. 6.4.1.1), service diagnosis (see Sec. 6.4.1.2), control rules (see Sec. 6.4.1.3), service repair (see Sec. 6.4.1.4), and service adaptor (see Sec. 6.4.1.5).

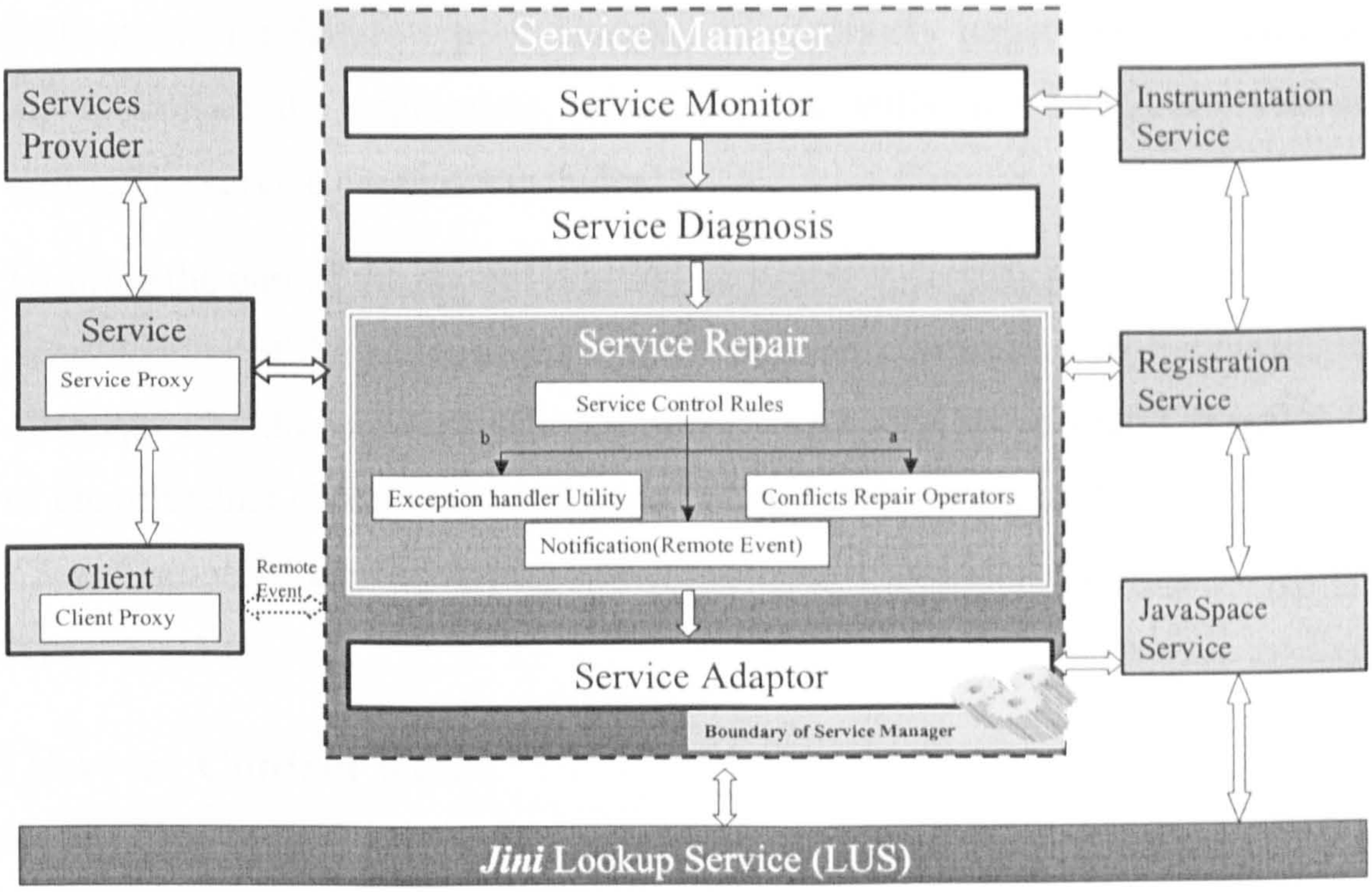


Figure 6.7: The service's manager Architecture.

6.4.1.1 Service Monitor

An essential utility of system self-management and adaptation is its ability to monitor its environment behaviour and translate that information into a high-level system state. In our approach, this is required for monitoring the target system/application to detect and analyse runtime behaviour ensuring it is within acceptable and/or specified bounds. Otherwise, a conflict resolution task is initiated throughout which a continuous monitoring process is used to prevent any mismatch between the desired system behaviour and the actual system behaviour.

The proposed service monitoring relies on ongoing research at Liverpool John Moores University on a Dynamic Instrumentation Framework For Distributed Systems [120], which enables the system to listen, analyse the information that is received from the instrumentation service. Consequently our monitoring process provides the system with a high-level representation and properties of the running system. Such information is essential for the diagnosis and repair processes to represent the current state of the system, or to display an alert dialog about a conflict. Moreover, the information of the monitoring process must be meaningful in some context to the diagnosis process. For example, when the monitoring process receives low virtual memory detection, it is translated as a parameter failure. Another example is a connect () method invocation failure, which results in a *RemoteConnectionException* failure or method failure.

6.4.1.2 Service Diagnosis

The identification of a failure type is essential to accurately target the root cause of errors and to initiate the appropriate remedial plans whilst preventing any further system errors. The service diagnosis includes:

- *Locating* the part of the control rule that generates the conflict.
- *Identification* of the cause of the conflict through examination of the messages intercepted by the instrumentation services where services attributes or methods of the offending object may be acquired using Java's reflection API.
- *Classification* of conflict types, which provides a basis for selecting a conflict repair operator.

6.4.1.3 Service Control Rules

The purpose of the control rules is to measure the service parameters before selecting the resolution or repair operators that are activated by the repair process or allow any required alterations.

Most service constraints or rules are embedded (i.e. internal) within a service manager. Some of the service control rules may not be attached (i.e. external) to any particular point within an action or strategy, but are usedfor checking throughout the execution of every action or strategy while it forms part of a service repair process [121]. For example, if there is a failure in the repair process itself, the service control rules detect that conflict occurred in the repair process itself.

A service constraints or control rules combine the expected service state and the service desired action. These control rules include two main levels, namely, *low-level* parameters like the size of a service (e.g. Kbytes) or the number of clients that access a service (e.g. 75% of the use of a service occurs on Monday and Friday between 9am-5pm) or control rules that measure *high-level* parameters such as remote methods invoked by a client or remote events.

6.4.1.4 Service Repair

The resolution and repair process was specified using a contract-based design, pre-conditions or typical operators. Examples of these are shown in Chapter 7. The service repair is developed to provide operators to resolve conflict. As these operators provide

primitive operations they are integrated into the service manager particularly in its service repair.

Three key tools or actions fir self-repair are required to make them useful for taking the appropriate decision for detected conflicts:

1. The Notification service [122]: to enable clients to register interest in particular messages or events and a one-way response to such notifications. If a particular service wants to subscribe or register interest in particular notification messages or events, it must implement a source of the notification interface to manage the subscriptions or registrations of such notifications [122]. Therefore, to start the notification of a requester service, the subscribed operation or method on the interface is invoked; notification messages then flow from the source to the requester. An event-based mechanism [119] is used, either to receive events from the service diagnosis or send notification to clients and/or servers, whenever a conflict resolution or repair operator becomes available. We use remote event notification in our repair process because there are times when either services or clients want to know directly when services start/stop or are added/removed to provide fast, direct communication and connection. For example, the editor can be notified from the disk service manager that the disk service has started so it is ready for use in saving a file.

2. Repair Operators: As mentioned earlier, the result of runtime constraint is checked; self-repair selects the required operators and operations that support dynamic changes to the service during its lifetime. Thus, instead of breaking a running system or performing unacceptabe behaviour, systems automatically adapt and change behaviour by defining a set of *operators* that identify the means used in such a repair. Oherwise another set of *exception* classes are invoked to establish the safe termination of a system that has failed. *Operators* provide a set of primitive operations for adding and removing components and connections; nevertheless, certain processes can provide operators with a higher level of abstraction. Such operators have been used before in an architectural style model [108] that described architectural configuration changing; although similar in many aspects to our approach we applied some sets of natural operators at runtime for

unpredictable behaviours such as adding, removing clients/servers as way of solving the arisen service conflicts. There is two main factors used to determine the selection of the operator for specific conflicts, *first* the service's attributes and parameters satisfaction and the *second* factor is the availability of carrying out the operator in consideration of the run time operation changing and without exception.

2. The exception Handler: a mixture of heuristics, planning functions and activities is indicated, each representing the classification of a failure and the assumptions that can be made about such a failure. This uses Java's exception handling [37] facilities to catch exceptions. These are thrown when the operator cannot respond to a control rule that is executed because of a conflict in the service repair. Exceptions are dealt with according to priority. These may be low, intermediate or high to accommodate varying degrees of fault-tolerance by three main sequences including (see Fig. 6.8 below):

   - Monitoring Model: which compares the nominal or required system state with the current system state.
   - Classification Model: which distinguishes between the exception kind or class and the exception occurred.
   - Verification and Recovery Model: which inspects certain features and recovers the system again.

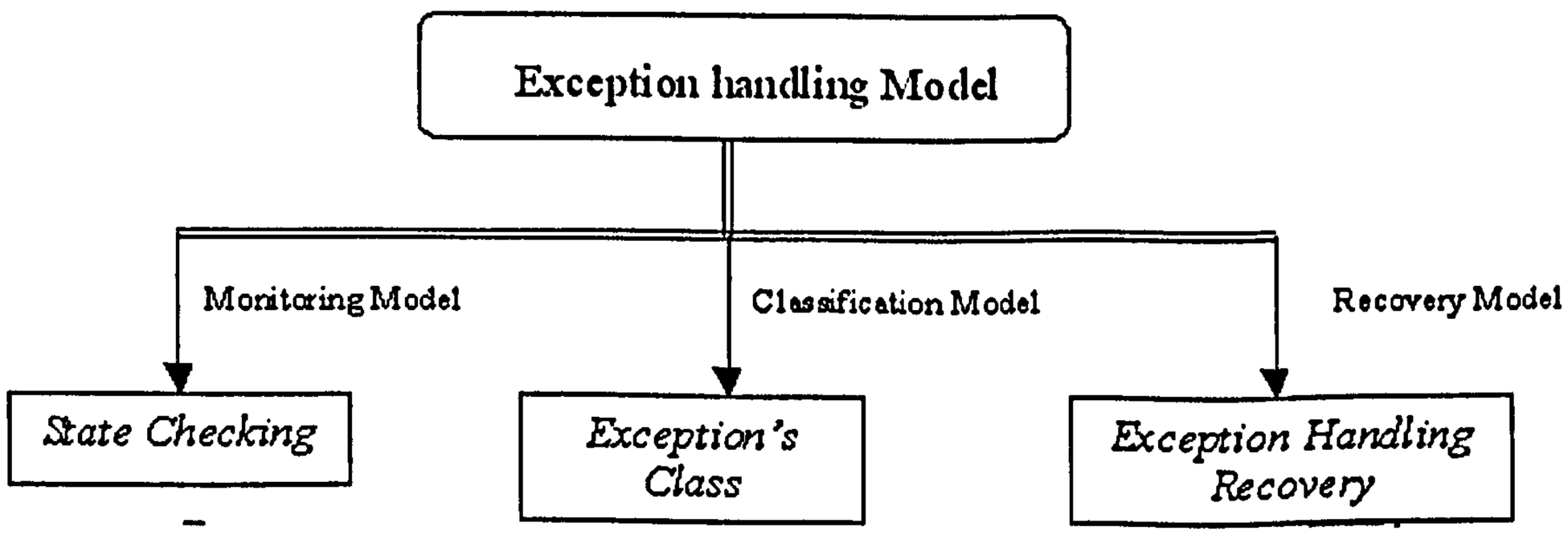


Figure 6.8: The sequence of the exception-handling model (adapted from [62]).

6.4.1.5 Service Adaptor

Adaptation management plays a crucial role in our approach. Although the service manager controls conflicts based on runtime changes, appropriate adaptation strategies are required to be defined and constructed.

Although a major characteristic of self-adaptive software operating in an uncertain environment the self-repair process alone is not enough to satisfy system adaptation needs [15]. An effective adaptation strategy is also required where the parameters and attributes are managed and controlled by:

- A predefined limitation of the service parameters, a range of variation of the possible adaptation.
- To prevent more changes arising by reducing the computational process required for planning or strategies.

There is a general approach to create that adaptation process which depends on the following aspects [123]:

- Pre-condition, the required event type, representing the limitation or other constraints on input parameters that are required for the function to perform adaptation.
- Post-condition, the observer/operator of the generated event types.

Our approach provides a dynamic service adaptation in which the adaptation process is based on the use of a *parametric adaptation loop* construct. Here, the service adaptor receives messages or notification from the service manager, in such a case, the service adaptor can easily determine the changeable *parameters* and attributes of the service for updating/changing during the system lifetime. For example, removing/adding client/service requires changes to the predefined attributes or parameters needed for the change in meanwhile the monitor process continues to monitor the existing adapted service to guarantee that relevant information can be extracted and that the changes are robust in the running system.

Conceptually, the *feedback* loop provides the service adaptation with a return *loop*, which will send the adaptation result back to the service monitoring to determine the success of the selected repair operator and continue the system workflow again.

6.4.2 System Controller

The control service should be enabled to dynamically resolve and coordinate the whole system where the service manager reports a service failure or a service manager lease expires. Therefore, the *system controller* selects an appropriate strategy in order to coordinate their inter-related coordinated services. As such control and coordination is moved from being a design time activity to a runtime selection in order to meet their required circumstances.

Resolution strategies are used to represent the effect of various alternatives solution based on the Beliefs Desires, Intension (BDI) model of deliberative systems[7] [68]. We adopted the BDI model to underpin our autonomic approah by attaching a utility function to the repair process that evaluates the intended strategy to avoid unexpected behaviour coming from an open environment. BDI concepts and example (Fig.6.9) are defined below[68]:

- Beliefs correspond to a specific service and/or environmental state information accessed from the current environment; such information comes from either different sources or the Beliefs of other services.
- Desires represent the state of affairs (i.e. in an ideal world) that the system wishes to reach as a *high-level* goal.
- Intentions represent desires that the system has committed to achieving as a *low level* service goal, which can be immediately transformed into action.

In this section (Sec 6.4.2), we present the design of the system controller service to establish system self-control, interpretation, coordination and reconfiguration, taking into account the system control rules (Fig. 6.10). The system controller is responsible for control of the whole system (i.e. a distributed application service) as central control processor. A such, the system controller has a higher level of control compared to the service manager (see Sec 6.4.1), which has the capabilities to coordinate and integrate between application services with the assistance of the JavaSpace service (see Sec.6.4.3). The system controller service contains three main components, namely; (a) a system monitor, (b) a system repair strategy process, and (c) a system reconfiguration process, which are shown in Figure 6.10 and explained in details below

---

[7] Bratman [62] argued the merits of the BDI architecture for providing a foundation for the design of a flexible reasoning process for intentional systems.

```
BDI-interpreter
  initialise-state();
   repeat
        options := option-generator(event-queue);
         selected-options := deliberate(options);
         update-intentions(selected-options);
         execute ();
         get-new-external-events();
         drop-successful-attitudes();
         drop-impossible-attitudes();
  end repeat
```

Figure 6.9: An example of the event queue of BDI structure [13].

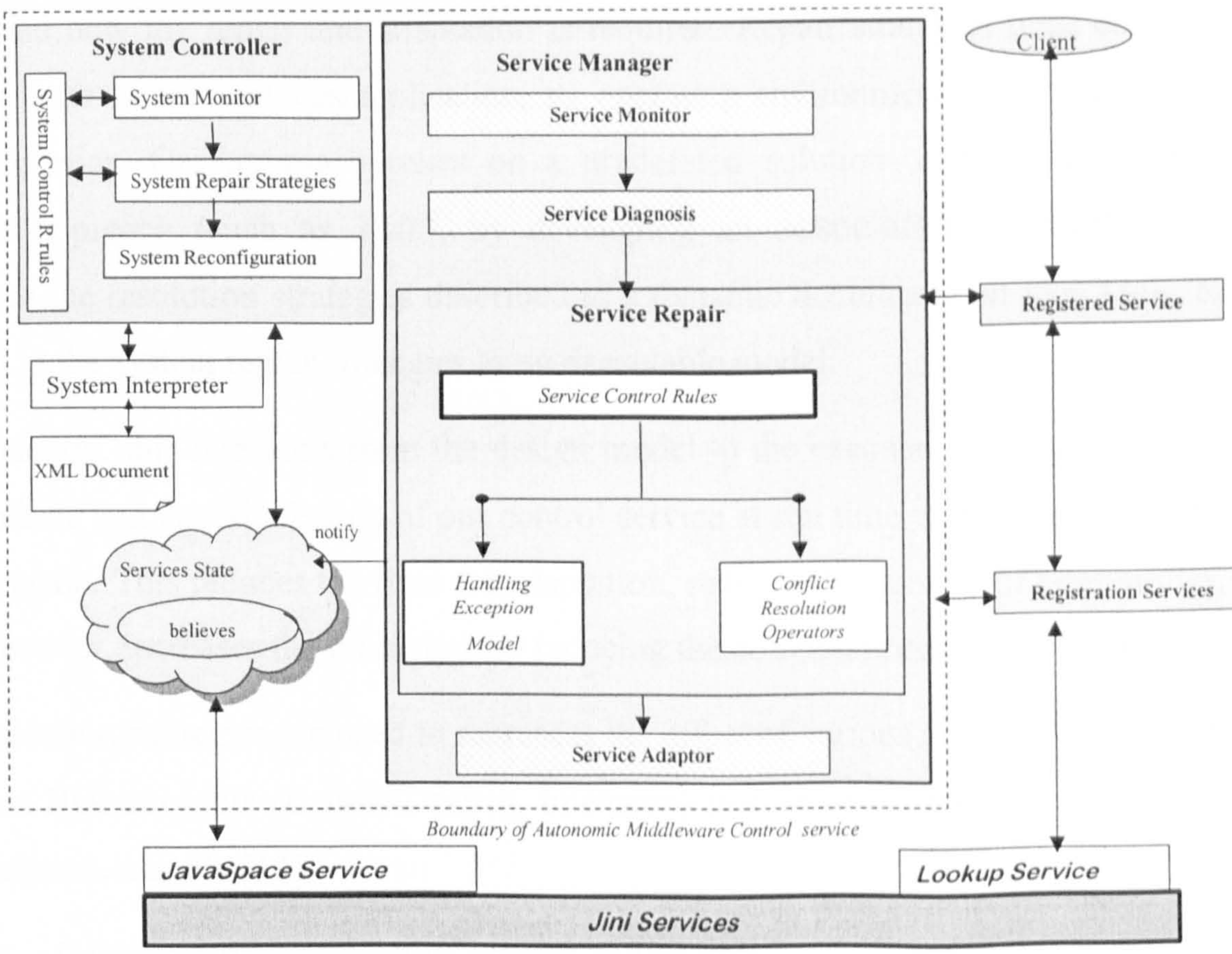


Figure 6.10: The low-level autonomic middleware control service.

6.4.2.1 System Monitor

The system monitor has the ability to collect or receive information that is required to support and guide the resolution strategies within the control process. In the large, open

and dynamic nature of distributed services, the system controller cannot choose suitable strategies and execute appropriate actions without continuous monitoring of its application services. Such system monitoring (i.e. self-detection) is a continuous process and is fundamental for supporting the control and coordination process (i.e. self-repair and self-adaption).

The system monitoring starts either by (a) checking the distributed shared space (e.g. *JavaSpace or T-Space* Sec 6.4.3). (b) Or by using well-specified control functions with feedback loops to monitor and regulate a target application in accordance with its given control process.

6.4.2.2 System Repair Strategies

Repair strategies are also an essential aspect of self-adaptive software. Self-adaptation requires distinct forms of strategies, which are resolution actions; it determines when, where and how the repair and adaptation is required. Repair strategies must consider the functions of the services/application, its operating environment and its attributes and properties. Our approach relies on a predefined solution ,even in a dynamic solution approach (such as *XML*), by developing an associated interpreter that translates the resolution strategies described in a dynamic document/text like XML, to be used in the system repair strategies as an executable model.

Consequently, this interprets from the design model to the executable model and adds the dynamic and generic feature of our control service at run time without recoding the system again. This reduces the time for adaptation, reduces the number of possibilities, and moreover, increases the efficiency by reducing the computational process required.

Our resolution strategies are used to represent the effect of various alternatives solution based on BDI concepts (Beliefs Desires, Intension) that represent the Beliefs, Desires and Intension, namely [13]:

- *Desires*: represent a set of actions and/or desired aims needs to achieve at a specific time.
- *Beliefs*: are represented by two structures. A model of the external world and another of the current internal status of the system.
- *Intention*: is determined by a process of deliberation, which *translates* desires with respect to the current beliefs about both the environment and the 'stance'. For example, *intend to* replace one component with another, the component

specification and *beliefs* should be satisfied and *desired* with one being replaced along with its interface.

Each repair strategy is a sequence of executable actions. Each plan is examined by pre-condition or constraints as described below:

- Examine parameters, properties and rules of each service.
- Determine the applicable plan or strategy to resolve the conflict. Strategy or plans are formed out of a lower level of actions or operators based on the *Beliefs, Desire* and *Intentions (*BDI) design (i.e. explained above). The strategy consists of an identifying number and name. The system control rules determines whether or not this strategy matches system *desires* and selects the appropriate *intension* when the system's *desires* are compared with its current *beliefs*.
- Selecting one valid strategy in a situation where several may be applicable should be decided according to policy, which could be heuristic (such as control rules) or may be sequential like First in First out (FIFO), or other specific constraint and translates such strategy into executable actions.
- Reconfiguration of the whole system after selecting the appropriate solution is the essential phase. The system repair strategy should notify the reconfiguration process of the required reconfiguration needed to establish and provide a faithful "up-to-date" representation of the application component and service configuration at runtime.

6.4.2.3 System Reconfiguration

Determining whether the applicable strategies have updated the system appropriately is essential, as sometimes the result can be catastrophic or bring instability to the whole system [19].

Reconfigurability is an important requirement for distributed application systems, as it allows the system to cope with changes. In addition, the optimal configuration may depend on specific system and situation, as self-adaptive software should reconfigure itself to maintain its optimal configuration.

Therefore, self-adaptive software should itself be reconfigurable. This implies that the very structure of the system can be modified, even when the system is running. This

concept is quite new, compared to traditional concepts of reconfiguration. Normally, the reconfiguration phase uses some calculation on system properties, for example, QoS performance, etc, but the system reconfiguration applies the required reconfiguration via the resolution strategy, which is translated from an XML document using our associated interpreter to translate the repair and reconfiguration operators or strategies from the text format to the executable format. This is achieved without recompiling the whole system again, as the translation and execution of each action is reported and stored in the JavaSpace service and used to notify the system controller to activate the other actions or alternative as show in Figure 6.11 below. This described the autonomic control service behaviours and interactions amongst the interacting services.

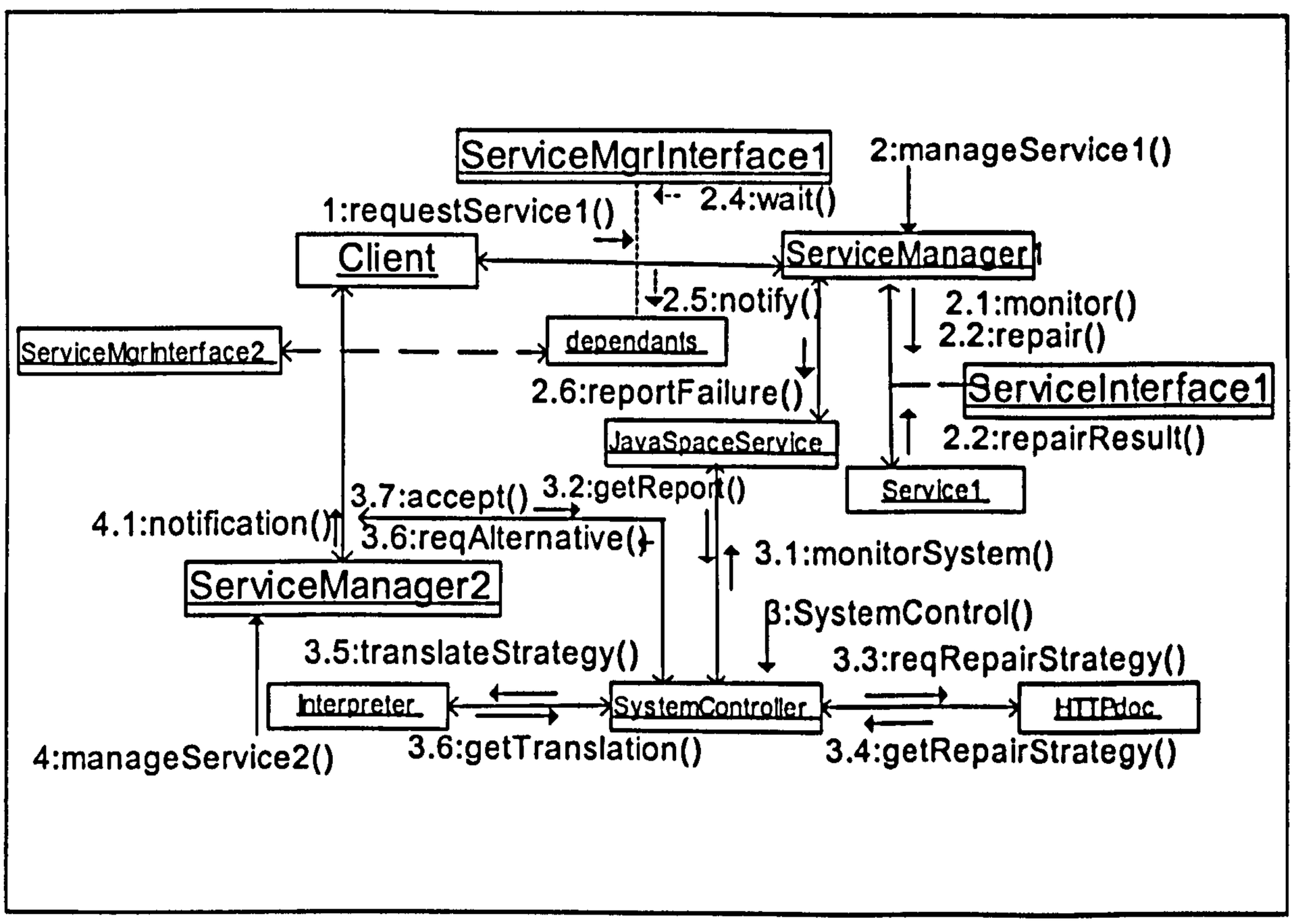


Figure 6.11: The control service's collaboration diagram.

For example, if the resolution strategy select an alternative service instead of a failed one, so the reconfiguration system should establish a required changes come with the resolution strategy such as; getNewManager(), notifyClient(), and newConnect(), dynamically at runtime[8].

[8] We have used Java Reflection API `java.lang.reflect` to invoke methods dynamically from XML, more detail of the implementation can be found in Chapter 8.

6.4.3 JavaSpace Service

The JavaSpace service is a persistent service that provides the autonomic middleware control service with a distributed shared memory/space that is used to coordinate the relationship of shared resources or services in the distributed application over the network. In addition it has the capability to store required information. The JavaSpace Service works closely with both the Service manager (see Sec 6.4.1.1) and the system controller service (see Sec 6.4.2.1), by allowing the service manager to store both the service state and the value of the manager's expired time in the space. This is regularly checked by the system monitoring to determine if the manager still "alive". If the manager's lease time has expired, the system controller will detect the failure of the manager itself and begin the self-repair and self-reconfiguration sequences. The system-monitoring unit could also be notified bythe JavaSpace (i.e. through remote event notification) when the service manager stores the service state and the value of its expired time in the shared space entry.

JavaSapce is not only used by the system monitor unit, but also by the system reconfiguration to check the successful execution of the reconfiguration strategy action and either continue or detect a further execution failure and start to select another repair strategy.

6.5 User's Service Applications Layer

The user's service application (*third layer*) regards the application as a federation of services distributed over a network. A service represents a logical concept, such as a printer or chat service. Services are discovered dynamically by clients and used according to a mutual contract of use. This service-oriented abstraction provided a higher-level or end-user service that should benefit from an autonomic middleware control service, especially in conflict circumstances. These, that may include Commercial Off The Shelf (COTS) services, which are assembled, configured, instrumented, managed and controlled at runtime by the second layer services (i.e. an example of a user service application that used our control service is detailed in Chapter 8).

- The application service could be variations of software, hardware or a hardware/software combination:

- Hardware: services accessed through a standard protocol using a software driver. For example, a printer or scanner.
- Software services: services consisting of software generally accessed by Java RMI and TCP/IP. For example, databases or a chat-room.
- Hardware/Software combination: services accessed through a private protocol. For example, a central lock or alarm clock.

6.6 Summary

The dynamics inherent in distributed applications make it is difficult to understand their behaviour. Furthermore, the possibility of different component technologies and different network protocols can give rise to conflicts and inconsistencies.

The conventional engineering services of service manager, adaptation and system controller can be combined to overcome these difficulties and assist in the runtime management of distributed applications.

The main strength of the proposed design stems from its ability to simplify the overall process of self-control and self-adaptation by factoring out the managerial and adaptation code from application components into distinct autonomic middleware control services.

The design of the decentralized and centralized control and management that are represented by the service manager and the system controller respectively provides a separation of concerns that reduces complexity and integrates application services. In addition, the JavaSpace service, which supports such coordination between distributed services, is used as database service (i.e. different from a relational database) to store service information provided by the service manager or the associated system interpreter for later access by the system controller, also JavaSpace service can also directly notify the service manager or the system controller through the use of the remote event model.

Chapter 7

Manager Service and JavaSpace Implementation

7.1 Introduction

As our challenge is to optimise the management of distributed self-adaptive software and seen as crucial to the enhanced performance of distributed applications, there is also a need to optimise the flexibility and efficiency of end-user application-level services (or components). Such services, which feature significantly throughout this thesis, may either provide services (servers) or use services (clients) or a combination of the two. These services may have been developed using heterogeneous technologies and may be hosted by a variety of different hardware platforms, ranging from powerful computer-servers to small PDA devices. Furthermore, with the increased interest in wireless networks, the level of network connectivity, or more particularly its variation must also be considered in current and future models of autonomic applications.

In this work the software prototype is implemented using the Java programming language and Jini[9][119], which provides a set of APIs and network protocols that facilitate the development and management of distributed systems. In Jini these are regarded as a federation of services. Jini essentially establishes a software infrastructure through which all the devices of a Jini network may communicate, irrespective of their operating system or interface constraints. Each device is said to provide a service, which it does by publishing its own interfaces. Other devices can then find or lookup these interfaces to communicate with the device and thereby access its service. Figure 7.1 below, shows how Jini implements the connection technology below the network application layer and builds on Java technology. Through this architecture, Jini can

[9] A Java-based middleware technology developed by Sun Microsystems [119].

accommodate a variety of devices, including portable devices, electronic consumer appliances and devices not normally found in a computer network.

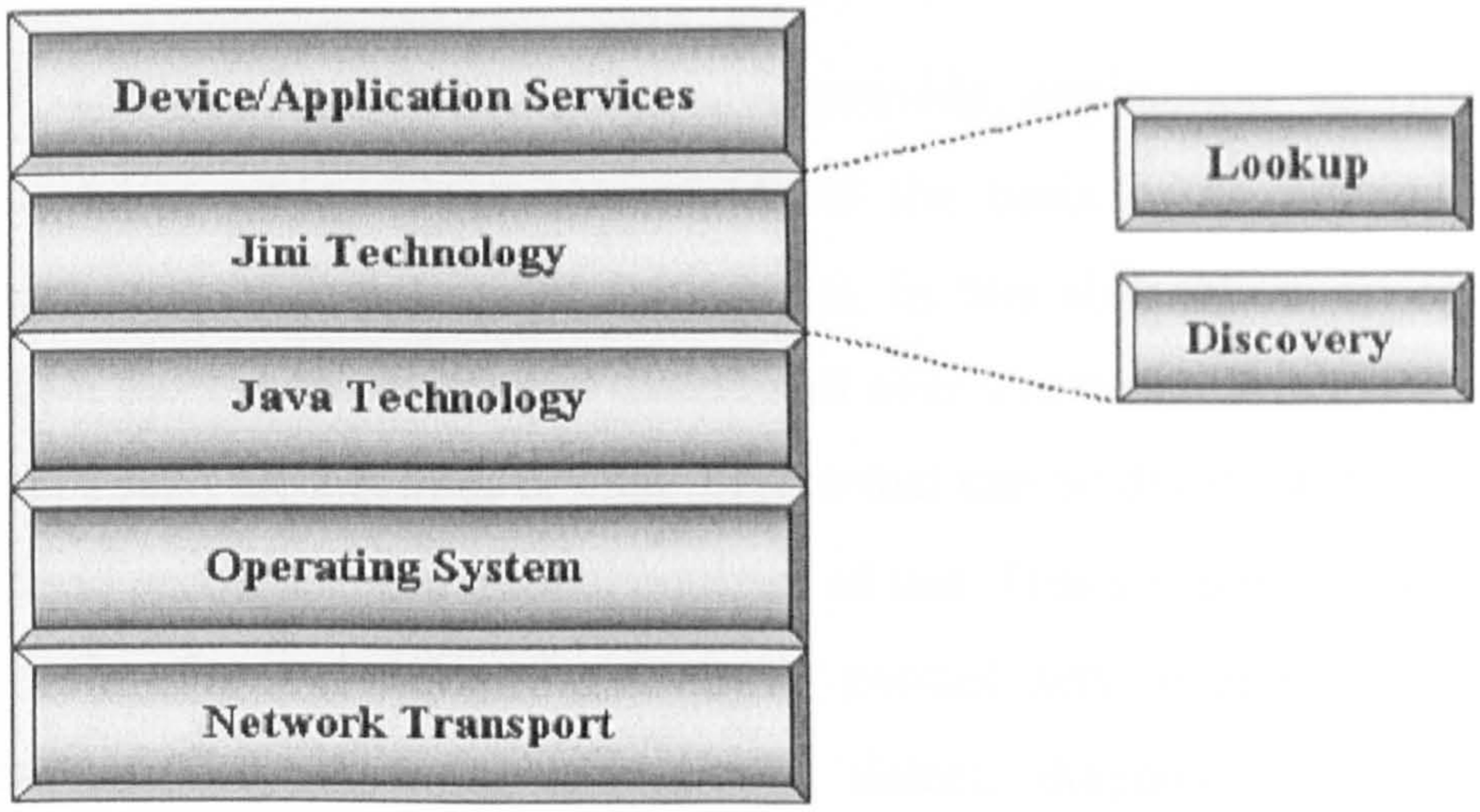


Figure 7.1: Jini network architecture.

As the main aim of this work is to provide an autonomic middleware control service (or simply a control service), to facilitate the lifetime management of application-level components without having to include large amounts of additional adaptation within the components, a programming model was developed through which application services need only implement certain interfaces to provide access to their structural and behavioural properties. This programming model code is referenced and used during the control process. Overall, the approach provides a practical and effective solution that can be used, in conjunction with core middleware services, to facilitate the runtime support of distributed self-adaptive software.

Both this and the following chapter (Chap. 8) will provide a detailed description of the implementation of the proposed autonomic middleware control services, namely; the *Service Manager,* the *JavaSpace service, and* the *System controller service.* This will be followed by a description of the interaction model between the three services throughout the given middleware facilitated control cycle of a software application's self-healing process. In particular, this chapter will present details of the *service manager* (see Sec. 7.3) and the *JavaSpace service* (see Sec. 7.4).

7.2 The Implementation Requirements

Implementations of software services management in a distributed application are self-contained binary implementations, consisting of one or more objects. These objects

occur as instances of the classes that make up a component. Components communicate with each other through connectors that are implemented via software interfaces and distributed applications are often described using a component-connector abstraction.

In addition, components also require and provide application services to other components and/or users. These services form the basis of an alternative service-oriented abstraction of a distributed application. In this abstraction, an application is considered as a federation of services distributed over a network. A service represents a logical concept such as; a printer or chat service that can be discovered dynamically by clients and used according to a mutual contract of use. This service-oriented abstraction forms the basis of our developed autonomic control service and provides a set of services that can be used to automatically detect, diagnose, control, adapt and reconfigure the application services to support the distributed self-adaptive software.

Distributed applications are difficult to implement or manage because of their inherent dynamics and the heterogeneity of their implementation, topology, deployment and network requirements. Middleware technology has come to the rescue by easing and facilitating the development and interoperation of distributed applications. However, it is still required for middleware technologies to control dynamic behaviour with runtime self-management of distributed applications or in other words an autonomic middleware control service.

In this chapter, we describe the implementation of such an autonomic middleware control service that supports distributed self-adaptive software. Our chosen implementation is based on a Java environment [124] and middleware technology (see Sec. 7.2). This implementation is illustrated through the use of two applications of the control services to support three experimental scenarios namely; the *GridPC* and *3in1phone* and a *web-based information service*. The *GridPC* application is used to demonstrate the idea of distributed application coordination using distributed shared memory concepts (Sec. 7.4). The second scenario, the *3in1phone* application, was also built using Jini services in general and particularly the Lookup service and JavaSpace service to emphasis the same idea from the first application. Some additional functionality was illustrated through the use of both the service manager and system controller with respect to the distributed system workflow and the coordination of services for self-governance of the distributed application services over the Jini network (i.e. further detail of the system controller implementation is contained in Chapter 8).

7.2.1 The Java Environment

The choice of the implementation language of Java was based on the following

- The mechanisms provided in Java such as object serialization, Remote Method Invocation (RMI), remote object activation and Java API allow suitable flexibility for lifetime management at runtime.
- A variety of standard and extension libraries facilitate the prototype's implementation.
- The wide deployment of Java JVM is appropriate for experimental and evaluation purposes.
- The safe shutdown/interrupt using exception-handling that facilitates catching and throwing exceptions. These which are thrown when a control rule detects a failure. Exceptions are prioritised as low, intermediate or high to accommodate varying degrees of fault-tolerance.

7.2.2 The Middleware Technology

Middleware is "... *connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network.*" [125, 126] and is essential for sending distributed applications to client/server applications and providing communication across heterogeneous platforms. In particular, middleware services provide operating system functionality; network services and provide the distributed application with:

- Communication and interaction between an application and/or service with each other across the network.
- A programming model that hides network programming and host operating systems complexity and heterogeneity.
- Reliability and availability.
- Capacity scalability without losing functionality.

Distributed applications may be spread across a range of computing platforms connected through fixed networks and/or mobile wireless networks. The interconnectivity between different computers is likely to vary according to end-user needs and the requirements of the computation. Such dynamic behaviour must be monitored and managed effectively, to provide reliable services. We describe the

requirements of a distributed application development and management system (i.e. such as the use of the middleware technology), as follows [127]:

- Dynamic system reconfiguration when system inconsistencies are detected on the network (hardware and software).
- Dynamic system adaptation in the case of adding or removing network computing resources.
- Strategy and patterns for resolving a detected failure at runtime.
- Provision of the required infrastructure and tools to measure, monitor and scale.

According to distributed application requirements, using middleware technology helps to achieve many of these application development requirements. This section provides a quick review of the importance of using middleware technology as a basis for developing the autonomic middleware control service (i.e. for simplicity called the control service). More particularly, by considering the proposed approach as a control middleware service for self-adaptive software, where middleware means raising the level of abstraction of programming distributed applications.

7.2.2.1 The Choice of Jini Middleware Technology

Jini™ is a Java-based middleware technology developed by Sun Microsystems [124], which extends Java RMI to provide middleware services and allows clients to dynamically access services and resources across the network.

Jini™ technology provides a protocol and system architecture whereby devices and services can be added to the network automatically and organized into federations. Jini groups and organizes the application service, which is accessed through Jini protocols and core services. In this section, we will explain the protocols and core services that have been used in the proposed autonomic middleware control service implementation. These are:

Protocols

- Discover/Join: discover a Jini lookup service(s) by the clients/services.
- Lookup: finds the application service proxy based on its name and/or type as a result of a client request.

Core Services

- Lookup/Service Locator: used for application service registration.

- Distributed events: notify services about any changes in their states when it registers interest in particular events.
- JavaSpace: provides accessibility of using shared memory/space for networked JVMs, it may be persistent or transient. JavaSpaces may be accessed using several primitive operations: read, write, notify and take, which is very powerful in large, complex, parallel-distributed applications or computations.

The choice of Jini™ middleware for the implementation part of this work is based on some of Jini™ strengths in supporting dynamic and spontaneous networking, such as [127];

- Addresses the fundamental issue of how services connect and register with the network.
- Knows when services leave or join the network.
- Facilitates user accessibility to the services at anytime, while allowing the network user location (or service) to change.
- Provides tools for developer and programming models that allow the robust and stable development of distributed systems.
- Allows non-Jini devices to join a Jini network through its surrogate architecture [128], which provides a surrogate host to devices that do not have the capabilities to support a full Java virtual machine.
- Provides a service-oriented abstraction.

7.2.2.2 Jini Services Requirements

Jini services require two things:

- A Java Virtual Machine (JVM)
- A TCP/IP connection (stack)

Non-Jini services may still be used through a Jini surrogate host.

Clients use stubs or proxies, implemented as Java interfaces to invoke application service methods, specified by the proxies and relying on Java's RMI to route the invocations over a network.

Jini applications consist of a federation of application services, middleware services (e.g. lookup) and a series of clients. The lower levels of the application service description are component-connector abstractions, where components consist of a

collection of Java objects defined in multiple class files and packaged into a Java Archive (JAR) file. These files can be retrieved through a URL, which is published by the HTTP Server. This value of the URL can be assigned to the system property (*java.rmi.server.codebase*) of the application server. For example, *java.rmi.server.codebase=http://ClientWebHost/classes_files*. Also, the security polices should be assigned to the system property (java -Djava.security.policy). For example, java -Djava.security.policy=policy.all client.ClassFiles [129]).

In a Jini system, clients communicate with servers through proxies (implemented as standard Java interfaces) using the Java RMI protocol. Based on Java RMI, Jini™ API enables distributed applications to be developed as a series of clients interacting with specified application services.

Although Jini™ was chosen as a middleware technology for the implementation of the proposed model of autonomic middleware control services, the design is generic and independent of the language and middleware used. In [130], a Web services based middleware was used to replicate the implementation of the proposed autonomic middleware control service.

7.3 The Service Manager Implementation

The service manager is the most transparent part of the service development and is developed to look after its provision of the control service. In general, a service manager play three essentials roles (a) can be used as a proxy between its service and the client; (b) also each service has a one-to-one relationship with each individual service manager and additionally, (c) the service manager can access its service indirectly by using *Service Interface* (Proxy), or directly using the remote event notification by implementing the *RemoteEventListener* interface, and (d) the service manager communicates both with system controller service and other dependent service managers.

The service management process allows the manager of each service to control its service by self-detecting and self-diagnosing any inconsistencies/conflict, and either recovering the service from that failure or throwing an appropriate exception.

Traditionally, service management approaches involved *either* the insertion of additional software constructs at design-time via compiler directives, or when the

system was off-line, during maintenance to manage specific events and/or control certain parameters. Manual management can be used with distributed applications, but only with limited success ,because of the disturbance and possible shut down of the whole system. For this reason, dynamic automatic management, applied at runtime, has recently attracted the attention of researchers concerned with distributed application development and management [131].

Based on the previous description of the service manager that is attached to each service at runtime to control the service behaviour and based on receiving a client request for a service [132], the proposed control service should be automatically activated.

The service manager consists of three main processes to control service behaviour, or to report the notification of failure to both the system controller to find an appropriate alternative and to other dependent service managers. These processes are:

- ***A Service Monitor*** that is used to receive messages/events about failure/conflict from the instrumentation service (which is responsible for monitoring service performance and process the measurement) and indicating the service behaviour or failure state or by remote event to activate the diagnosis model.
- ***The Service Diagnosis*** that accepts and analyses the received message from the service monitor to classify the type of failure/conflict by the three following sequential steps:
  - Locating by determining which parts of the service constraint are broken/ failed.
  - Identifying the cause of the conflict by the examination of messages intercepted by instrumentation services from which the attributes and methods of the offending object may be acquired using Java's reflection API.
  - Classifying the type of failure, thereby providing the basis to facilitate the selection of a repair operator.
- ***A Service Self-Repair*** specified to use a set of repair operators to control the conflict. These operators are specific to the functionality of the service, since the functionality provides the primitive operators for both the service and its client. In terms of our scenario, we defined three main operators,

namely: *notify()*, *remove_Client()* and *add_Client()*. Each of these operators is bounded with a control rule to examine the validation of the operators.

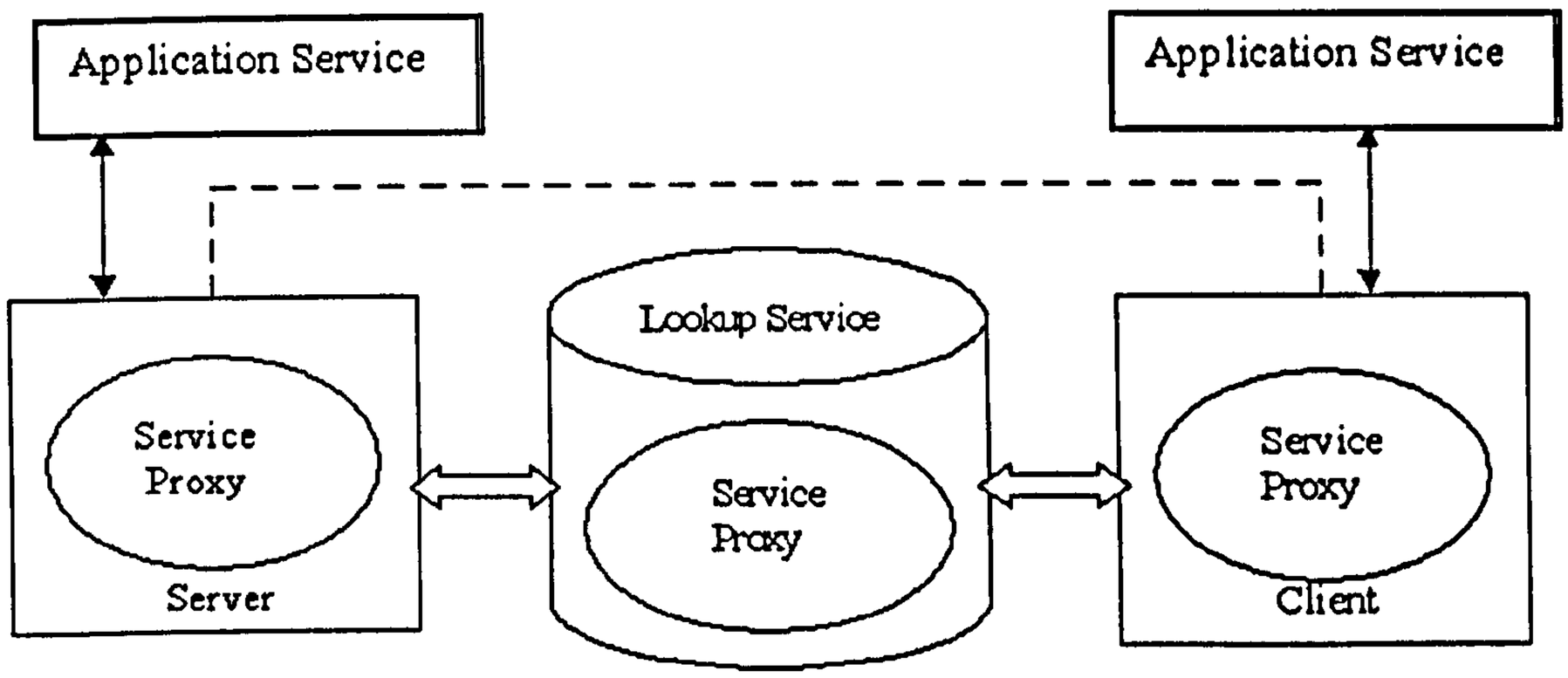


Figure 7.2: Client and sever communication through Jini proxy.

In a Jini system, clients communicate with servers through proxies (implemented as standard Java interfaces) using Java's RMI protocol (Fig 7.2). Clients may locate the service they wish to use through a Jini lookup service, which acts as a trader/broker between the client and server. When a client discovers the service, it downloads or marshals a copy of the service's proxy to be stored locally within the client. Hence the client should able to use the service by invoking the methods specified in the proxy and these invocations are routed back to the server via Java RMI. This model of client-server communication is illustrated in Figure 7.2 above for a simple Jini system, where the block arrows represent the marshalling of proxies between server, lookup service and client.

7.3.1 Service Manager Interactions

The service manager communicates with its services and its environment by implementing their proxies (see Fig. 7.3, 7.4 below), where the manager (e.g. *ServiceManager.class*) can control its service by sending and receiving events of the method invocations made on the service (e.g. *SimpleService.class*) via the service proxy (e.g. *SimpleServiceProxy.class*). The classes *ServiceManager.class* and *SimpleService.class* are extensions of the *UnicastRemoteObject* class. The created stubs *ServiceManager_Stub* and *SimpleService_Stub* are invisible to the programmer.

A ServiceManagerProxy definition

```
public interface ServiceMangerProxy implements Remote, RemoteEventListener{
    public void getSimpleService( ) throws RemoteException;
    public void serviceDiscovery( )throws RemoteException;
    public EventRegistration  addRemoteEventListener
(RemoteEventListenerlistener,MarshalledObject    handback) throws RemoteException;
   public void notify (RemoteEvent event) throws UnknownEventException,
RemoteException;
}
```

Figure 7.3:The implementation interface of ServiceManagerProxy.

A SimpleService Interface definition

```
public interface SimpleServiceProxy implements Remote{
    public void connect( ) throws RemoteException;
    public void disconnect( ) throws RemoteException;
    public int getServiceArgs( ) throws RemoteException;

}
```

Figure 7.4: The implementation interface of SimpleServiceProxy.

We will illustrate the required sequences/subsequence of implementation steps that allow the communication between the service manager and its service, and between the service manager and both client and system controller (i.e. the client can communicate to the service through its service manager and the system controller communicate with the service manager for a report of its service state) as follows (see Fig. 7.5, and Fig. 7.6 below):

1. *ServiceManager* generates its *ServiceManager_Stub* by use of the *rmic* compiler.
2. *ServiceManager* implements the *Remote* interface allowing the methods of the *ServiceManager* (*detailed below*) to be called from its proxy, and these methods could be invoked remotely.
3. *ServiceManager* either inherits from the *UnicastRemoteObject* allowing RMI to export the *Stub*, which requires its constructor to generate and export a proxy or stub object that hides much of the programming detail and complexity away from the programmer, or calls the *UnicastRemoteObject.expert()* method as an alternative of *UnicastRemoteObject* inheritance.

4. *ServiceManager* contains the *getSimpleService()* method, which should retrieve *simpleservice* as a registered service with the lookup service, using *registrar.Lookup(template)*, where *SimpleServiceProxy* is as a template for matching and casting in the search process.

5. *ServiceManager* communicates with its service as soon as the client establishes its request. On the other hand, the *ServiceManager* gets the service proxy by implementing its service interface to establish both direction sides of the connection over the network.

A SimpleService definition

```
public class SimpleService  implements SimpleServiceProxy{
    public void connect ( ) throws RemoteException{
        //service connection constrains
          Service! = null;
          binding= Binding. Free;
          Service.num_connections < service.max_connections
          // call connect() methods
          binding= Service.connect ( );
      }
    public void disconnect ( ) throws RemoteException{
            / / …. code to disconnect client
    }
    public int getServiceArgs( ) throws RemoteException{
        // code to get  service parameters value and store it in variable service_args
          return service_args;
        }
}
```

Figure 7.5: The implementation of the simple Service.

Therefore, our implementation uses the previously illustrated communication between the service manager and its service, client, other dependent service manager and system controller for communication or sending/receiving remote events between them. We have used such communication from the Jini Remote event (i.e. simply RemoteEvent) to facilitate the communication efficiency between the service manager and the other mentioned services.

A ServiceManager definition

```
public class ServiceManager extends UnicastRemoteObject implements ServiceMangerProxy {
ServiceTemplate template = null;
SimpleServiceProxy ssp = null;
public void getSimpleService throws RemoteException {
try {
      Class[ ] classes = new Class[ ] { SimpleServiceProxy.class};
       template = new ServiceTemplate (null, classes, null);
       ssp = (SimpleServiceProxy) registrar.lookup(template);
} catch(java.rmi.RemoteException e) {
     eAll.printStackTrace();
     System.exit(1);
   }
   catch(java.lang.Exception lang) {
     lang.printStackTrace();
     System.exit(1);
   }}
public EventRegistration  addRemoteEventListener (RemoteEventListener listener, MarshalledObject
handback) throws RemoteException{
try
     {
       listeners. put(listener, handback);
       return new EventRegistration (long eventID, java.lang.Object source,  Lease lease; long seqNum)
      }
     catch(Exception e) {
       e.printStackTrace( );
       return null;
     }}
// The method manager_monitor()
// The method fireNotify(long eventID, Lomh seqNum)
//The method notify(RemoteEvent event)
}
```

Figure 7.6: The class implementation of the ServiceManager.

7.3.2 Service Control Rules

The *ServiceManager* has its own control rules defined in terms of the Service Constraints and parametric control-loop (see Sec. 6.4.1.5), which can be used in the service monitor, service diagnosis and service self-repair. Nevertheless, the control rules may not be attached to any particular point but used throughout the execution steps for checking [41].

The control service contains several types of control rules or constraints according to the design or functionality. For example, control rules to check the validation of the service's parameters. Similarly to [41], our implementation is concerned with two types of control rules, namely;

- The *low-level* control rules: derived from services attribute variables generated during a declaration. These variables have a defined boundary range. For example, the number of clients accessing the service should not

exceed the maximum number of clients and could be described as service.num_connections < sevice.max_connections.

- The *high-level* control rules: generated as a result of invoking method(s), to examine the messages/events coming to/from the invoked methods. For example, examining an event coming from a remote method that was invoked by a client.

7.3.3 Service Monitor

The *monitor* of a service manager communicates with the *Instrumentation services* [120] in a similar way to *SimpleService* (Sec. 7.3.2) via the *ServiceManager_Stub*. The monitor detects unexpected behaviour, by examining information provided either by an instrumentation service or by the feedback loop that continuously monitoring the service's behaviour at run time using a set of control rules (Sec. 7.3.3) against which behaviour is monitors to detect any conflicts or inconsistencies.

The service monitor starts by invoking the *manager_monitor()* method, which create an object of the SimpleService interface(SimpleServiceProxy ssp) to get the service attributes by invoking the method ssp.GetServiceAttrs(). This returns the SimpleService parameters.

```
The Service Monitor Declaration
public int manager_monitor( ) throws RemoteException{
int first_parameter=0;
  try{
     first_parameter = ssp.getserviceAttrs()  ;
  }catch(RemoteException e){
    e. printStackTrace( );
  }finally {
       //return the control to Admin
  }
       retrun first_parameter;
}
```

Figure 7.7: The implementation of monitoring model functionality.

For example, as in Figure 7.7 above, to monitor the value of the service parameters, the value should be checked by the control rules to compare between the current value and the desired value. Finally, the model reports whether the current service state is nominal or not. In the unnominal case, the process continues through the diagnosis and repair models.

7.3.4 Service Diagnosis

Once the service monitor has completed the conflict detection process, then a message or event is sent to the *service diagnosis* to locate and classify conflicts. The service diagnosis identifies and categorises the conflicts to minimize and facilitate resolution (i.e. the service repair). Conflicts can be categorized using a number of different dimensions, such as the type of rule that was broken, the type of action that caused the failures/conflicts and the importance of the failure/conflict [41]. In our implementation, we selected two types of conflicts. These are:

- Conflicts that arise from the rules that were broken. For example, the service attribute is outside out of its acceptable boundary, so the control rule was broken and thus a *service attributes* conflict has arisen.
- Conflicts that arise from the type of action causing the conflict. For example, conflict that arise from event or action trigger (e.g. Exception) called RemoteConnectionException, so that a conflict arises from calling the method connect(), thus a method invocation conflict results.

Figure 7.8, outlines how the manager_diagnosing( ) method implements the *service diagnosis* and the conditional triggering of the appropriate type of conflict resolution strategy (actions).

The Service Diagnosis Declaration

```
public void manager_diagnosing( ) throws RemoteException{
   Boolean conflict_detected = null;
   ConflictType conflictype= null;
      try{
            If (service.num_connections == service.max_connections){
               conflict_detected = true;
               conflictType.serviceAttrsConflict( );
     } catch(RemoteConnectionException exp)
        exp. printStackTrace( );
         //then try the appropriate solution etc.
     }
}
```

Figure 7.8: Diagnosis model to classify conflict types.

7.3.5 Service Self-Repair

Conceptually, by using the previous service monitor and diagnosis, the service manager should be able to establish the required solution to resolve a detected conflict. This can be effected *either* by *service self-repair* operators which use primitive operators such

as: notify (), add_Client( ), remove_Client( ) (see Fig. 7.9 below) or by using an *exception-handling* that throws the appropriate exception. For example, the strategy for solving the RemoteConnectionException type failure may *either* be removing one client who has a low priority *or* to wait for a predefined time according to the control rules. The self-repair primitive operators are:

1. **`Service Self-Repair operators,`** which are triggered by the Service control rules to choose the appropriate operators and perform the required changes for the self-repair process or to catch the proper exception (as in case 2) in the case of failure to repair. The repair model operators are described as follows:

   - notify (RemoteEvent event), which notifies the listener who has registered an interest in a particular remote event. The client uses a method called addRemoteEventListener() to register interest in listening to a particular event (e.g. using listeners.put(listener, handback). The result of that registration is to return a new EventRegistration (long eventID, java.lang.Object source, Lease lease, long seqNum) with the parameters; eventID, *Source, lease* and *seqNum*. On the other hand, the method fireNotify() notifies a service of the triggered event (*which is registered by the addRemoteEventListener methods*) immediately. Thus, the listener (SimpleService) can be informed by the event generator calling fireNotify(), which then executes the notify () method for each listener.
   - add_Client( ) adds a new user to the list of clients that are using the service as soon as the service's constraint/rules are checked. For example, service.num_connections<sevice.max_connections, where *num_connections* is the current number of connected clients and *max_connections* is the maximum number of connected clients.
   - remove_Client selects one client to be removed or to end its connection to particular service by considering the rules for removing or disconnecting any service. In our implementation invoking the method remove_client() is invoked, if the number of

connected clients is larger than or equal to the maximum number of connected clients. For example, removing a client who has a low priority or has been connected for more than 30 minutes. If the self-repair process successfully removes an old client, a new client is given the opportunity to connect, otherwise an exception is thrown (As in case 2).

2. **`Exception-Handling,`** which can be generated from the network-centric nature of Jini [38]. For instance, an exception could occur when the services link disappears, the server machine has crashed, the service provider has died or a problem has occurred with the HTTP server that delivers the service. Many of these exceptions specify their own exception types, such as LookupUnmarshalException (which can occur when unmarshalling objects). Also, a large number of exceptions are simply a RemoteException, which has a detail field for the wrapped exception [38]. The exception-handler adapts to these exceptions, for example, (i) to catch them. (ii) To ask if the program can continue. For instance, the *ServiceRegistrar.lookup( )* can fail to indicate some network error in the connection with a particular service locator [38]. (iii) To terminate safely, for example when the program states have been corrupted but not irrecoverably. For instance, the *LookupDiscovery* constructor can fail to indicate a critical network failure rendering the program unable to repair such a failure, thus terminating with an error value [38], and (iv) To exit when the program state has been irrecoverably damaged. If one part of the program can exit with a non-zero or an abnormal/error value, then a successful exit should signal its success with its zero exit value. If this is not done, then the exit value becomes indeterminate and of no value to other processes which may wish to establish whether the program exited successfully or not. The methods invocation of the conflict repair operators like `connect()`, may result in a special type of exception such as a `RemoteConnectionException`, due to the unavailability or failure of the service. Our service repair model, in the `ServiceManager`, uses Java's exception handling facilities to catch exceptions that are thrown when a control rule/constraints triggers as the result of a conflict that cannot be solved. Exceptions are dealt with according

to priority thereby accommodating varying degrees of fault tolerance. Whenever an exception is caught, these results throw a suitable exception type. For example, if the remote event failed to reach the listener, an exception will be thrown in the first catch block, e.g. an IOException exception to catch `UnknownEventException` in the first catch block or ClassNotFoundException in the second catch block.

The Notification declaration

```
protected void fireNotify(long eventID, long seqNum) {
   if (listener == null) {
      return;
   }
   RemoteEvent remoteEvent = new RemoteEvent(this, eventID, seqNum, null);
   listener.notify(remoteEvent);
}
public void notify(RemoteEvent event)throws UnknownEventException, RemoteException{
     try {
      switch ((int)event.getID ( )) {
       case 0:
         System.out.println("Event ID 0");
         ssp.add_client( );
         break;
       case 1:
         System.out.println("Event ID 1");
         ssp. remove_client( );
         break;
       // ...
       default:
         System.out.println("Unknown Event ID");
         break;
     }
   }
 catch(IOException e)
     {
       throw new UnknownEventException("IOException: " + e.getMessage ( ));
     }
 catch(ClassNotFoundException e1)
     {
       throw new UnknownEventException("ClassNotFoundException: " + e1.getMessage ( ));
     }
}
```

Figure 7.9: An example of notification between the ServiceManager and its service.

7.4 JavaSpace Service Implementation

Distributed systems are hard to build. They require careful regard of problems that do not occur in local computation. This section describes the architecture of the JavaSpace

service, which is designed to help to solve two related problems, namely; (i) distributed persistence service and (ii) distributed applications coordination.

JavaSpace is essentially an optimised Java version of the original tuple spaces [133], and thus, unlike Linda tuple spaces [49] it runs on many platforms. JavaSpace services use RMI and the serialization feature of the Java programming language to accomplish these goals [134]. In addition, a JavaSpace service holds entries that are a specified, typed group of objects. These are expressed in a class for the Java platform that implements the interface net.jini.core.entry.Entry. Use of these entries supports some methods/operations that let clients/developers use entry objects. JavaSpaces provides three types of operations:

- `write` operations- that store one or more entries, usually for future matches/uses.
- `read` operations-operations that search for entries matching one or more templates.
- `take` operations-operations that return one or more entries.
- `notify` operations-used when an entry that matches a specified template is written. This is done using the distributed event model contained in the package net.jini.core.event.

It is possible for a single method/function to provide more than one of the operation types. For example, if the *matched template* is returned in a given method. Such a method can be divided into two operation types (read and take). Matching the template uses entry objects of a given type, whose fields can either be wildcards (i.e. null references) or have values (i.e. references to objects). If we consider T is a template for matching against entry E, so a field with values in T should have the same values in E. Wildcards (null values/references) in T, match any value in the same field of E.

The chosen implementation of JavaSpaces technology provides a mechanism for system coordination by storing a group of related objects and retrieving them based on a value-matching lookup for specified fields. This allows a JavaSpace service to be used for storing and retrieving objects (entries) on a remote system. We implement a `BasicEntry` class, which is used as a factory for creating the required entry and implements the Entry interface. Each entry has two attributes for providing the service

manager with the ability to store both the service state and the service manager leasing time (see Fig. 7.10 below).

```
public class BasicEntry implements Entry{
public String name;
public Integer value;

public BasicEntry(String ServiceState) {
  this.name = name;
 }
public PlanEntry(String ServiceState, int leasingTime) {
  this.name = name;
  this.value = new Integer(value);
 }
public Integer increment() {
  value = new Integer(value.intValue() + 1);
  return value;
 }
public Integer decrement() {
value = new Integer(value.intValue() - 1);
return value;
  }
}
```

Figure 7.10: The implementation of *BasicEntry* to implement the *Entry* Interface.

JavaSpace provides our approach with a number of features, such as, (a) it simplifies the dynamic communication, coordination and sharing of objects between network resources, (b) it acts as a virtual space between providers/servers and requesters/clients of network services and finally, (c) it allows participants in a distributed solution to negotiate or exchange tasks, requests and information in the form of Java objects.

7.5 Summary

This chapter discussed the approach used for the *service manager* and *JavaSpace service* (*i.e. two main services in our control service*) implementation. The system architecture of the current implementation is integrated with the Jini software architecture and based on a Java environment. We explained how our approach is used to automatically detect, manage and coordinate distributed application services. In particular, the implementation description and details of two main services in our control service, namely the service manager and JavaSpace service are provided.

The service manager supports self-detection, self-diagnosis and self-management during the runtime of its service. In particular, the service manager dynamically "manages" its service using a service monitor and diagnosis to detect, identify and classify a failure once it occurs in the service. As soon as the service manager detects

and diagnoses any failure or inconsistency, the service self-repair unit is activated to begin repair by using the repair operators (e.g. notify (), add_Client (), remove_Client (), and catch (Exception e)).

The second main service is a JavaSpace service that is notified from the service manager with the service state, which may be either be available or not (i.e. zero or one) and writes its service state as a JavaSpace entry for sharing and facilitating access or read from the JavaSpace service by the system controller (*i.e. the system controller is the third service in the autonomic middleware control service and its implementation is detailed in Chap. 8*) . The system controller is responsible for the system control processes used when its service manager does not resolve a conflict.

Chapter 8

System Controller Implementation and Applications

8.1 Introduction

This chapter completes the implementation aspects of the *system controller*[10], which is the third main service of the proposed autonomic middleware control service. In addition, three illustrative examples namely, GridPC, 3in1phone and a web-based information service are used (see Sec. 8.3) to present the implementation details of the proposed autonomic middleware control service, and the programming and interaction model to support such application services and the three main control services namely; service manager (Chap. 7) and JavaSpace services (Chap. 7), and system controller (Sec. 8.2).

8.2 The System Controller

As the control service facilitates the management of the distributed application services at the middleware level, the cooperation and coordination of components or service is a crucial aspect considered in our control service. Such system coordination is provided by the system controller service (see Fig. 8.1 below). The system controller service provides runtime self-control to govern applications' self-adaptation. This is achieved primarily through direct communication and supervision of an associated JavaSpace (Sec. 7.4) to ensure coordination between the application services. The system controller service provides a set of tasks including; system self-monitoring for detecting runtime changes in its services states, dynamic software system management that allows a developer to dynamically maintain, control and reconfigure services with

[10] The other two main services namely, the service manager and JavaSpace service are detailed in Sec. 7.3 and Sec. 7.4 respectively.

respect to services coordination and integration, and services sharing resources and common environments.

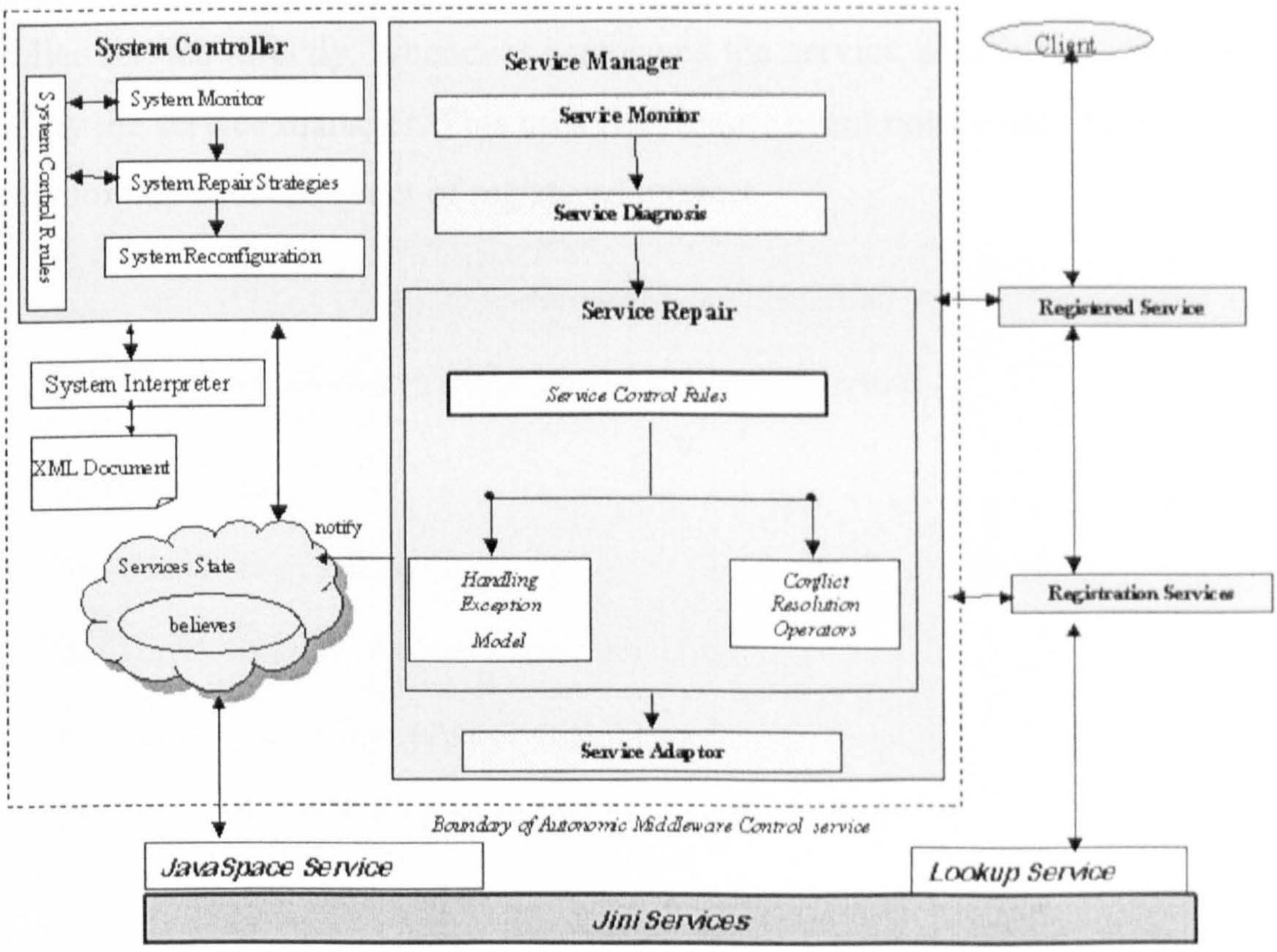


Figure 8.1:The interaction between the system controller and other services.

8.2.1 System Monitor

The system monitor checks a given application's runtime behaviour by examining its current state (*beliefs*) against its *desires*, both of which are stored in an instance of the JavaSpace service. There are two possible ways to facilitate interaction between the system controller and the JavaSpace service, namely;

Indirect interaction: through a system controller proxy[11] to retrieve regularly a service's beliefs (i.e. services current state) that have been sent/written to/in a JavaSpace by its service manager. The service manager initialy tries to solve its service conflict/failure, if this is not possible then the manager writes service's current state including; beliefs and execution status in the associated JavaSpace, providing awareness information such as the execution and workflow status. Such information is monitored by the controller service using its dedicated instrument/sensor – referred to here as the system monitor (see Fig. 8.2 below). The latter implements a set of operators, including sense, analyse

[11] Implemented here by the SystemControllerProxy.class.

and actuate, which are achieved by reading the JavaSpace, analysing the specified model and acting/notifying the Controller Service of certain events (see Fig. 8.2 below). Direct interaction: through a remote event, the JavaSpace service notifies the system controller service directly, whenever it receives the service beliefs (i.e. services state) posted by the service manager. This uses Jini remote event notification for notifying the system monitor with the event of registered interest .

A monitorServiceState() declartion

```
public void monitorServiceState throws RemoteException{

String str = null;

BasicEntry service_state =null,

BasicEntry serviceTemplate = new BasicEntry();

JavaSPace space = SpaceAccessor.getSpace();

BasicEntry myService = new BasicEntry (str,0);

try{

    service_state = (BasicEntry) space.read(template, null, Long maxValue);

}catch (remoteException rem_exp) {

  rem_exp.printStackTrace();

 }

}
```

Figure 8.2: The declaration of monitor the service state from the JavaSpace.

8.2.2 System Repair Strategies

The proposed control service is used to regulate and coordinate runtime software change including; architectural transformation and reconfiguration using both the repair strategies and the requirements model of the application service under consideration. The system repair strategies module implementation is based on a proposed extensible Beliefs, Desires and Intension (EBDI) model. As described in Section 7.4.1.1, the BDI model was first proposed by Bratman [68] as a design for deliberative software agents. Various extensions to the BDI model were proposed to address some of its documented weaknesses, including; revised BDI and normative BDI. The proposed EBDI provides a

highly suitable architecture for the design of situated intentional software that continuously monitors and/or observes their environment and acts to change in accordance with their situated BDI, grounded in normative settings, Here;

- Beliefs; correspond to service information derived and/or accessed from a range of sources, including; domain, environment or beliefs of other services.
- Desires; represent the state of affairs (i.e. in an ideal world), which often maximise the service's own goals. By comparing a system beliefs set (observed system states) against its desires, the system may detects a mismatch and triggers (instantiate a set of intentions) [68].
- Situated intentions; representing action sets for the system to undertake in a given situation to achieve its specified desires and/or to address the mismatch between the system environment (beliefs) and the system's desires (goals).
- Normative intention; representing a set of actions to be undertaken to ensure a specified set of norms including obligation and responsibility rules are observed before a given intention is enacted and/or affective rules emerging as a result of an enacted intentions set.
- Utility intention; represents a set of system actions to optimise its goal-oriented intentions.

In this implementation and in accordance with [135, 13], the beliefs, desire, goal and intention can be described as being collections of constraints, each of which represents distinct pieces of beliefs, desire or goal and so on. These constraints are generated using service beliefs and desires. In this implementation, beliefs are a runtime service's states such, as a service is available for client requests or not. Intensions are the system actions (execution), which are, for instance, triggered because ofa mismatch between the system beliefs and system desires sets using its norms.

To ensure fault-tolerance of the proposed controller and self-repair service, the system beliefs and desires are here stored in a JavaSpace until the decision and normative intended repair plan is completed and enacted. Such architecture has the benefit of providing a simple coordination mechanism for distributed computation (Chap. 7), and an awareness module storing a service's beliefs and desires, repair plan and workflow.

In the case of detecting service failure, the intention set (repair plan) is executed by applying the appropriate resolution strategy and using the system self-repair strategies.

These are implemented in Java and integrated with the control service for runtime execution. In line with system self-repair strategies the execution representation is structured as follows:

The strategies are a high-level model and contain a lower-level called *plan* or *action.* Each plan is formed from another lower level of *properties* applied after successfully examining and checking a list of control rules/constraints.

Each strategy has a specified scope or context of application. They can also be updated and/or added to at runtime. Thus, an externalised strategies repository was generated, where strategies are encoded in XML and validated using an XML schema (Fig. 8.3) that supports the dynamic context externalisation of the application's strategies generation at runtime.

```
<?xml version="1.0" ?>
- <!--

AUTONOMIC_MIDDLEWRE_CONTROL_SERVICE.XML -->
- <!-- Microsoft xml schema showing the Elementtype -->
- <!-- element and element element -->
- <Schema xmlns="urn:schemas-microsoft-com:xml-data">
    <ElementType name="property" content="mixed"
      model="closed" order="many" />
  - <ElementType name="Properties" content="mixed"
      model="closed" order="many">
      <element type="property" minOccurs="1" maxOccurs="*"
        />
    </ElementType>
  - <ElementType name="Action" content="mixed" model="closed"
      order="seq">
      <element type="Properties" minOccurs="1" maxOccurs="*"
        />
      <element type="property" minOccurs="1" maxOccurs="*"
        />
    </ElementType>
  - <ElementType name="Strategy" content="mixed"
      model="closed" order="seq">
      <element type="Action" minOccurs="1" maxOccurs="*" />
      <element type="Properties" minOccurs="1" maxOccurs="*"
        />
      <element type="property" minOccurs="1" maxOccurs="*"
        />
    </ElementType>
</Schema>
```

Figure 8.3: An example of XML schema for our system repair strategies.

A basic interpreter is used to map and bind repair strategies' tags (encoded in XML) to Java executions including; middleware and application services invocations. This is implemented using the Java reflection API [37] to invoke the extracted methods or operation from the XML document.

In the case of more than one matching self-repair strategies being found, the selection algorithm selects one strategy from the search space according to a predefined conflict resolution mechanism, taking into account resolution policy, First In First Out (FIFO), and utility functions.

8.2.3 System Reconfiguration

Reconfiguration of a software application is an essential phase in a self-healing process, in that, the control service after generating a repair course of action (repair strategy (see Sec. 8.2.2)), instantiates a validation process before the generated repair plan is enacted. In addition, to achieve "optimal" reconfiguration, the controller takes into account a number of considerations to avoid any further downstream inconsistencies, including; performance degradation and/or failures. For instance, this includes some calculation of the system properties (e.g. QoS performance and/or guarantees).

Similarly to [108], we defined a set of reconfiguration operators required to undertake an application software architectural transformation and encoded in XML (see Fig. 8.8 below). These include;

- connect(): used to connect the client either with the requested service or with an alternative (i.e. in the case of a failure to connect to the requested service) through a new service manager.
- getServiceManager(): used to establish communications between the client and the service manager or the service manager of the new alternate service.
- getClient(): used to establish a communication between the new service manager and the client that requested the service. For instance, in the case of failure to connect the client with the requested service manager.
- notifyClient(): used to notify the client of the occurrence of a particular event, such as notifying the client of the availability of an alternative service.

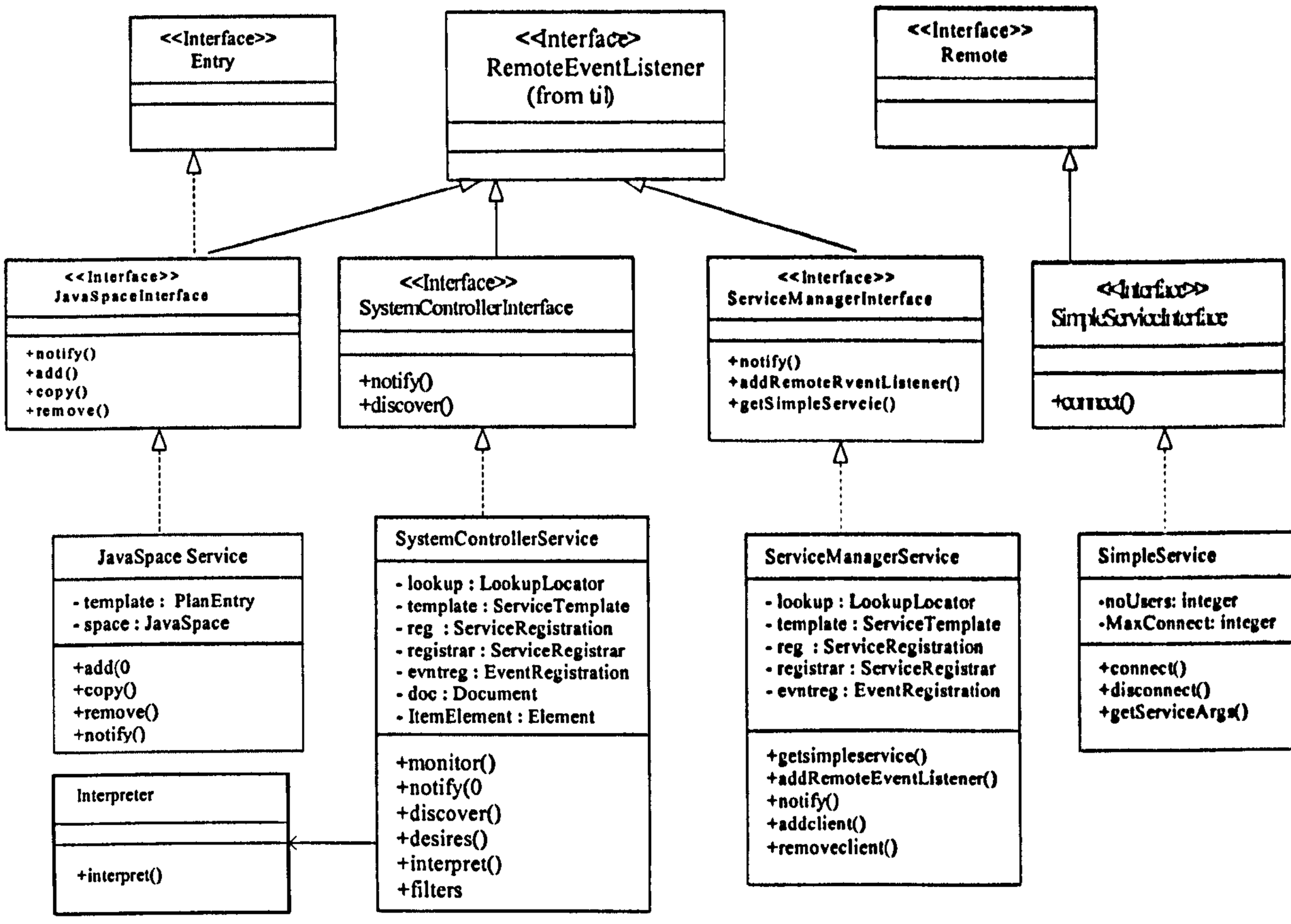


Figure 8.4: The UML class diagram of the proposed control service

8.3 Applications

In the following sections, we will describe the three applications that were implemented using Jini middleware extended with the developed, autonomic middleware control service (Chap. 7). These prototypes are used to illustrate, test and evaluate the various features of the developed middleware[12]. The applications were developed using Java JDK 1.4 and Jini 1.1 and tested on a Windows 2000 platform. Figure 8.4 above, outlines a simplified UML class diagram of the autonomic middleware control service.

8.3.1 Application 1: The GridPC Example

This application is a basic example of our initial implementation of the control service. Here the GridPC scenario is used to show the use of a shared space to coordinate and manage use of distributed resources.

The GridPC example application (or also referred to in this thesis as PCNET) provides a networked facility for requesting, accessing and upgrading a client environment with

[12] A more detailed evaluation of the work is presented in Chapter 9.

required server-side software and hardware services [136]. In this scenario, it is assumed that the clients have a minimal configuration. For instance, the clients have no local disc drives and use remote file storage. Some of them have very little memory; processors may be not more than 286s. Thus, the client automatically reconfigures its desktop, but sometimes services may be in use, offline or not currently available. The GridPC example tries to address such types of conflicts and uses the autonomic middleware control service to resolve these. In the current implementation, the GridPC application uses three main services, namely;

- A services provider, which provides services and determines the availability of a requested service, as it may already be in use by other users. To reserve/request services, a set of primitives operators have been defined and implemented, such as; add (), delete (), or list ().
- A service manager, which enables the management of an associated set of services. Also, in the case where a conflict is detected because of a client request, the control service manages the conflict resolution without returning to the manager or interrupting the system. The resource brokerage and availability is, in this example, managed and coordinated through JavaSpace.
- A JavaSpace service, which provides a distributed space or memory to enable easy access to all other distributed services or resources; a JavaSpace's entry could be added, taken, or read to/from the space using *write* (), take () and read () methods respectively. However, coordination between these services is still required and realized using the autonomic middleware control service. The "Request For Software-RFS" *entry* is an example of a JavaSpace entry. This is *extends the* abstract class called *BasicEntry* to allow the use of the previous operation.

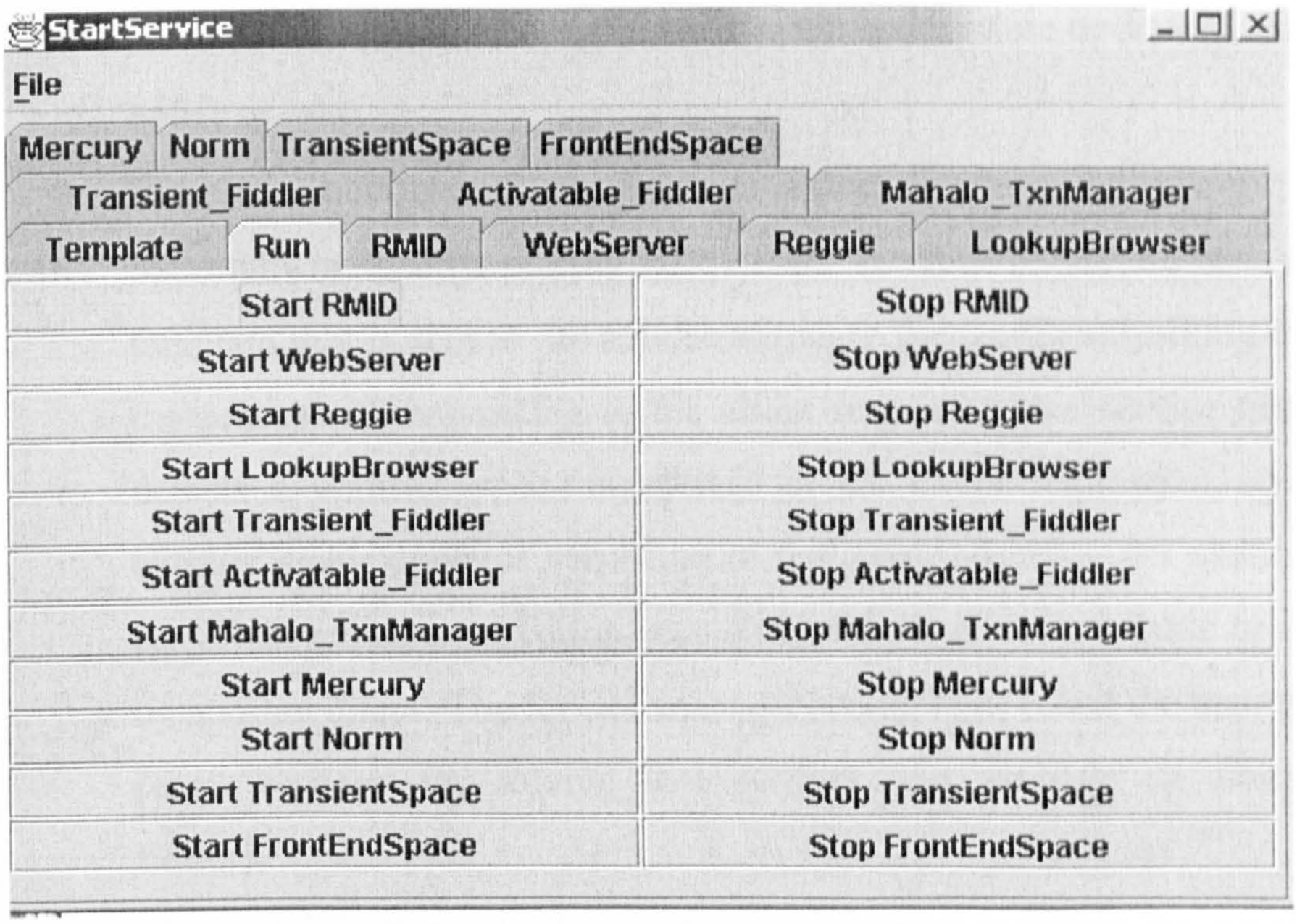


Figure 8.5: The Jini *StartService* Application.

The control service is invoked and activated in the following manner;

1. Run the Jini core middleware services including; RMID, WebServer, Reggie, FrontEndSpace services (see Fig. 8.5 below).

2. Run the client service application. This is used by the client to query or request a services/resources. The client application (see Fig. 8.6 below) has three main sections from running from top to button:

    - A text field that allows a client to specify the service name, which is used by the service manager as an entry template for reading or taking an entry from a space (i.e. a template for query the space about the client requested service). Blank fields are template wildcards; non-blank fields are values that must be matched.
    - A set of buttons for sending the client's requests to the service manager, as requested in a previous text field. This button activates the service monitor to check whether the client request result in any inconsistencies.
    - A text area for displaying the result of the service monitor process.
    - There are five buttons *Take* and *Notify* in the software panel, *enter action* in the hardware panel, *radio button* and second *Notify* in the operating system panel, which is detailled next. Each button triggers the

services self-repair process to activate the appropriate operators, such as notify or take.

- The *Take* button acts as a service repair operator for taking values optionally wildcards (empty field) from the text fields, or, to create a template that is sent to the service manager for conflict checking during the process of responding to the client request. If the service manager detects a conflict or inconsistency in the client request, it activates another repair operator according to the repair strategy. For example, if the client requests an editor (e.g. IDE), which is not available or in use by another client. The service manager would then select the appropriate repair strategy for solving that conflict (e.g. provide an alternative Editor).
- The *Notify* button, which acts as service repair operator for registering interest in event notification based on templates. Whenever the interested service's template is added to a space, a listener is notified. As zero or more clients could share the same space.

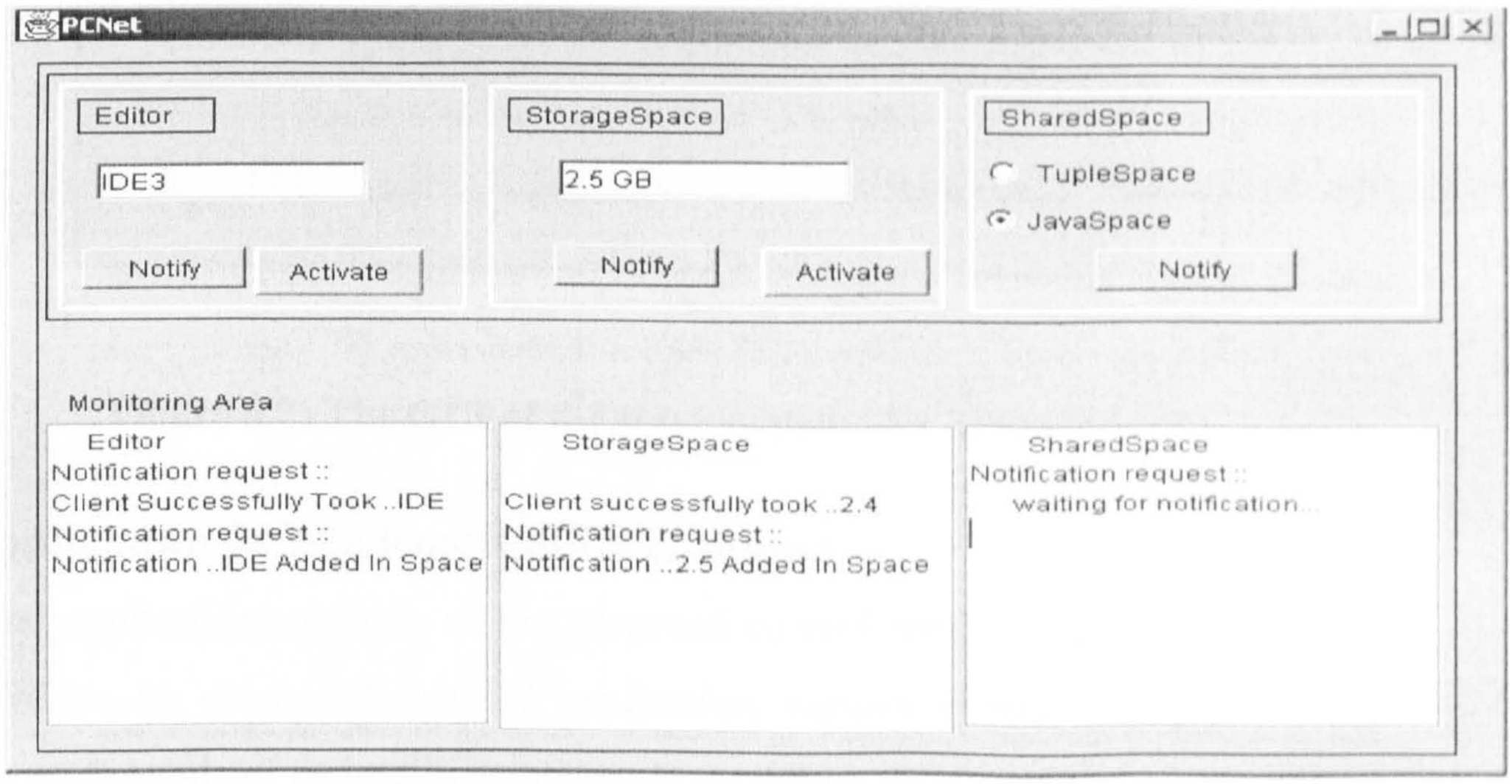


Figure 8.6: The GUI of the client service using control service.

3. The *service provider* application, which is used to add (i.e. write() entries into a JavaSpace) services to the space. This application has three main sections (see Fig. 8.7 below):
    - Three text fields are used to define an entry, which is stored in the space. For example, the three field labels are Editor, Storage, and Space, which

submit values for the RFE, RFS, and RFSS services (i.e. entries) respectively.

- The button's action activates operators such as, The *Add* button to submit (i.e. write) text field values to a space, the *read* button to read existing services from a space and the *clear* button to clear the fields. Therefore when any client registers interest in a particular service, when a service is either added or removed from the space, the button immediately notifies the interested listener.
- A text area for displaying the result.

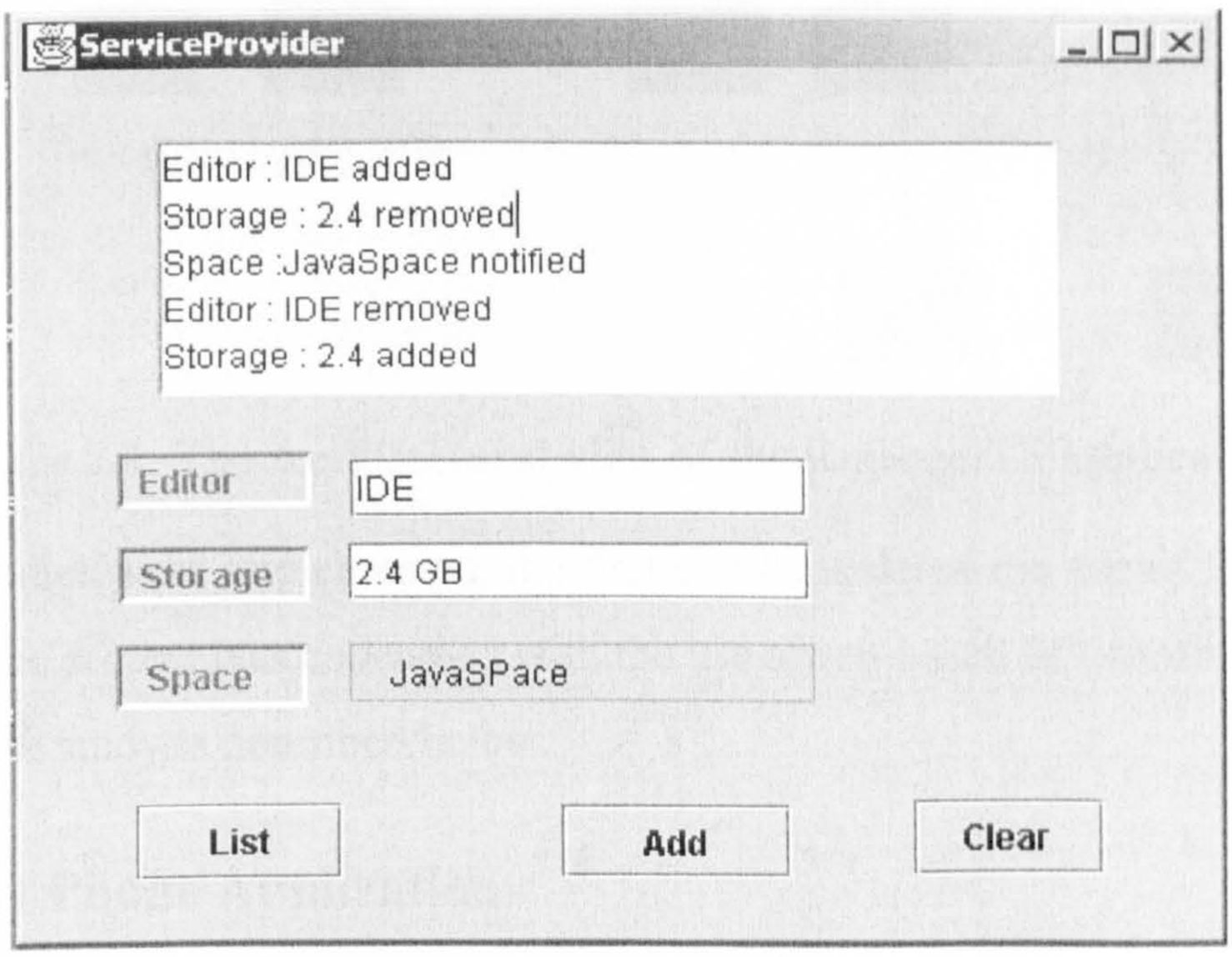


Figure 8.7: The GUI of the services provider Management.

8.3.2 Application 2: The EmergeITS Example

The second application example of the proposed control service implementation has been tested on an existing Jini-based application, namely EmergeITS, [137]. This represents an example of an intelligent networked vehicle, developed and prototyped in collaboration with the Merseyside Emergency Fire Services. Essentially, EmergeITS allows emergency fire service personnel to access a variety of distributed services, from centralized corporate systems through to remote in-vehicle computers, PDA and mobile phone devices. As shown in Figure 8.8 below, the EmergeITS architecture consists of a collection of services providing components and a Service Manager responsible for

registering application components and managing their services. Services are discovered and used accordingly by in-vehicle client computers.

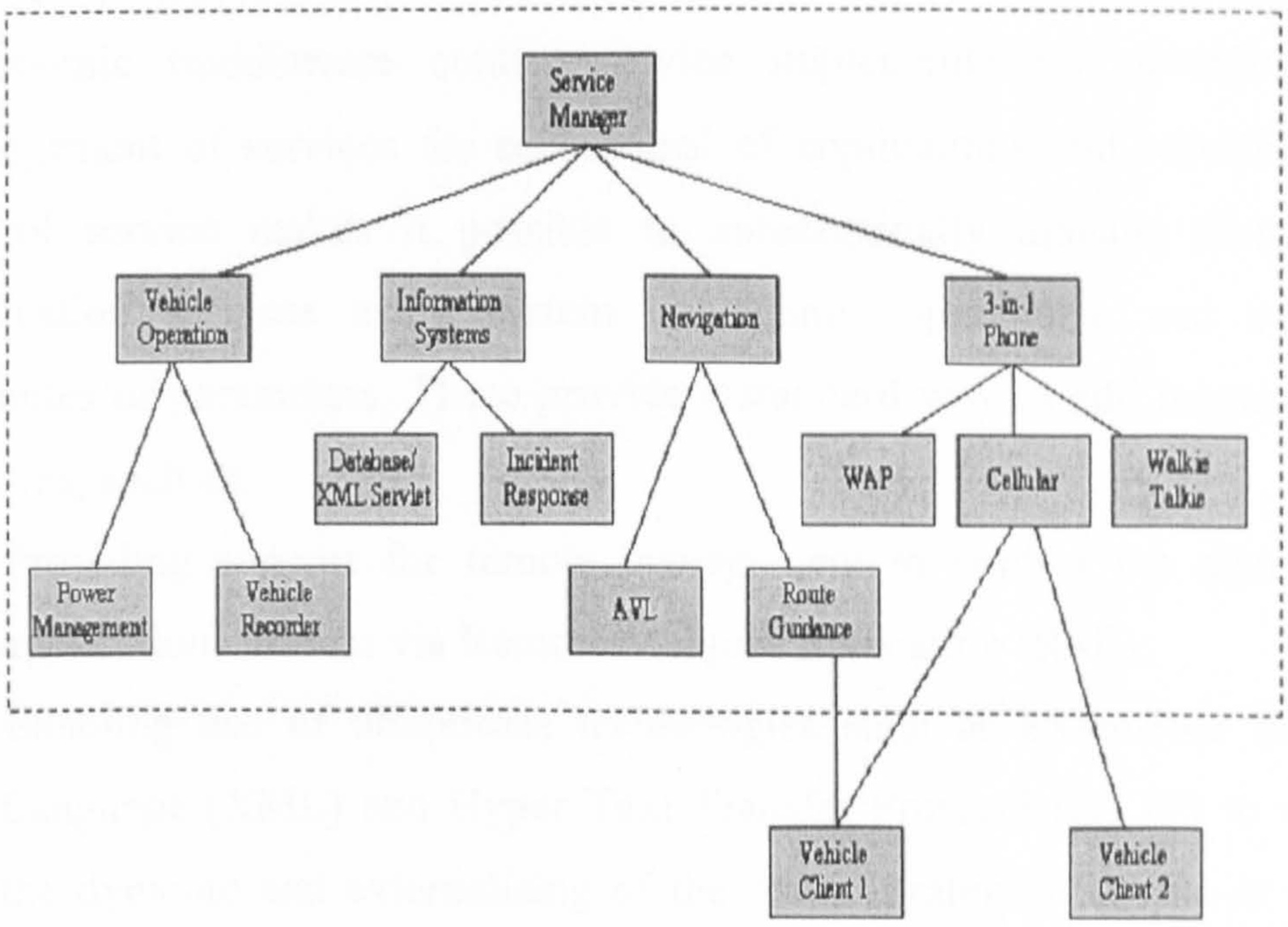


Figure 8.8: The Architectural view of the EmergeITS application.

Two case studies were implemented, the first one considered the use of the 3in1phone service and the second was concerned with the use a web-based IS (information Service (IS); each case study is described below.

8.3.2.1 3in1 Phone Application

The 3in1phone service allows a mobile phone or PDA device to be used in one of three different modes, for either voice communication or to receive multimedia content, subject to the requirements of the user and availability of a communication service provider.

The chosen 3in1phone implementation added a key feature to the first example (i.e. the first example addressed the same management and coordination aspects of the distributed services), which is the ability to create applications composed of reusable (published) components and services to support the governance of self-adaptation. This implementation is based on two key abstractions, namely;

1. Middleware core services, which describes how the proposed control service allows the components of typical platforms, available to the network, to perform as services. The core services that are provided are a) the Java environment (i.e. JDK 1.3) which can provide asynchronous direct

connectors such as RMI and b) Jini middleware services, which provides service *lookup* technology, such as *discovering a Lookup service* and the JavaSpace service.

2. Autonomic middleware control service implementation, providing the management of services for self-control of applications and services. The control service makes it possible to autonomically manage distributed application services using system constraints, operations, and services attributes or parameters. These provide a standard way to add management plug-ins, such as:

   - Providing support for remote management to control the distributed application services via Remote Methods Invocation (RMI),
   - Enabling use of ubiquitous technologies such as eXtensible Markup Language (XML) and Hyper Text Transfer Protocol (HTTP) to enable the dynamic and externalising of the repair strategies actions or plans, and to facilitate the abstraction that provides a reusable autonomic control service instead of recoding the system again.
   - Supporting the ability to register for notification of events using remote event to facilitate direct communications between the services. This event could be added, removed or changed using Remote Events Models.
   - Allowing indirect communication and invocation of the control service operations to be performed remotely at runtime on services by using the *Interface*, as remote manager interface describes the operators supported by the service manager for indirect remote management of the distributed application services.
   - Throwing exceptions, which is used for the safe termination of all the processes that are provided by the control service without shutting down the whole system or resulting in a termination error.
   - A sharing space specification that allows the shared services coordination beyond the space, considering the required core services to form a space. In addition, services that are sharing the space can be used anywhere in the space (i.e. the chosen space service for this implementation is the JavaSpace service).

The 3in1phone application service is hosted by an in-vehicle computer, which also acts as a gateway. But if the local service deployment should fail, then a backup service can be deployed from the control centre computer, thereby providing a degree of fault tolerance During the study, another project for monitoring called instrumentation [120, 132], and the proposed control services should be attached dynamically to the instrumentation service to monitor client requests on the 3in1 Phone service and control the use of the services.

The system monitors the method invocations made by clients, such as; connect () or disconnect (), and send () or receive () methods and informs the *control services* to activate its norms/rules and process as a consequence of any exceptional behaviour. Following such exceptional behaviour, control service rules trigger a monitor service to initiate a conflict resolution process and reconfiguration as appropriate. For example, the current state of the 3in1phone service may indicate a request to use the GSM service and the invocation of the connect() method on the GSM service, may result in a RemoteConnectionException due to the unavailability of a GSM service. This exception is then checked by a control service rule, resulting in the activation of a suitable conflict resolution strategy. The repair strategy first attempts a specified number of connections retries. If the retries are unsuccessful, the strategy then searches for an alternative GSM service provider or connects to another service such as WAP. Figure 8.9 below, shows an example of the conflict repair strategy (see Fig. 8.10 below) encoded in XML format.

```
<?xml version="1.0" ?>
- <!-- Simple Description of 3in1 phone Strategies -->
  <!DOCTYPE strategy (View Source for full doctype...)>
- <Strategies>
  - <Strategy id="1" type="desire">
    - <Action id="1" type="plan" name="Connection">
      - <Properties>
          <property id="1" name="host">cmsnbadr</property>
          <property id="2" name="Location">GPS_loc</property>
          <property id="3" name="Max_connected">maxNo</property>
          <property id="4" name="Method">connect</property>
        </Properties>
      </Action>
    </Strategy>
  - <Strategy id="2" type="intension">
    - <Action id="1" type="plan" name="Retry">
      - <Properties>
          <property id="1" name="No_trial">two</property>
          <property id="2" name="serviceStatus">not null</property>
          <property id="3" name="Method">connect</property>
        </Properties>
      </Action>
    - <Action id="2" type="plan" name="Alternative">
      - <Properties>
          <property id="1" name="host">cmpnbadr</property>
          <property id="2" name="New Manager ">ManagerProxy</property>
          <property id="3" name="Client Interface">ClientProxy</property>
          <property id="4" name="get Client ">getClient</property>
          <property id="5" name="Location">GPS_loc</property>
          <property id="6" name="Max_connected">maxNo</property>
          <property id="7" name="Method">connect</property>
        </Properties>
      </Action>
  </Strategy>
</Strategies>
```

Figure 8.9: The *XML* document used to describe the repair strategy sequences.

```
Strategies
    3in1Connection
Rule
     Service_state : s  not NULL
    Alternative_state : a  not NULL
     B: Binding := Binding.FREE
Actions
      Retry, Alternative, Connection
Strategy
{
  host_1 : h1 = localhost
  host_2 : h2 = localhost
  method m: connect
  s:= Retry(h1,m) or
  a:= Alternative(h2,m) then
  b :=Connection (a)
}
Action Retry
{
 do{
  Service_state : s = not NULL
  No_trial : n = 2
 Loop
 {
  b:= Connection(s)
  until
    (No_trial =2) or ( B: Binding :=
         Binding.Connection)
 }}
Action Alternative
 {
    Max_connected : a.Max <=MaxNo
    a.Manager mgr: not NULL
    s.client c not NULL
    do {
     b:= Connection(a)
    } }
Action Connection
{
      Max_connected : s.Max <=MaxNo
       B: Binding := Binding.FREE
      do {
      B: =s. connect()
      return b
      } }
```

Figure 8.10: The 3in1phone control repair strategy

The implementation's sequences in the 3in1phone example are explained below:

1. Run the Jini service as explained in Section 8.3.1, Figure 8.5.
2. Run the main application services, namely; ServiceManager.class (see Fig.8.11 below), SystemController.class and the JavaSpace service. The application of our 3in1phone example is implemented in Java using Swing components and can run as a stand-alone application (see Fig. 8.12 below).

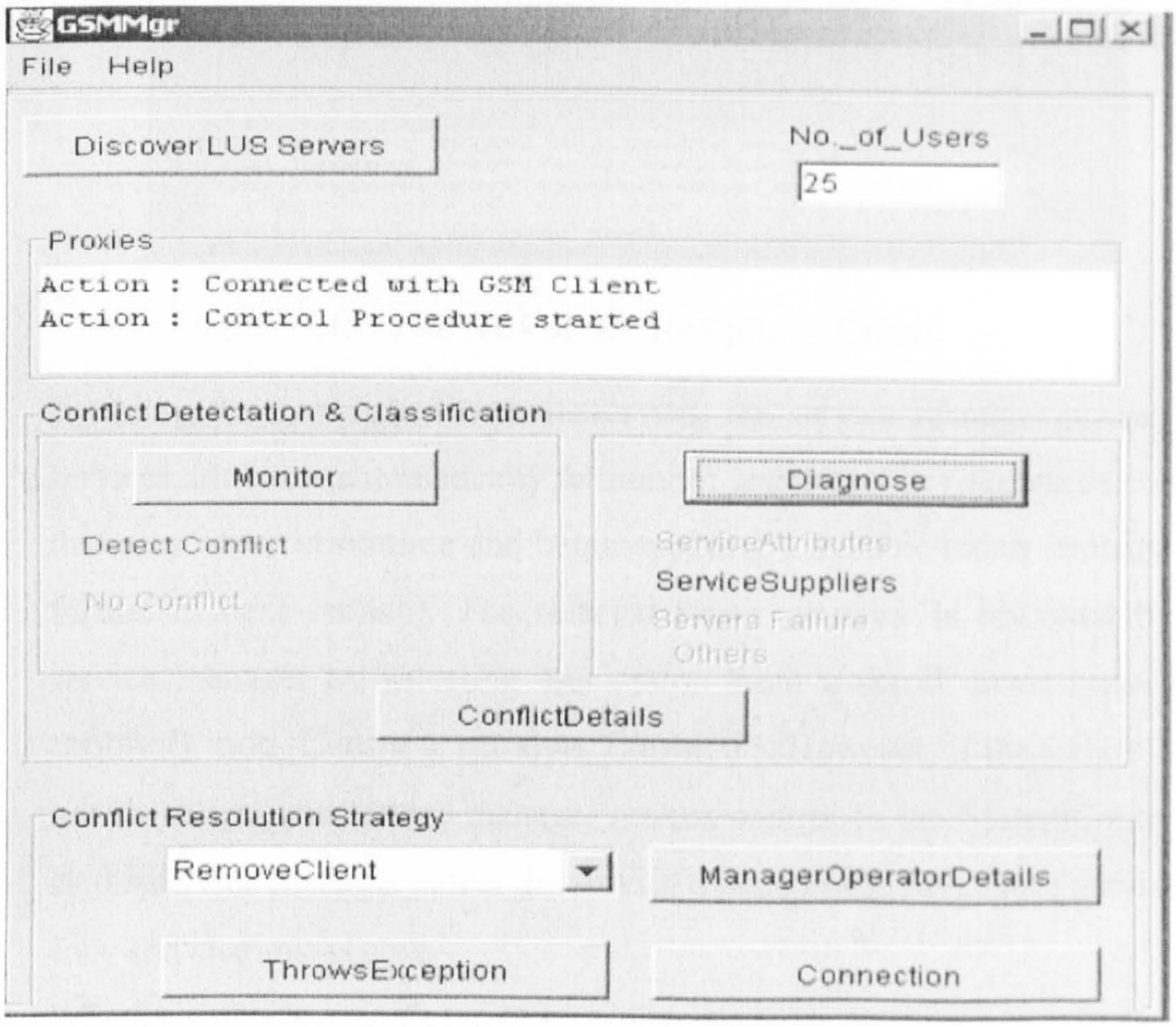


Figure 8.11: The GUI of the GSM Manager.

3. When a client requests a service, it then establishes the required communication between the services, which enables the sending and receiving of remote events and messages (i.e. more details Chap. 7) between the service and its manager before any notification or message is sent in response to the client request.

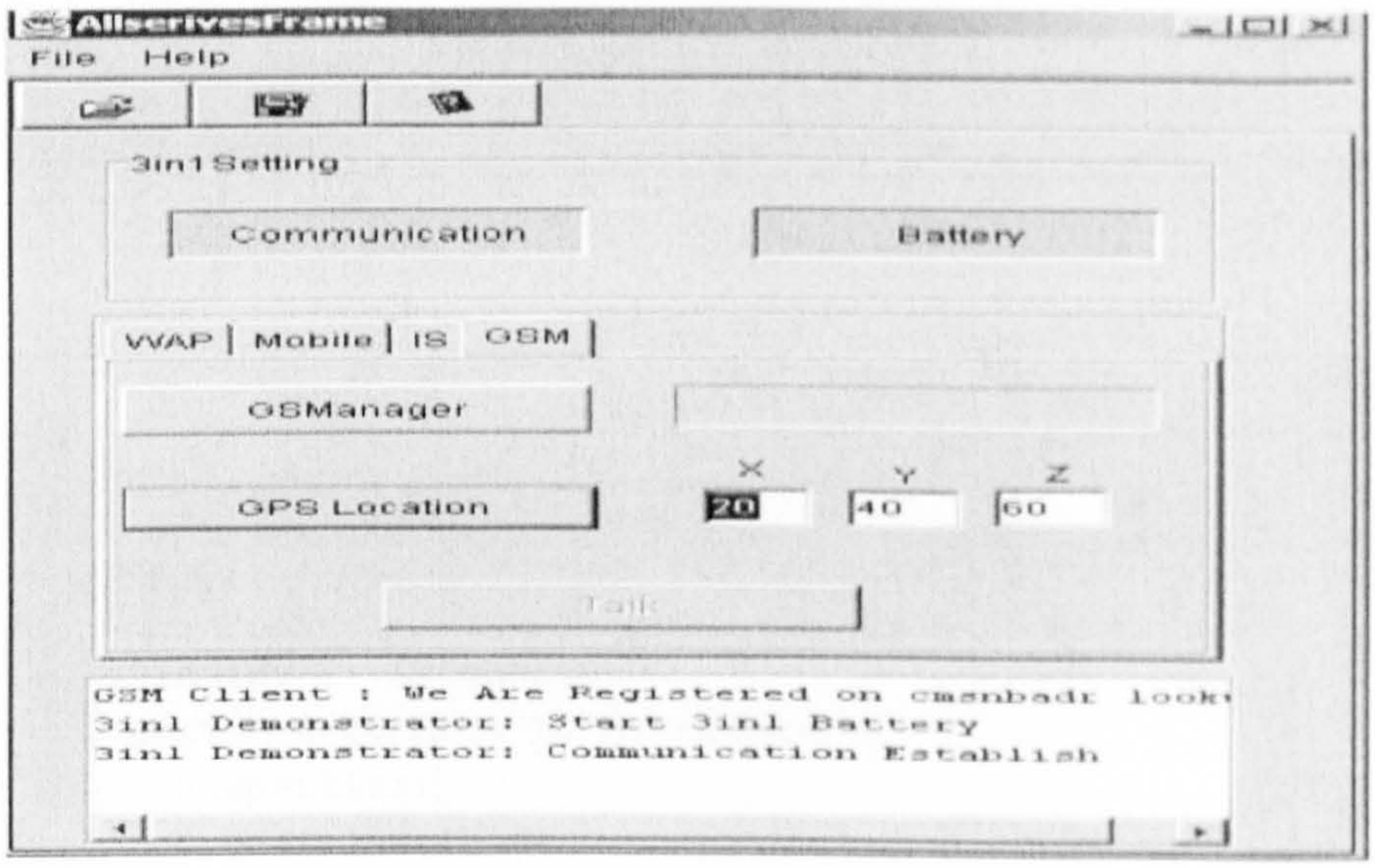


Figure 8.12: The GUI of the 3in1 phone Client.

4. By using the established communication, the service manager measures a services attributes automatically to monitor and detect any failure or conflict that may occur at runtime and begin applying a suitable repair strategy (i.e. further in next section). The selected repair operator is activated by the service manager by selecting one option from a scroll down menu (e.g. 'notify (), add_Client( ), remove_Client( )'). However, if the failure could not be solved, the service manager returns control to the SystemController, as the service manager stores the service's state in the JavaSpace service for easy and sharable access.
5. Whenever the SystemController service reads or monitor a service's state from the space and detects a failure, the system controller attempts to find an appropriate repair strategy, such as connect with the WAP as an alternative to the GSM service (Fig. 8.13).

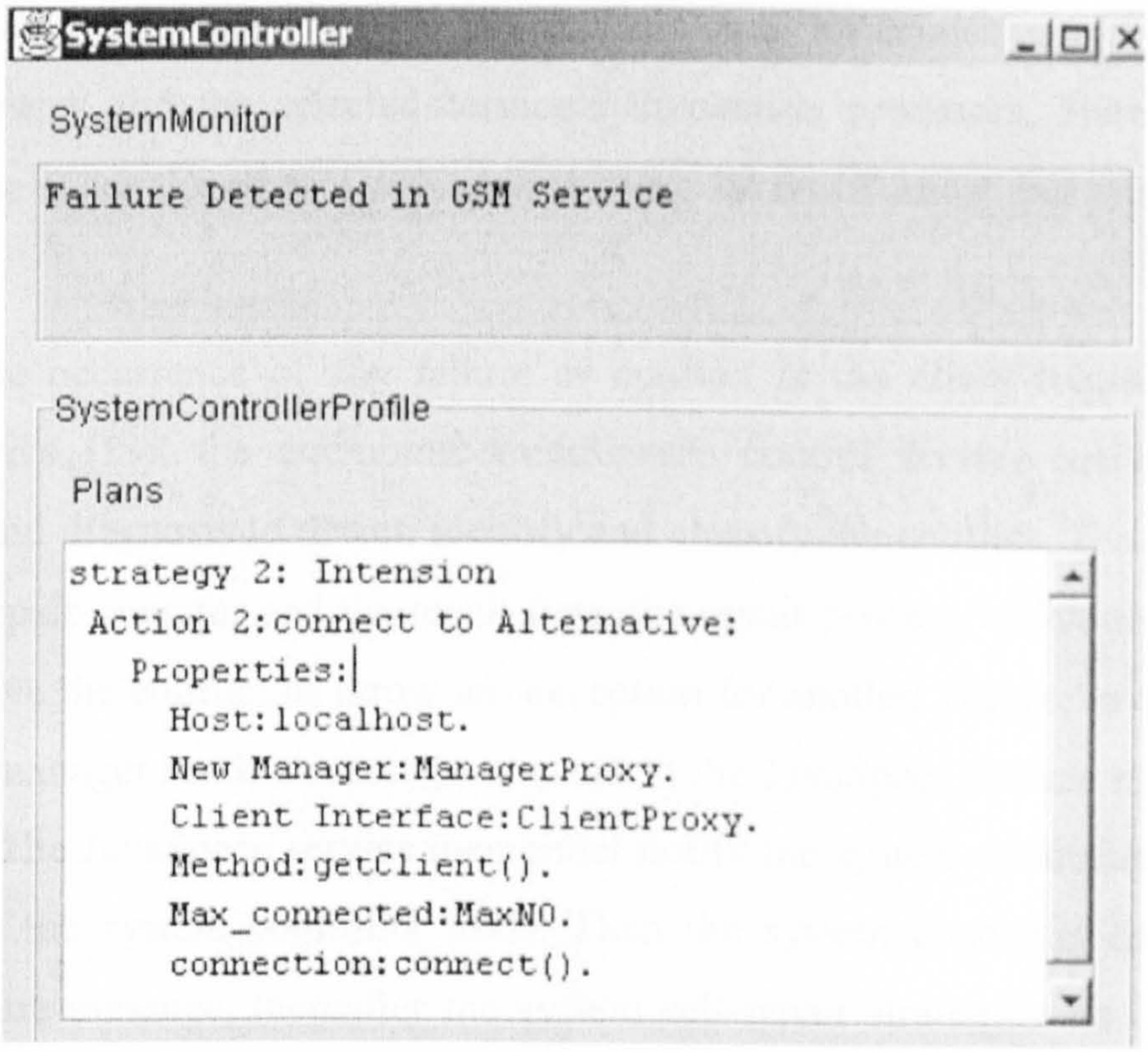
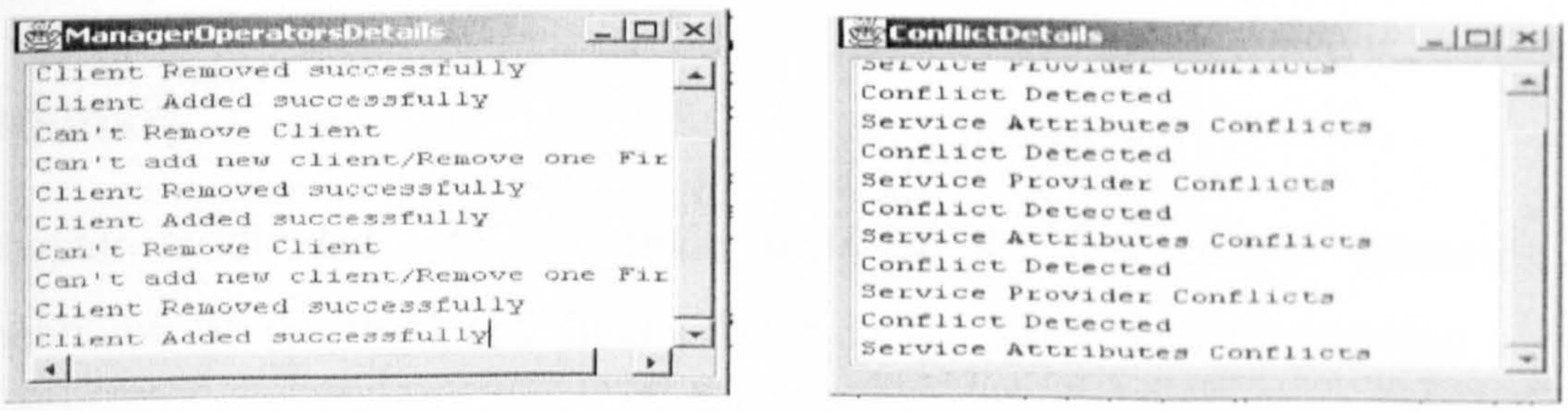


Figure 8.13: The GUI of the SystemController service.

6. Two application are provided for the ConflictDetails and ManagerOperatorDetails, shown in Figures 8.14a, 8.14b, respectively.



(a) ManagerOperatorDetails GUI (b) ConflictDetails GUI

Figure 8.14: The Conflict Description GUI.

8.3.2.2 Web-Based Information Service

The Web-based Information Service (IS) application is used to access the mobilisation information. Similarlly to the 3in1phone service, the IS service is hosted by the in-vehicle computer.

This study was performed to demonstrate how to use software instrumentation to monitor the IS. When a failure is detecting the "heartbeat" signals are not recived, the service manager will notify the controller of the event, initiating a self-repair cycle. In

this implementation, the repair strategy requests a "hot-swap" to an alternate IS node by initiating a discovery and the selected service's invocation processes. Here, the IS application service is developed as a web service using Jakarta Tomcat and provided as a Jini Service

In the case of the occurrence of any failure or conflict in the client request to the Information Service (IS), the autonomic middleware control service activates the service monitor and diagnosis to detect, identify and classify the conflict. This triggers the service self-repair operator and the result from the repair process which may either successfully resolve the conflict or throw an exception for another failure. In the latter case, the service manager sends a message or event to the JavaSpace service to register the service state. The JavaSpace service then either notify the system controller service or directly notify the system controller itself. Then the system controller begins by detecting the failure message, thereafter the system self-repair strategies are activated and start selecting and firing the appropriate strategies to reconfigure the system according to the new changes. For example, connecting the client with an alternate IS service provided by another host and notifying the client of such changes, establishing the associated changes required and feed back to the system monitor through the reconfiguration process.

8.4 Summary

This chapter is divided into two main sections; the first section discusses the implementation of the third and final service in the autonomic middleware control service, which is an essential service for the system self-control process considering the coordination aspect. The second section confirmed our implementation explanation by using the GridPC, 3in1phone, and finally the IS applications.

The system architecture of the current implementation is integrated with a Jini software architecture and based on a Java environment. We explained how the proposed approach is used to automatically manage and coordinate distributed application services. The autonomic middleware control service providing self-detection and self-diagnosis was implemented with interfaces to provide access to their structural and behavioural properties thereby establishing self-management and self-reconfiguration in the implementations of the three application examples. Overall, the chapter provides a practical and effective solution that can be used, in conjunction with any core

middleware services or web-servers to facilitate the runtime management and adaptation of distributed systems applications.

Chapter 9

Evaluation

9.1 Introduction

This chapter presents an evaluation of the developed autonomic control middleware service. This has been designed primarily to provide support for distributed applications' lifetime management and self-governance throughout a required on-demand runtime change or adaptation.

Evidently, the evaluation of such a model and associated middleware service is a difficult task, because there is no straightforward way of evaluating a self-managing and self-adapting software infrastructure, nor are there any clear metrics or accepted benchmarks.

9.2 Methodology

Consequently, this evaluation has been designed to demonstrate the use, and effect of the controller on the overall system's behaviour when facilitating the lifetime management of a given distributed application from both qualitative and quantitative perspectives. In other words, we analysed the effect of the control services on the software, and the runtime system overhead incurred by the control service in terms of processing time.

9.2.1 Objectives

For the purpose of this evaluation, we have developed two case studies namely; sorting algorithms and the 3in1 phone. The latter has been detailed in Chapter 8. In each case study, we compare the software system with and without the autonomic middleware control service. For example, we use elapsed time to undertake a sorting process as a quantitative metric to indicate the applications performance profile with and without the

autonomic control middleware service. Finally, we describe the evaluation from a qualitative perspective too.

In addition, we apply a set of evaluation metrics often used for control systems to provide a general guide for evaluating our autonomic middleware control service including;

- Stability: This is one of the most important metrics of control systems. The system is said to be stable if its responsiveness to its control rules is in a desirable interval. In our study, the system is stable if its controlled variables are within an allowable range of values and response time [138].
- Robustness: This is a metric on the controller itself. For control systems, it may not be sufficient to be nominally stable. They have to remain stable even if the process is different from the intended process model yet remaining stable.
- Time Performance Profile: This measures the amount of time it takes for a system to achieve the whole process, and where the value of the control variable is within the desirable values.
- Average Latency: This measures the average time required for the controller to begin its control cycle.

9.2.2 Approach

Although, this evaluation is not intended to be a formal performance evaluation of our developed autonomic controller service, here we will use elapsed time as an indicator to be used by the middleware controller to ensure safe and/or efficient operating conditions of a given user application. This may trigger an application change process leading for instance to self-tuning or self-healing of the considered application. In both of the case studies a range of preliminary experiments have been conducted including;

- Running a number of trials to measure and determine the efficient operating range, tolerance and control rules, for instance, applicable to the sorting algorithms.
- Defining an upper and lower performance limits for a given application, which will provide conditional triggers and control rules, for instance, to guide the controller to swap sorting algorithms to maintain a specified overall system performance – this is a kind of self-tuning.

- Measure the controller latency time, which is used here as a nominal time tolerance measure. Further details will be given in each case study.

9.2.3 Overall settings

As detailed in Chapters 7 and 8, our applications has been implemented using and extending Jini middleware, such that the autonomic control middleware services provide support for some self-management by detecting conflict and/or required behavioural changes, and establishing appropriate remedial actions say to resolve a detected conflict and/or deviation from a service normal operational model. In particular, Jini middleware is here extended by the developed autonomic control middleware service to provide a runtime control mechanism between the application service layer and the middleware service layer *with* minimum intervention from the users and *without* system disturbance (see Fig. 9.1 below). A detailed description of the design and implementation of the autonomic middleware service can be found in Chapters 6, 7 and 8.

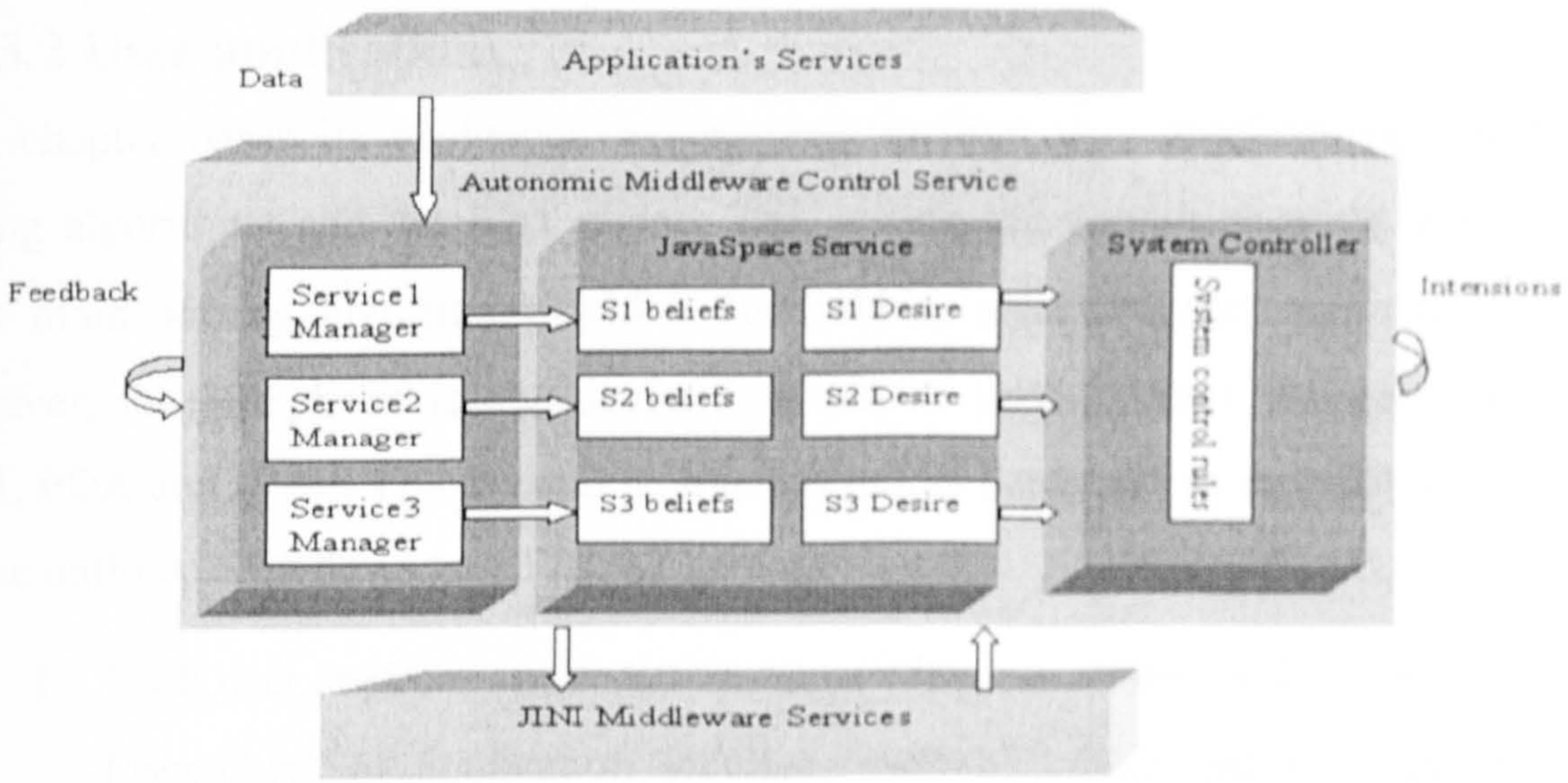


Figure 9.1: The architecture of the autonomic middleware control service.

9.2.3.1 Evaluation Requirement

In order to start the evaluation it is necessary for each case study to define the control requirements that will be dynamically required for the autonomic control middleware service (Java classes). Such control requirements encompasses a range of applications and/or domain knowledge including;

- Control rules with a boundary range to *examine runtime conflicts* and to check whether the system is stable or not.

- Resolution strategies whose main operation is to *achieve system stability*.
- Utility function to underpin the system decision-making processes when evaluating different adaptation plans. For instance to select a repair strategy (repair plans) from a large search space of possible repair plans.
- Beliefs Desires and Intentions knowledge source, which is an associative knowledge between sets of systems' or environment beliefs states and desires, which if in conflict/discrepancy, can trigger a set of associated intentions (actions).

After initialising and starting the Jini core services, the autonomic middleware control service is activated and published when the client requests a service from its service manager. The service manager begins the service management processes and notifies the system controller and sends a waiting message to the other dependent service managers to wait untill the system controller completes the self-repair and self configuration processes.

9.2.3.2 User applications

This chapter bases its evaluation experiments on two user applications, which are sorting algorithms and the 3in1 phone. The sorting algorithms example is based on three main sorting algorithms, which are bubble, selection and quick algorithms. However, the 3in1phone is one device that can be used in three different modes -- GSM, PDA and WAP. The use and interaction model between the application' services can be outlined as follows (see Fig. 9.1 above):

1. Each user application (and/or instance of) is associated with to an federation (assembly) of application services, each of which has a unique service manager, which monitor and manages its application's specified normal behavioural model (Sec. 7.3). This is achieved through the use of externalised control strategies (Sec. 7.3.5).
2. The application service sends the client's request to its service manager either by RMI or Java remote event.
3. The service manager adds its service state including service beliefs to an allocated JavaSpace (Sec. 7.4).

4. The system desires are added to the JavaSpace. The system desires are stored in an external repository as XML documents described in a proposed EBDI markup language (Chap.8).

5. At a specified frequency the system controller reads the JavaSpace to monitor its service status and/or detect any conflict or failure.

6. The system controller compares the associated system's beliefs (i.e. step 2) against the system's desires (i.e. step 3) generating if required the system *intensions* using its control rules.

9.2.3.3 Environment

The evaluation is performed using an X86 Authentic ~1.5 GH processor with 261 MB of memory, running MS Windows 2000 and connected via Ethernet 802.3. The applications and the autonomic middleware control services were implemented using the Java programming language (JDK 1.4) and Jini 1.1 middleware.

9.3 The Quantitative Evaluation

The sequence of actions carried out for both of the following case studies are as follows:

- For each service there is a service manager that is responsible for examining the service control rules and constraints by checking the values of its service attributes. These values arrive as input or feedback.
- So the service manager performs a control action when an inconsistency is detected, and sends to the client, a remote event notification to wait for a specified time while a solution for the conflict is identified.
- In the meantime, the service assumes both a mediator/proxy role between the client and the service manager. The service manager also acts as a proxy between the service and the system controller in the case of a conflict occurrence, as in this situation.
- JavaSpace is used as a distributed shared memory hosting/displaying messages from service managers concerning their services status. Hence, the system controller monitors the service status (beliefs) from the shared memory at a specified frequency.

- The system controller searches for an appropriate strategy (i.e. Intension) that is described using our proposed control strategy markup language encoded in XML (Sec. 8.2.2).
- The system controller interprets XML encoded intentions into Java executions, which often lead to user application changes and/or reconfiguration – runtime service discovery and binding. This is achieved by a translation of XML intentions (performatives) to class names (string), and then using the Java reflection API [139], the named class will be reflected to automatically discover the appropriate method name to be invoked through an RMI process (see Chapter 8 for more detail).
- When the system controller finds an appropriate solution, the results are re-submitted to start the feedback process that evaluates the control process. For example, if the maximum average latency for the control service process is 0.4 millisecond (i.e. approximately) and if the `average_latency > maximum_latency`, then the system will detect a conflict or throw an exception (i.e. start the control process again).
- The elapsed time for each control service process is measured and compared with the previous measurements (i.e. all results to ensure that are in the acceptable range). The metric unit used to evaluate autonomic middleware control service efficiency was time in milliseconds. A test run finishes when the system's rules are triggered either successfully or not and either with a conflict occurrence or not.
- Measures the elapsed time for the control process only and uses the elapsed time values as an average latency range for the control service process itself. The average latency is also used in the feedback process to monitor the control service or the system performance.

9.3.1 The Sorting Algorithm Scenario

We propose using sorting algorithms, as they are an important benchmark in scientific and commercial applications. This scenario presents a variation on traditional sorting algorithms [140] by adding an autonomic middleware control service, which enables a "kind" of autonomic self-organisation, tuning and/or self-healing of the sorting application, in response to unpredictable system behaviour.

For this experiment, three sorting algorithms are used, namely; Bubble, Selection, and Quick algorithms [141]. As mentioned earlier, our study was designed to fulfil self-adapting, self-managing or autonomic behaviour. Thus, to evaluate this goal, in the first step of this experiment a calibration exercise was conducted, in that, the control process was run for each sorting algorithm 20 times with 20 randomly generated arrays of varying sizes. The sorting time for each sort process is then recorded. In particular, for each dynamic dataset, we specified an initial array size and an increment. For example, if we want to sort arrays of sizes 1000, 1500, 2000, … , 5000, the initial size would be 1000 and the increment would be 500. For each of the 20 array sizes, the decomposed experimental steps are described below:

1. Initialise the array size (see Fig. 9.2 below) using a developed application for dynamic initialisation, which is called by the *getInitSize( )* method, as the initial array size for each sorting algorithm can be dynamically changed. Also as shown in Figure 9.3, the *getIncrementSize( )* method receives the increment variable, which defines the size of the next generated array[13].

```
public int getInitSize ( ) {
        int init= 0;
        try{
        init = Integer.parseInt (frame.jTextField2.getText( ) );
            }
        catch (Exception e)
        //throw exception
        return init;
            }
```

Figure 9.2: Initialise the array size process.

2. Generate an array of a specified size, which will be automatically generated by a specially developed random array generator (Fig. 9.3 below). The generated array will be populated by randomly ordered integers chosen from the [0 to Array(Size)] interval.
3. Create a sorting object by loading the Sorting class, which enable the controller to dynamically invoke a chosen sorting method such as; *bubbleSort( ), selectionSort(),* and/or *quickSort()*. Each of which implements Bubble, Selection,

[13] To this end a random array generator has been developed for this experiment.

selectionSort(), and/or *quickSort()*. Each of which implements Bubble, Selection, and Quick algorithms respectively. Figure 9.4 below, illustrates the processes for sorting with these algorithms.

```
public init_Rand_array( ){
 //call initial array size
       getInit ();
        // array Size Increment
       getIncrementSize ();
      for (int i = 0; i < NUM_ARRAYS; i++) {
       int testSize = initialSize + i * sizeIncrement;
       int testArr[ ] = randomArray(testSize);
     } }
```

Figure 9.3: Fill the specified size array with random integers.

```
public class Sorting {
 public void bubbleSort (int array[ ] ) {
 for (int i =0; array.length >=0; --i )
 for (int j = 0; j<i; j++) {
 if (array[j] > array[j+1] ) {
    temp = array[j];
    array[j] = array[j+1];
     array[j+1] = temp;      } }
 public void selectionSort (int array[ ] ) {
 for (int i = 0; i < array.length - 1; i++) {
 int index = i;
 for (int j = i+1; j < array.length; j++) {
 if (array[j] < array[index] )
 index = j; }
 //call the swap method
 swap(array, i, index);    }}
public void quickSort (int array[ ], int first, int last) {
//call the partition method
part = partition(array, first, last);
quickSort(array, first, part-1);
    quickSort(array, part+1, last);     } }
```

Figure 9.4: The main process for sorting using the three selected algorithms.

4. Return the elapsed time in milliseconds taken by each sorting process, which is calculated by taking the difference between the initial time[14] and the final time measure using the *System.currentTimeMillis()* method before and after the sorting process respectively. This difference uses an *integer* variables not a *long* as before (the difference will be small enough to store as an *int*). Figure 9.5 below shows an example of the elapsed time measurement for each algorithm.

```
Long startTime = System.currentTimeMillis();
    //call the chosen sorting algorithm,
    sorting.selectionSort();
    Long endTime = System.currentTimeMillis();
Int elapsedTime = (int) (endTime - startTime);
```

Figure 9.5: Example of calculating the elapsed time for any algorithm.

5. The elapsed time result is used to generate the boundary range and algorithm intervals. This is required to generate the *control service rules* for lifetime management of the system's behaviour. For each algorithm, the measured elapsed time can indicate the time performance profile and thus the efficiency boundary of each algorithm. So the measurement of the elapsed time is used as a measurement metric to generate the time limitation for each sorting process. As soon as the system reaches this limitation the control process detects the failure and starts the control process.
6. Although from the experimental result (i.e. step 5) we define the start point for the control process to detect failure, it is still possible for another failure to arise from the control process itself requiring another detection. For this reason we measured the latency of the control process itself for each sorting algorithm, and defined the maximum latency for the sorting algorithms (see Fig. 9.6 below). This average latency value will be used in the system feedback process to detect any further conflict that may arise during the control process time.

[14] As the number returned is relatively large a long variable was used instead of int.

```
Long startTime = System.currentTimeMillis();
        //define the appropriate control rules for detection,
                If(arraySize >= max_bubbleSize) {
                  If(no_swaps) >= max_Bubbleswaps {
         //start control
                      Class c=Class.forName(className);
                      Class partypes[]=null
                        Method meth=c.getMethod(methodname,partypes);
                        NotifyClient();
                      } }
 Long endTime = System.currentTimeMillis();

 Int latency = (int) (endTime - startTime);
```

Figure 9.6: The process of an average latency calculation for the sorting algorithms control service.

9.3.2 The Sorting Algorithm Experimental Results

The main results drawn from our experiment are used as a basis for defining the control rules for our control service as follows:

- The number of elements being sorted and the process ends with the data in sorted order.
- The sorting by any particular algorithm is fast enough and the elapsed time is reasonable.
- The average latency for the control process does not exceed the maximum latency (maximum latency comes from the experimental result).

To generate the service control rules and constraints by considering the previous measurement aspects, we analysed the effect of the changes in the number of each algorithm dataset input on the system time performance profile. In other words, we measure how many milliseconds it takes for each dataset with each sorting algorithm (i.e. our experiment uses three sorting algorithms, namely that are *bubble sort*, *selection sort* and *quick sort*). The results (shown in the tables below) demonstrate that the processing time increases and the system starts to slow down as the number of the elements in the array increases, but the limitations of such an increase are not known. So we ran each algorithm was run to find the ideal choice of

array size for each algorithm. From the test results, we discovered that when array sizes are too small no useful information is elicited (such as zero or a very small number). With too large array sizes, the algorithm took a very long to run. In considering the computer's processor speed, we should pick different sets of array sizes that work well with the processor speed and are not zero or too long as shown in the following Table 9.1:

Bubble Sort algorithm	
Array (Size)	Time (ms)
500	10
1000	10
1500	20
2000	50
2500	70
3000	100
3500	140
4000	171
4500	220
5000	281
5500	350
6000	401
6500	471
7000	550
7500	621
8000	711
8500	801
9000	902
9500	1011
10000	-

Selection sort algorithm	
Array (Size)	Time (ms)
5300	70
6600	100
7900	150
9200	201
10500	260
11800	330
13100	420
14400	501
15700	601
17000	697
18300	811
19600	932
20900	1051
22200	1192
23500	1332
24800	1482
26100	1612
27400	1793
28700	1973
30000	-

Quick sort algorithm	
Array (Size)	Time (ms)
30000	10
60000	20
90000	30
120000	60
150000	70
180000	90
210000	100
240000	115
270000	131
300000	150
330000	160
360000	181
390000	201
420000	220
450000	238
480000	256
510000	269
540000	280
570000	320
600000	339

Table 9.1: Examples of the elapsed time for different sorting algorithms.

For each algorithm dataset, we represented a linear scale for the time performance profile as a function of the array size (where the x-axis is the array size and the y-axis is the time in milliseconds) as shown in Figures 9.7-9.9 below:

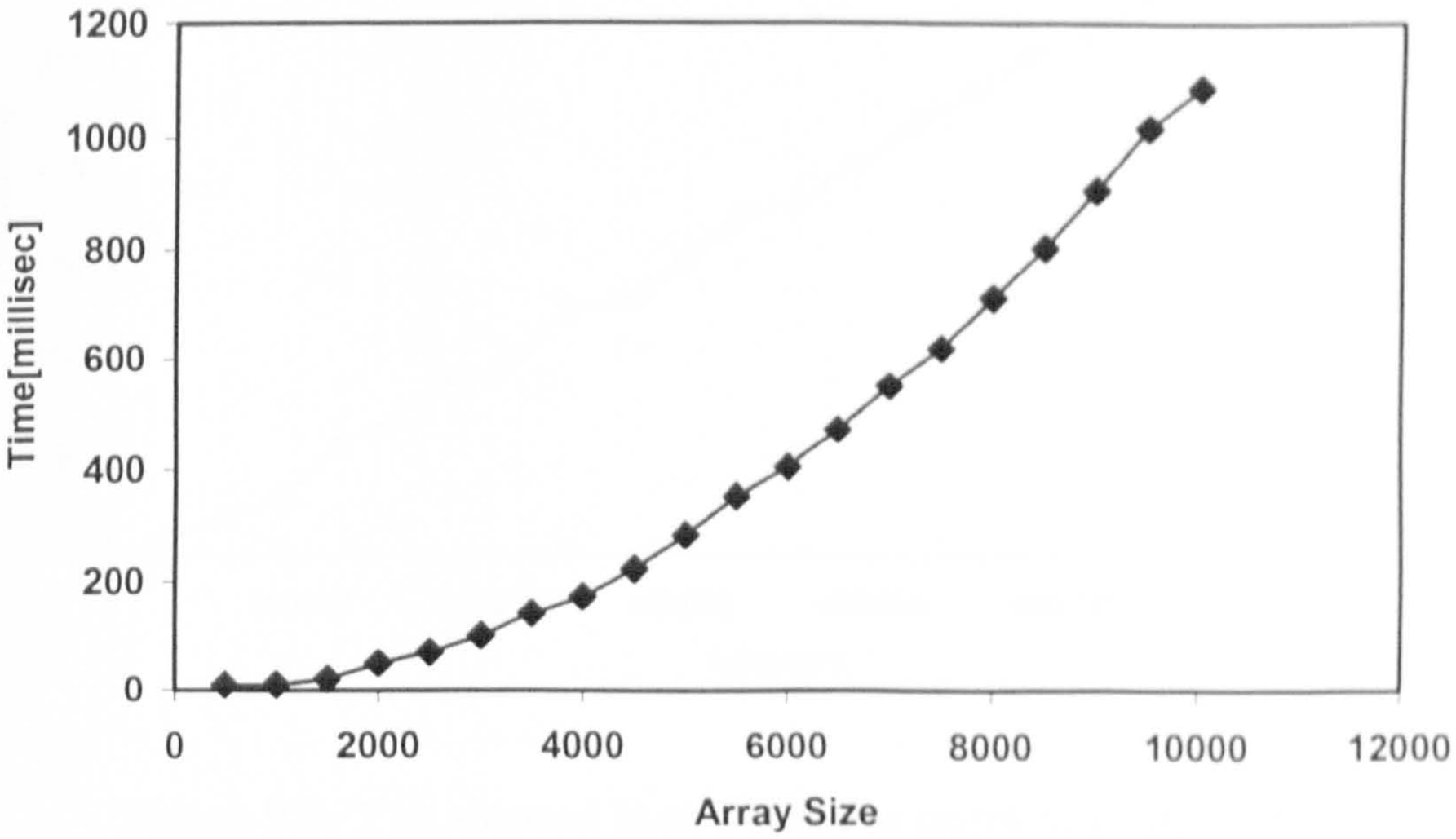


Figure 9.7: The elapsed time using the bubble sort algorithm.

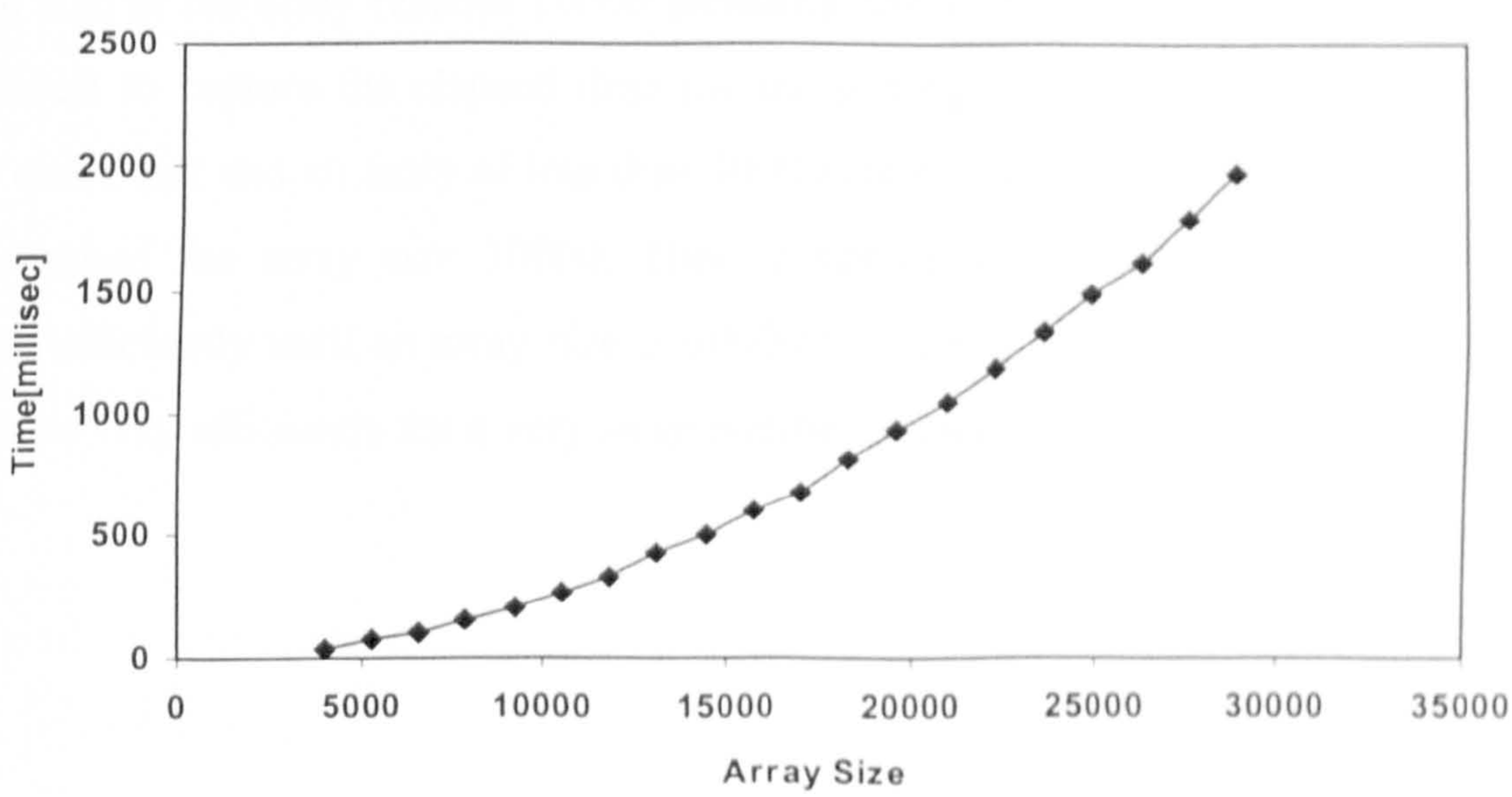


Figure 9.8: The elapsed time using the selection algorithm.

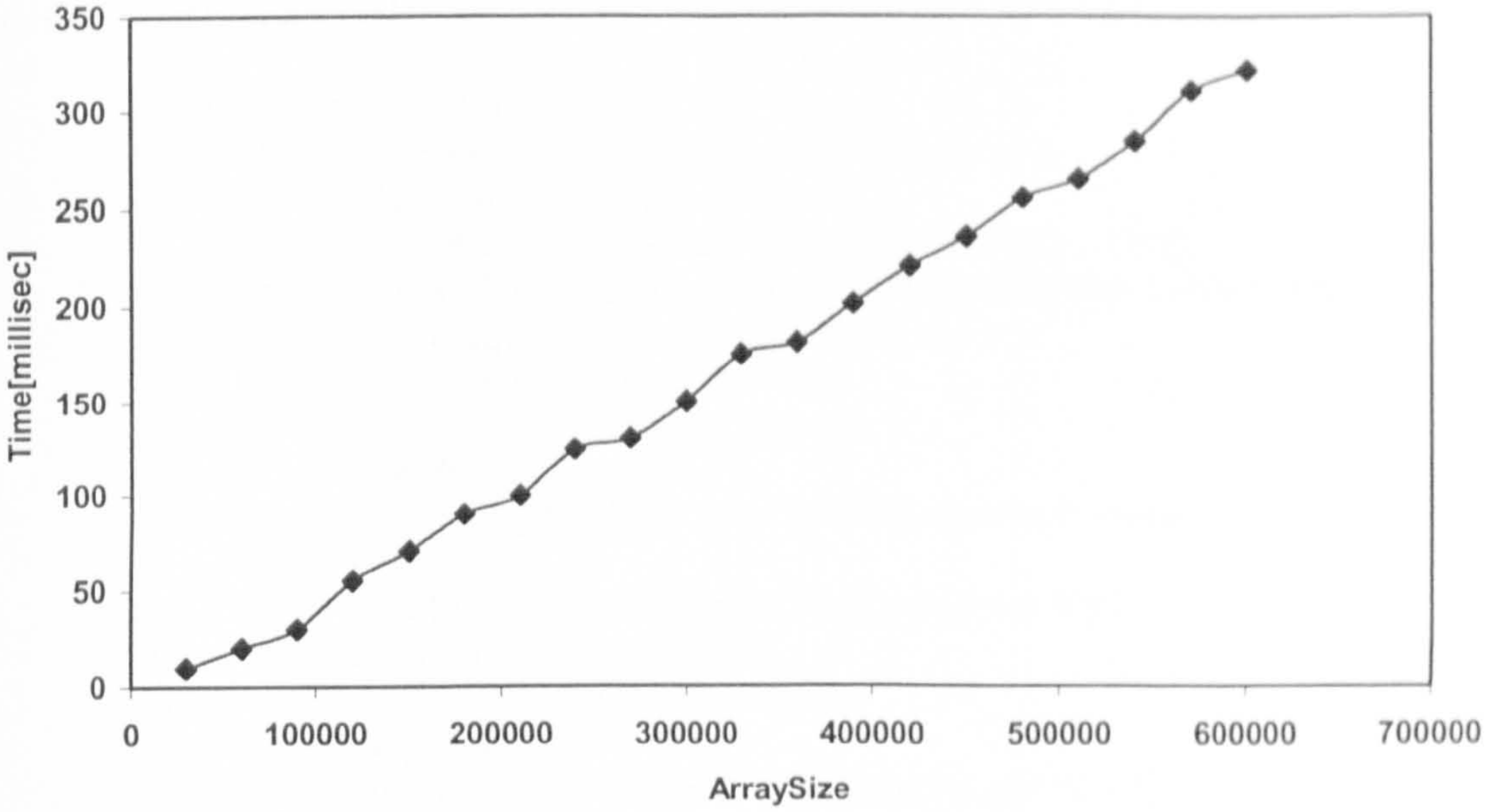


Figure 9.9: The elapsed time using the quick sort algorithm.

In the test, it was noticeable that the measured time increases as the size of the array increases for each separate algorithm. For example, with the bubble sort algorithm when the size of the array reaches 10000 elements, the system slows down and it was very difficult to capture the elapsed time for the sorting process. On the other hand, with the quick sort and an array of less than 30000 elements, the elapsed time was zero until it reached the array size 30000. Then it started at 10 milliseconds and kept measuring efficiently until an array size is 600000 elements was reached, so the quick sort operates very efficiently for a very large number of elements.

```
public void getbeliefsVar(){
        try{
            DocumentBuilderFactory
            factory=DocumentBuilderFactory.newInstance( );
            factory.setValidating(true);
            DocumentBuilder builder=factory.newDocumentBuilder();
            Document doc=builder.parse("http://localhost:8080/beliefsVar.xml");
            listofItem=doc.getElementsByTagName("var1");
                    if (listofItem.getLength()!=0){
                      Node Item=listofItem.item(0);
                    Intended_Attribute =
                          Item.getChildNodes().item(0).getNodeValue();
                    }
                     listofItem=doc.getElementsByTagName("var2");
                 if (listofItem.getLength()!=0){
                      Node Item=listofItem.item(0);
                   Utility_Attribute =
                      Item.getChildNodes().item(0).getNodeValue();
                   }}
          catch (java.rmi.RemoteException  exception){
                //thrown exception
            }
      }
```

Figure 9.10: Extracting the utility and intended attribute variables from the XML file.

As mentioned before, the experimental results provide the guidelines for generating the system boundaries for the control service rules to achieve system stability and robustness by using time measurement. Two main variables are used here to generate the control rules for each algorithm. These are generic variables in our code, but are specified using the XML file, the system then uses the Java reflection API [139] to get the class and method name (i.e. which are specified in the XML file), and hence invokes the method name (i.e. more detail about that implementation in Chap. 8). The two variables extracted from the XML file for the sorting algorithm example are *array_size* for testing system stability (e.g. an intended attribute) and *no_swaps* (e.g. utility attribute) for examining system robustness. The size of each array is used to determine the approximate range for each algorithm and the *no_swaps* variable is the number of swaps that have been performed in the sorting process. Initially, the controller examines the array size, if it within the algorithm range, then *no_swaps* is examined. If this is larger than the algorithm number of swaps limit, it assigns the appropriate algorithm instead. This directs the system to choose the appropriate algorithm. Here are examples of the linguistic values for the *array_size* or (intended attributes) variable for selection sort. These are *within_selectionRange*, and

out_selectionRange values. The values for the *no_swaps* variable are *above_average* and *below_average*. Examples of control rules are shown in Figures 9.10 and 9.11.

```
boolean Belief_Constraints = false;
 String Intended_Attribute= null;
 String Utility_Attribute= null;

public boolean check_beliefs ( ){
If   (Intended_Attribute <= within_selectionRange &&
            Utility_Attribute <= below_average)
 {
    Return Belief_Constraints = true;
 }
Else If (Intended_Attribute >=within_selectionRange &&
          no_swaps >= above_average)
 {
    Return Belief_Constraints = false;
 }
 public void filter(){
  boolean flag = Beliefs();
 if (flag != false){
  try{
    desires ();
    //invocation of the desire method, e.g. selectionSort(testArray);
  }catch(java.rmi.RemoteException remotexception){
    //thrown remote exception
 }}
  else{
  try{
   intentions ();
   //invocation of the intension method, e.g. quickSort(testArray);
   }catch(java.rmi.RemoteException remotexcept){
 //thrown remote exception
 } }
  public void execute( )throws RemoteException
  {
   getbeliefsVar()
   filter();
  }
```

Figure 9.11: Design of the Control rules corresponding to array_size and no_swaps.

Array (Size)	Time (ms)	Selected sort algorithm
500	10	Bubble
7500	550	Bubble
14500	521	Selection
21500	10	Selection
28500	20	Quick
35500	25	Quick
42500	33	Quick
49500	40	Quick
56500	47	Quick
63500	58	Quick

Table 9.2: The autonomic middleware control service manages different array sizes using the autonomic control service.

When the value of the array size is *within_Range*, the system will use the selection sort to achieve system stability, but in the meantime the controller detects that the value of the *no_swap* variable is *above_average*. This means the number of swaps performed by the selection sort with the previous array was above average and the system is not able to achieve robustness. Hence, the system controller will self-control the system again and assigns this process to the quick sort instead to achieve robustness.

We examined the control rules of the control service on 10 different datasets each of a different array size (Table. 9.2 above) and allowed the autonomic middleware control service to force the system to the most appropriate solution. Without the autonomic control service the system would not be able to sort the arrays of a higher size in particular or may crash.

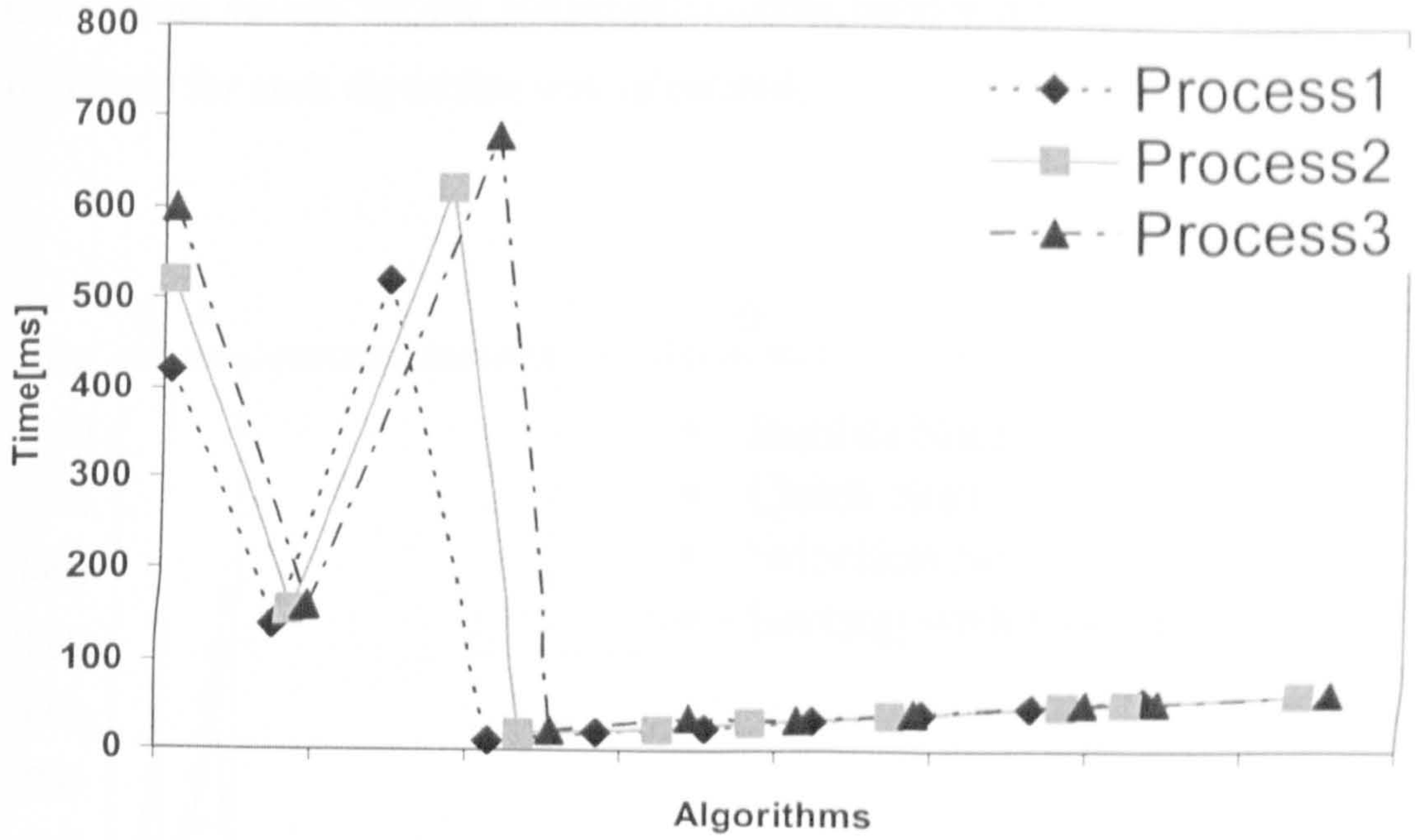


Figure 9.12: The chart indicates the sorting of different arrays with the autonomic control service.

As shown in Figure 9.12 above, we found that if we ran the system with the autonomic middleware control service more than once, it was clear that the service controls its behaviour according to the run-time environment, as the autonomic control service detected and managed system run-time inconsistency. For example, in Figure 9.12 above, when the system received an array of size 14500 elements, the system controller assigned this array to the selection sort algorithm instead of the bubble sort algorithm. Since the bubble sort is not suitable for that size of the array as defined by its time performance profile.

It is very clear here that the system using the autonomic control service is *stable* because it is responsiveness to the control service is still in desirable boundaries and its control variables are within a range of desirable values, in addition the control service is provide the system with the continuous monitor and repair capabilities, which is main the system remain stable or in other mean robust.

services *turned off*. The same experiments were input to the non-control service system with the same values for the autonomic control service *turned on* and the number of milliseconds for each algorithm was calculated.

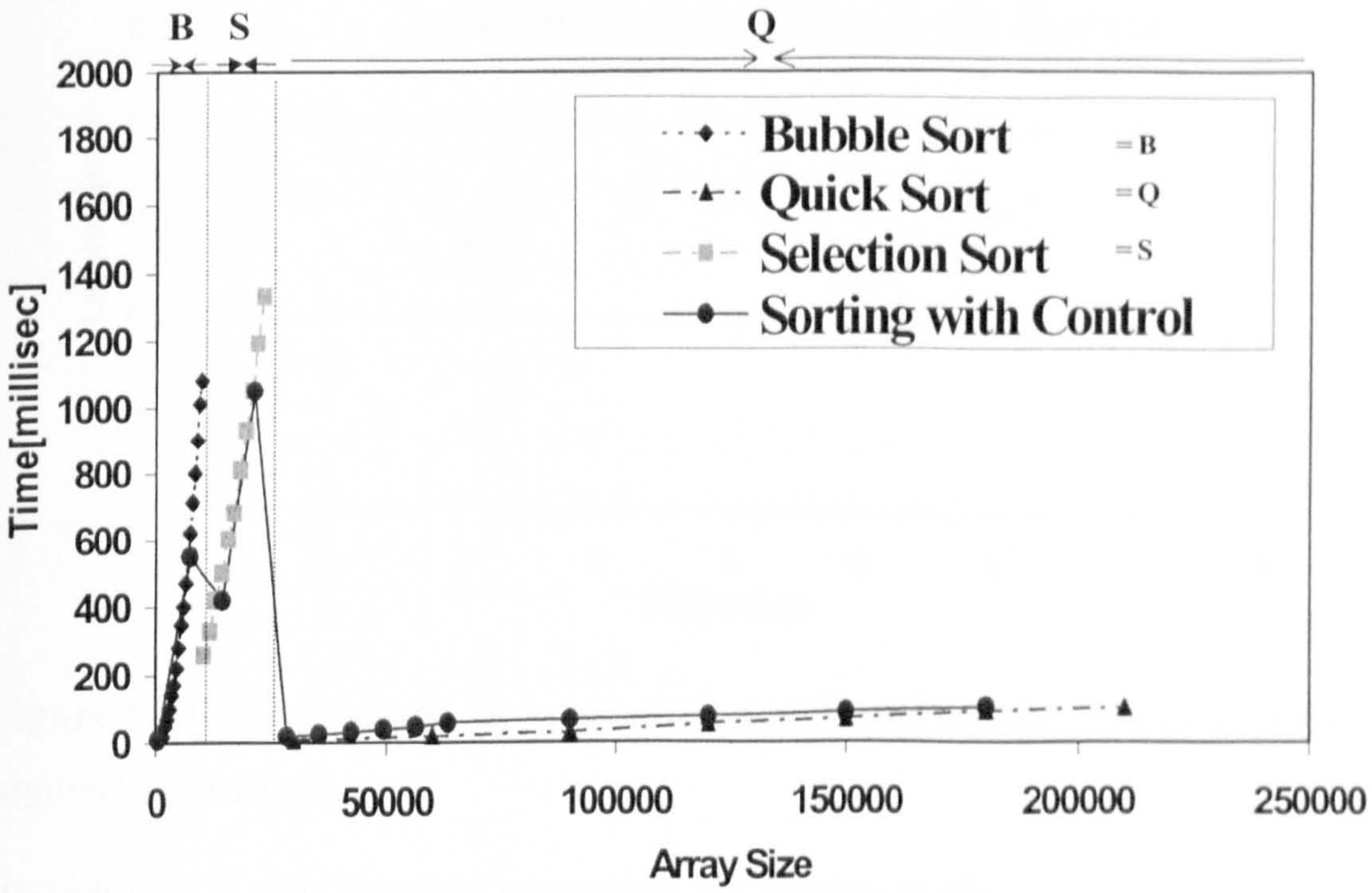


Figure 9.13: Comparison of the time performance profile of sorting algorithms with control service and without control service

In Figure 9.13 above, the elapsed time for sorting each dataset with the appropriate algorithm with and without the control service is presented. The horizontal axis corresponds to the range of the array size suitable for each algorithm. To make the comparison clear, we present two cases:

1. One case without the control service that is represented in separate lines for each individual algorithm, where the dashed line represents the selection sort algorithm, the dotted line represents the bubble sort algorithm and the segmented line represents the quick sort algorithm. As there is no control service in this situation, the system performance either slows down with the increasing array size or it takes a very long time for each process and could not continue after a certain point.
2. On the other hand, in the case of the control service existence, represented by the solid line, the system continues processing with no disturbance even if the

array size becomes too large for one algorithm, by adapting system behaviour at run time at the point the degradation of the time performance began.

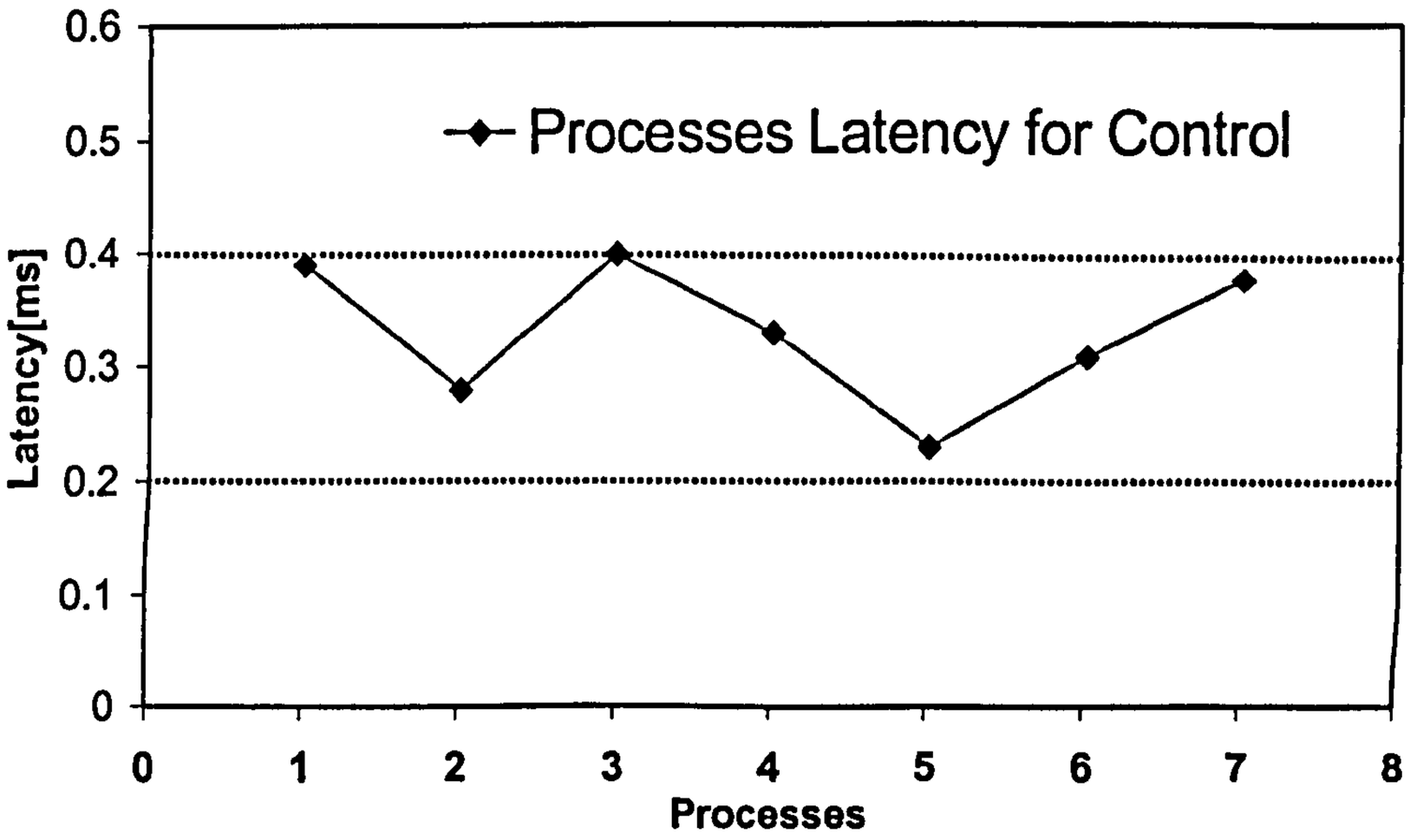


Figure 9.14: The average latency while running the control process of the sorting algorithm example.

In addition, in continuously measuring the system performance to prevent any further conflict arising during the control service process itself, we measured the average latency for every control action in each sorting process (Fig. 9.6). According to the measured average latency, a latency interval is defined (e.g. maximum latency). Once the control service takes more than the allowable maximum latency (i.e. the average latency is input to the control service using the feedback process), then a conflict arises and the control service begins the control process again to trigger the appropriate solution strategy.

In Figure 9.14 above, we show the approximate average latency of the system with the control service for the sorting algorithm example. We experimented on eight array sizes and found a maximum average latency of 0.4 (ms). If the maximum latency is more than 0.4 (ms) a conflict or inconsistency is detected and the control process finds an alternative sorting algorithm (i.e. resolution strategy).

The values of the previous experimented average latency is used here to provide the system with the maximum latency values which is used for monitoring and evaluating the autonomic control service itself, also it provided the system with the runtime continuous monitoring capabilities (i.e. feedback process).

9.3.3 The 3in1 phone Scenario

As mentioned previously in Chapter 8, a 3in1 phone allows a mobile phone or palm device to be used in three different modes, either for voice communication or to receive multimedia content, subject to the requirements of the user and the availability of a communication service provider. This section evaluates the 3in1 phone example that is detailed in Chapter 8, and evaluated the system efficiency and time performance profile using the same methodology as that used for the sorting algorithm study (Sec. 9.2). The system's performance was also compared, with and without the autonomic middleware control service (i.e. or simply control service) of the system.

The main advantage of the autonomic middleware control service is a resolution of detected conflict at runtime without the shutdown of the whole system (i.e. as emphasised by the sorting algorithm case study). Also the control service provides a dynamic runtime self-control that allows the control service to work efficiently for the lifetime management of any system that requires runtime self-control with minimum intervention from the system developers (i.e. as will be demonstrated by the 3in1 phone case study). The specific service's attributes and methods should be in the XML document and the control service reflects those values to invoke the specific services functionality in the control process.

The 3in1 phone case study is used to emphasis the previously identified advantages of our control service in the following system situation:

- Without conflict occurrence and without control services
- With conflict occurrence and without control services
- Without conflict occurrence and with control services
- With conflict occurrence and with control services

In this evaluation, the architecture is distributed over three machines, in that, one is used for the 3in1 phone client to request services, the second is for the service manager to look after its service and the third is responsible for controlling the whole system. Through JavaSpace the service managers add their service states and the period of time granted. This information being used by the controller to monitor, control, coordinate and regulate the application's behaviour.

The two main steps of this experimental scenario are explained briefly below (anddetailed further in Chapter 8):

- Calling ApplicationService.class starts the 3in1 phone application and then the *ServiceManager.class* is automatically instantiated as a manager of the requested service.
- The system controller begins by calling *SystemController.class* and continuously monitors the JavaSpace to detect any runtime conflict in the application services.

The experimental service attribute values we proposed and used to detect the failure/conflict in this evaluation are as follows:

- The service GPS location as a first attribute, we proposed a valid value and it is in an acceptable boundary range.
- The number of connected clients as a second attribute. This is received as a remote event notification from the service provider to the service manager. We assume that the value of this attribute does not satisfy its allowable boundary and a conflict is raised.
- The experiment is implemented twice, once with the control service (see Fig. 9.15 below) and once without (see Fig. 9.16 below).

```
public service_with_control () {
 try{
   if(GPSLocation <= GSM_Range
        &&no_Clients <=max_clients)

         GSM.connect();
 else{
        Class c= Class.forName("ApplicationService");
        Class partypes[] = null;
        Method meth= c.getmethod("WAP_connect",partypes);
       notify (event);
}catch (java.rmi.RemotException e)
      {throw Exception}
}
```

Figure 9.15: An example of the system without the control service and with and without conflicts.

```
public service_with_control () {
        try{
           if(GPSLocation <= GSM_Range
                 &&no_Clients <=max_clients)

                  GSM.connect();
            else{
                  System.exit(0);
         }catch (java.rmi.RemotException e)
          {throw Exception}
        }
```

Figure 9.16: An example of the system with the control service and with and without conflicts

9.3.4 The 3in1 Phone Experimental Results

To evaluate the autonomic control service of the distributed self-adaptive software, we quantitatively evaluate the time elapsed as an example measurement of the control mechanisms efficiency in response to a run time system conflict. In order to achieve our aim, we compared the time overhead of the software system with the autonomic middleware control service against the elapsed time of a software system without the autonomic middleware control service (i.e. the same methodology as in the sorting algorithm case study). This means we have performed the comparison on the same set of experimental values with the control service turned on once and off once. Additionally, the comparison is performed with and without conflict occurrence.

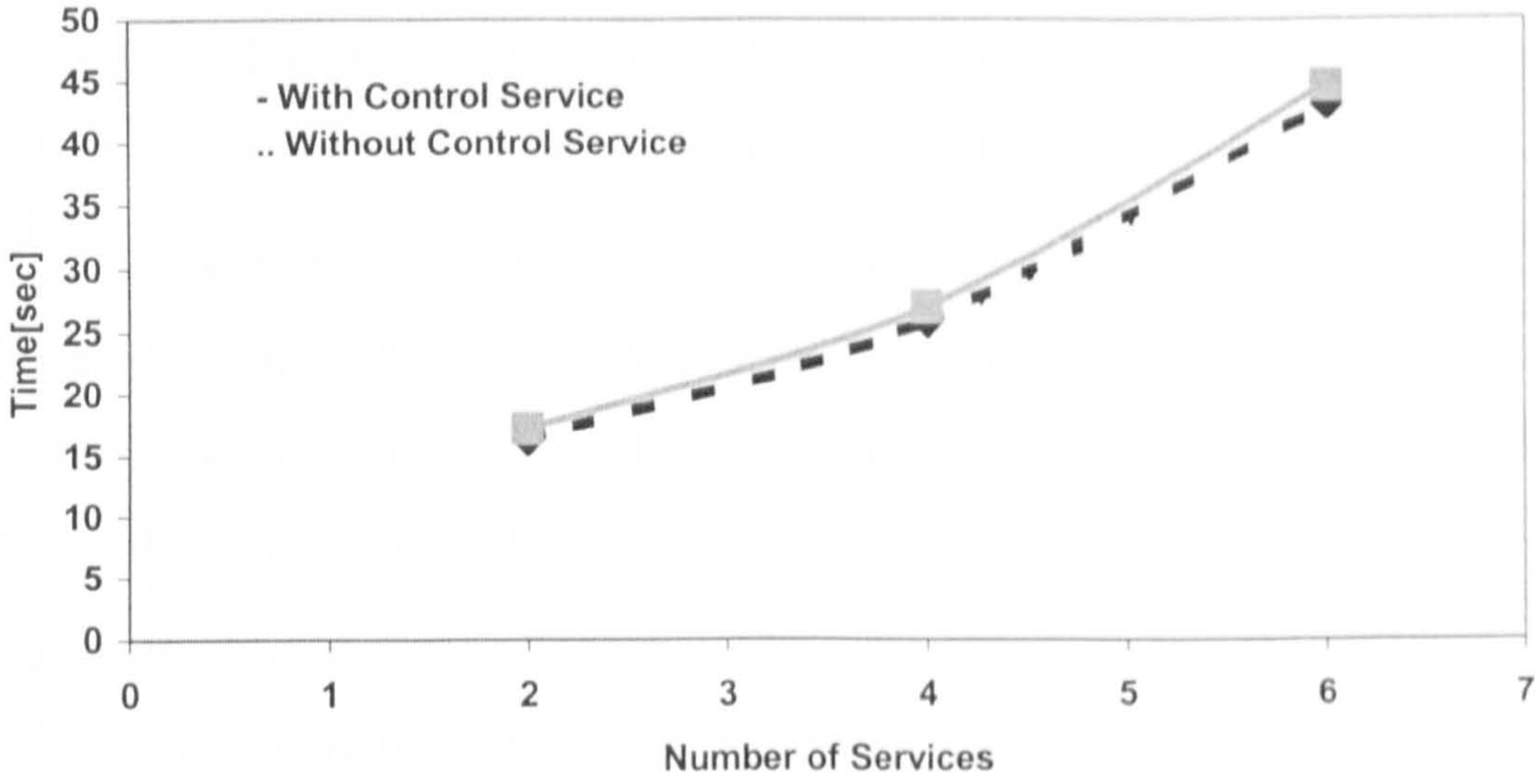


Figure 9.17: Comparison of the elapsed time with **and** without **the autonomic middleware control service without conflict occurrence.**

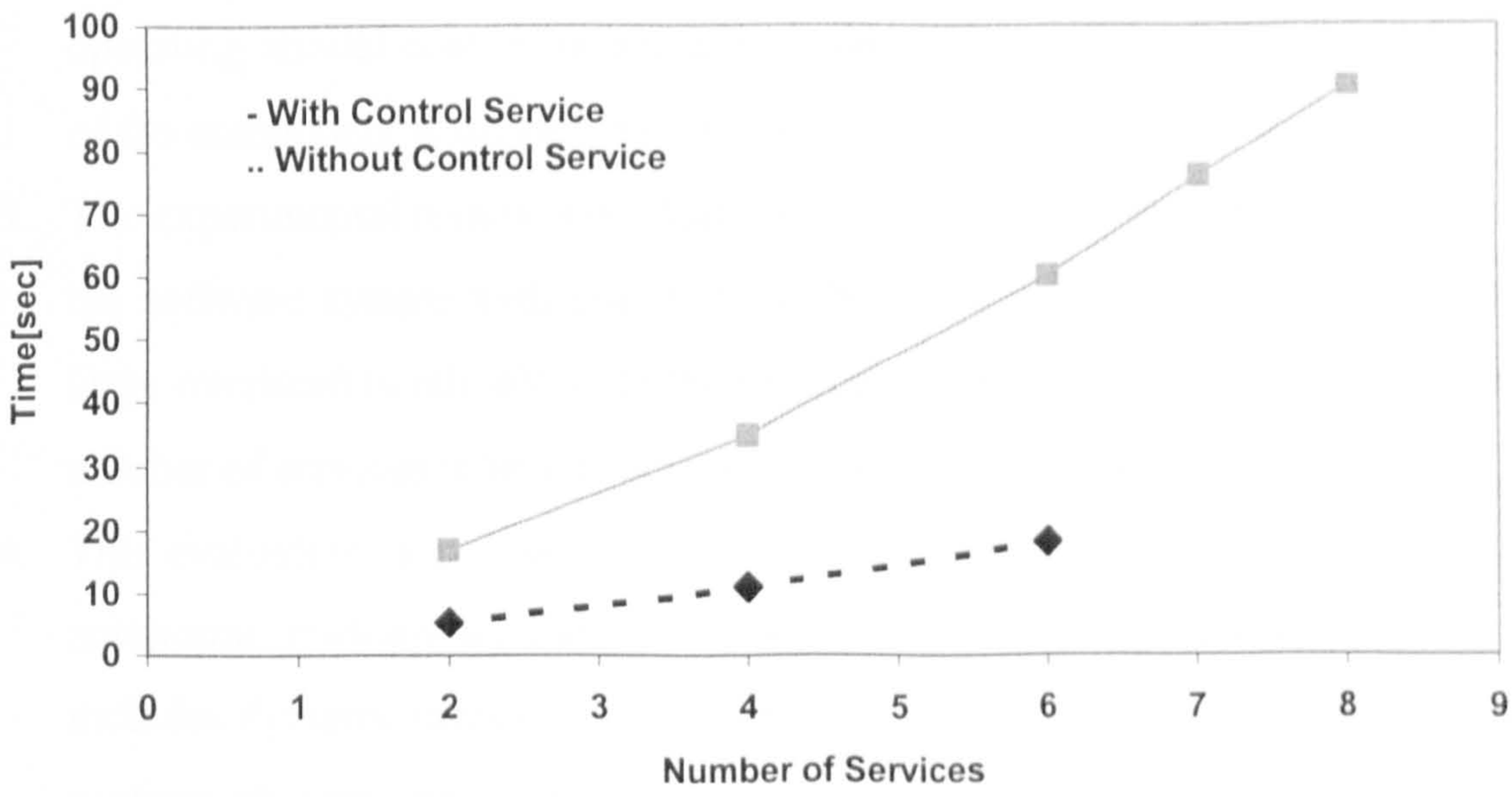


Figure 9.18: Comparison of the elapsed time with **and** without **the autonomic middleware control service and** with **conflict occurrence.**

The experimental are result shown in Figure 9.17 above, where the x-axis is the number of services and the y-axis is the elapsed time in seconds. The solid line represents the system with the autonomic middleware control service while the dotted line represents the system without the autonomic middleware control service. The graph shows one curve close to the other because both represent the same number of services with the same service attribute values, but without any conflict occurrence.

Therefore the system in the case of no conflict it still stable and as well robust either with the control service processes or without the control service processes, because it achieves its goals and desires with no evolvement of conflicts.

As shown in Figure 9.18 above, where the x-axis is the number of services and the y-axis is the elapsed time in seconds, the solid line represents the system with the autonomic middleware control service. The dotted lined represents the system without the autonomic middleware control service, the results are explained as follows:

1. The results show that the system with the control service is still stable within it s desirable intervals even if it consumes more time in a conflict occurrence situation. However, it is possible to manage the actual waiting time by adding checking the *average latency* of autonomic middleware control services, but it is difficult to eliminate the waiting time without the control mechanism
2. The cost performance should be relatively high in the case of failure modes, since the annual cost of the software application is the summation of its operating annual cost in its operating model. In addition, the annual initial cost of the component is divided by the active run time in years [142].
3. The experimental results show that the difference between the calculated time of the software system with and without the control service is still reasonable. A large overhead is not added to the running system, as the time increases, as the number of services is increased with and without the control service.
4. This evaluation is also an initial step in the correct way to identify appropriate autonomic middleware control services for distributed software systems. This includes dynamic issues such as; generic control rules, dynamic measurement of runtime changes and the use of generic standard based Jini middleware in a distributed system.
5. The comparison between the system with and without the autonomic middleware control service shows the system's continuous processing within its desirable boundaries even in the event of conflict or inconsistency by continuous monitoring and managing itself at runtime. Thus, ensuring self-stability and robustness at all time. Without such a control or supervisory mechanism, in the event of malfunction the system can crash and requires shutdown to undertake necessary maintenance work and restart.

Conceptually, to measure the effectiveness of our approach, we examined the average latency (i.e. the elapsed time it takes for the control process to operate) of the system with the control service, and how much time the system is delayed because of the control service use.

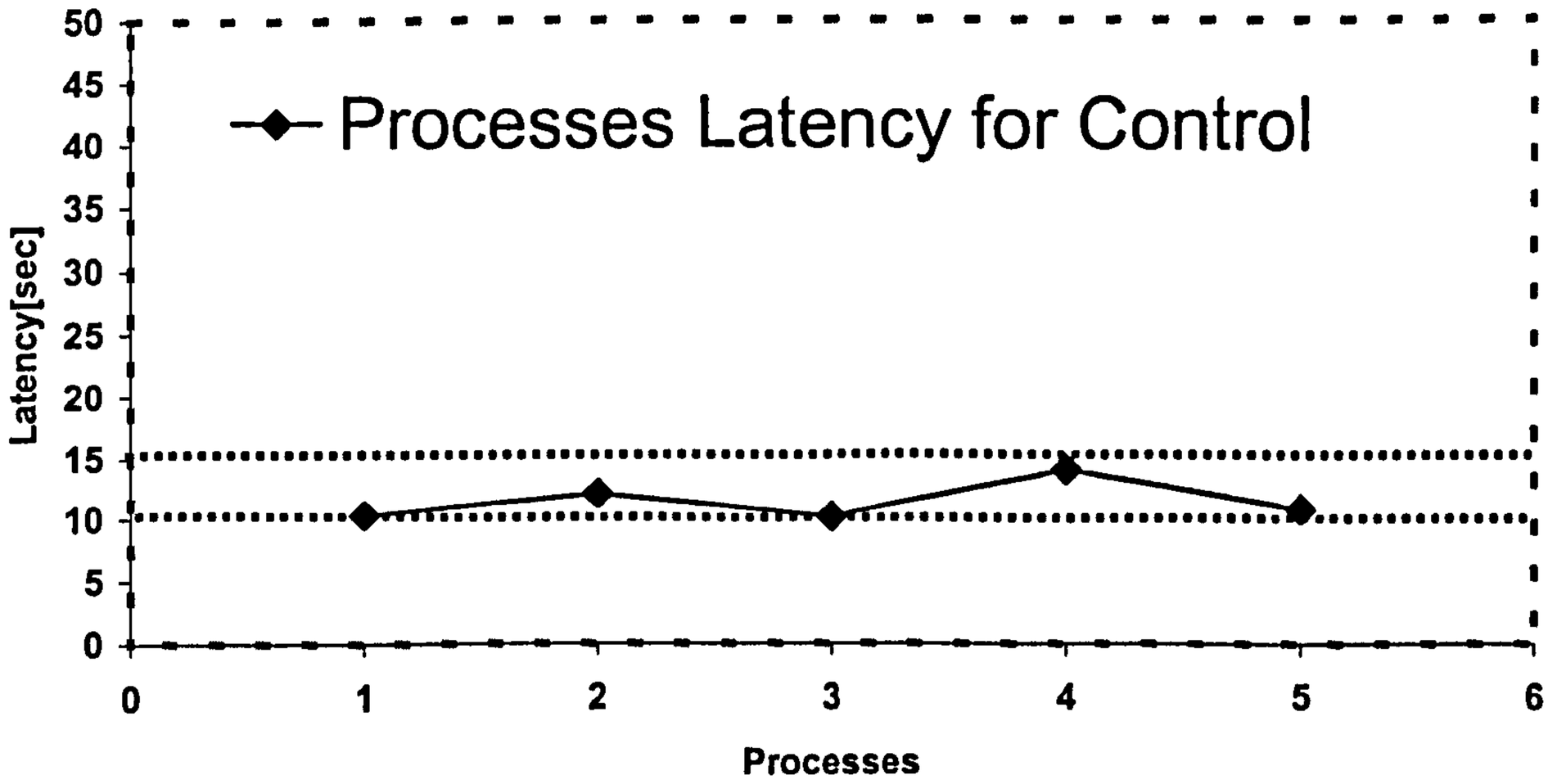


Figure 9.19: The average latency while running the control process for the 3in1 phone example.

Figure 9.19 above shows the average latency for the control process in the 3in1 phone example. We experimented five times and found the average latency to be between 10 seconds and 15 seconds. As explained in the previous example, this average latency is used as feedback to monitor the control service itself. We assume that the maximum average latency experienced is not more than 15 seconds otherwise another conflict or inconsistency is detected during the control process.

9.4 Qualitative Evaluation

This section describes a general software evaluation of the autonomic middleware control service, which was conducted to test and evaluate the general functional specifications of the prototyped autonomic middleware control services and the associated programming and control model.

To this end, a number of scenarios were examined, including; GridPC, sorting algorithms, EmergeITS, 3in1phone and a web-based information service to assess the middleware using a set of qualitative metrics, such as;

- Functionality, by enabling the distributed application service with the required functionality to self-detect, self-repair and self reconfigure itself at run time
- Generality, by designing a general structure capability that is not bounded or limited to any specific system or case study without essential changes to the code, such as using XML format.
- Flexibility, by reducing most of the complexity that could be embedded to any running system for self-managing itself by using the flexible infrastructure to support a full range of service management and adaptation.
- Extensibility, by referring a control middleware design pattern for a software developer to design and manage distributed application services taking into account the uncertainty and complexity issues related to such systems.

9.5 Discussion

It is a challenge to produce conclusive evidence of the benefits, merits, effectiveness, correctness and completeness of the proposed control middleware. However, both the qualitative and quantitative evaluations indicated that the proposed autonomic middleware control service and the associated programming model seem to fulfil the defined requirements and are generic and flexible enough to support the rapid development of a range of applications. In particular, in this evaluation, the control services succeeded in self-detecting, self-repairing, and self-configuring in the case studies, dynamically and at runtime without any essential changes to the code, indicating stability and robustness of the middleware. The main points are summarised below:

- The proposed approach provides stability and robustness to the distributed system and respond to uncertainty by using system repair strategies, to select and evaluate the selected decision based on the Belief, Desires and Intention (BDI) model. In particular, the robustness of the system satisfied by supporting our extendable BDI (EBDI), which provides the system with the ability to change its selected repair strategy at runtime if it does not achieve the continuous stability in the running system. In the case of simultaneous requests,

requests are stored in a queue and the system controller takes information from the queue and starts the control process sequentially for each client.

- Also, the reasonable time performance of the running system with the control service was evaluated during the quantitative evaluation. After tem expwriments, it was found that no large overhead is added to the system by using the control service. This supports the cost performance comparison with the cost performance in the case of failure modes as the latter involves relatively high cost. In particular, the continuous measurement of the runtime changes to the software service using a feedback process, where the measured maximum latency is used as a checkpoint allows for monitoring the control service performance itself. For example, if the control process for sorting any array size using one of the selected sorting algorithms (e.g. bubble, selection, and quick sorting algorithm) is over 0.4 (ms) the system will stop or detect any further conflicts.
- The ability to generically manage a distributed software system without the need to rebuild/recode the whole system by using the control service. The system is then able to control its behaviour in the case of unexpected runtime changes.
- The flexibility to apply the same control service to any system (i.e. application services), with the same results by a reasonable degree of "separation of concern" between the middleware control meta-service and both the core middleware service and the user application service. This separation of concerns between the system controller, the service managers and the JavaSpace service reduces complexity as the service manager assists the system controller in the system control process.

9.6 Summary

This chapter presents the evaluation results of our approach where the main goal is to delineate the required functionality of an autonomic control service for the existing middleware of a distributed software system. In meeting our aim, we analysed the relation between a well-known control theory approach and that of self-adaptive software and how the latter could be used to consider the evaluation issues in a self-adaptive software environment.

The measurement results indicated that the performance profile of the system with the autonomic middleware control service is more efficient than the system without the autonomic middleware control service. Although the elapsed time to control the system increased in some proportion to the number of services in the system, it nevertheless remained flexible and efficient. Furthermore, the elapsed time in the case of resolving conflict with the control service is higher than the elapsed time without the control service. Nevertheless, it guarantees that the system will perform its tasks without shutdown, disturbance of the whole system, or incurring huge costs in the case of outright failure. In addition measurement of the average latency of the control service to monitor the performance of the control service itself and stop any conflict during the process time.

Chapter 10

Conclusions

10.1 Motivations and Approach Summary

Over the recent years, the notion of software autonomy and self-adaptation has generated a flurry of research interests focusing on design, analysis and/or management related theories, models, middleware and/or tools to support runtime software dynamic adaptation and self-healing.

IBM[26] has promoted a characterization of autonomic computing in which systems are intended to have in-built capabilities for self-management, self-healing and/or self-protecting. Prior to the IBM autonomic computing initiative, DARPA-funded initiative on self-adaptive software has explored these issues through the application of control theory, AI planning and software reflection providing mechanisms for developing self-adaptive application; with a specific focus on the generation of generic software programming and control models for runtime self-adaptation. Other large initiatives such as, DARPA-funded Dynamic Assembly for system Adaptability, Dependability, and Assurance (DASADA) is exploring advanced software engineering concerns related to dynamic and self-healing software systems.

Whilst, much research works are focusing on the design, analysis and/or management of self-adaptive, autonomic software, and reflective middleware for adaptive software systems, this research project sets out to investigate the generic requirements to support the develop of software self-governance through meta-control services to provide a foundation for what can be termed as "adjustable autonomy" capabilities, through which for instance software systems can manage their operation and self-governance in accordance with their respective users' requirements, goals, norms and environments and self-adapting to detected inconsistencies (changes) in a guaranteed, predictable and safe manner.

In order to achieve this goal a number of technical challenges have to be addressed including;

- Reference model: the development of a baseline architecture and/or software design pattern for designing, deploying, and managing self-adaptive software which can self-monitor and analyse their own behaviour and self-adapt in the event of any detected change. Further considerations needs to be addressed such as:
    - Supporting predictable and normative triggers to facilitate lifetime management by detecting, filtering and repair system behaviour.
    - Supporting coordination aspects for solving conflicts emerging form self-repair strategy prior to its enactment.
    - Supporting dynamic configuration to enable the autonomic control service to customize and adapt the repair strategies according to a considered system's requirements, environment and domains.
    - Supporting a usage of normative model to specify the management 'norms and policies that self-governs the intended repair decision itself
- Experimental model: the practical demonstration of the systems and its subsystems, which monitor, repair, and reconfigure runtime behaviour considering the system properties, requirements and intended decision.

In line with the above indicated motivations and associated challenges, this thesis detailed a proposed meta-control model with its associated baseline architecture and autonomic middleware services to support software applications lifetime management through a deliberative self-adaptation.

- For theoretical support the research visited a number of fields including;

1. Self-adaptive software: using the feedback and feedforward mechanisms to providing continuous runtime monitoring and evaluation of system's behaviour against its goal and desires to select the appropriate decision.
2. Advanced Software engineering: using distributed middleware to facilitate the communication and the coordination between both base services (users application service) and the meta-services (autonomic middleware control services), and to bridge the gap between network layer and the application layer.

In addition using event-based notification mechanism to support direct communication between base and/or meta-level services. Also applying the exception handling concepts for safe shutdown and termination, and using the concepts of distributed shared space for developing distributed shared service that provides remote system coordination distributed database for storing the required information for control and coordination process.

3. Software Agent: extending the BDI (EBDI) model (Sec. 8.2.2) to filter the intended decision in the case of unpredictable environment changes, and acting to changes in accordance to their situated BDI grounded in normative settings.

In particular, this work provided by representing both practical and theoretical support for distributed application self management, namely:

1. The practical support is providing the essential and required infrastructure for developing an the computational and programmable model for autonomic middleware control service, such as
2. Service management, which is responsible about the service control and it control sequence process considering service monitor, service diagnosis, service repair operator and service adaptor strategies
3. System management, which is responsible for the control, coordination and the reconfiguration of the whole system to guarantees that the interacted service are still coordinated maintained.
4. Distributed shared space, which allow all system' service to be shared over the network and store any other information required for the control process (i.e. it act as database service but it is different from the relational database).
5. Control rule base, which is accessible by other services to define to define and determine the system ranges and intervals for each specific application and its repair and reconfiguration strategies.

10.2 Contributions

One of the main contribution of this work is defining the generic requirements and designing a baseline architecture that is essential for providing developers with a references model to design, deploy and manage distributed self-adaptive software

system by encompassing components such as; monitor, repair, and system enactment and reconfiguration in the case of any conflicts or inconsistencies in such uncertain system environment.

In addition, the work provided an insight into the design of a deliberative mechanism for runtime software components and services federations' self-governance to ensure safe and predictable software control and reconfiguration. This mechanism was based on a proposed extension to the Beliefs, Desires, and Intension (BDI) model, which is referred to here as the Extensible BDI (EBDI) model (Sec. 8.2.2). In particular, the EBDI model provides means and mechanisms to underpin the software coordination, supervision and governance during for instance either the self-management or self-healing processes.

In this work a proof-of-concept was implemented were the middleware control services was designed using the EBDI model in that:

- Beliefs; correspond to service information derived and/or accessed from a range of sources, including; domain, environment or beliefs of other services.
- Desires; represent the state of affairs (i.e. in an ideal world), which often maximise the service's own goals. By comparing a system beliefs set (observed system states) against its desires, the system may detect a mismatch and triggers (instantiate a set of intentions) [66].
- Situated intentions; representing action sets for the system to undertake in a given situation to achieve its specified desires and/or to address the mismatch between the system environment (beliefs) and the system's desires (goals).
- Normative intention; representing a set of actions to be undertaken to ensure a specified set of norms including obligation and responsibility rules are observed before a given intention is enacted and/or affective rules emerging as a result of an enacted intentions set.
- Utility intention; represents a set of system actions to optimise its goal-oriented intentions.

Also, the development of software meta-control model is considered one of the main strength and contribution of this research, which provides a mechanism of how distributed applications services act and interact with their associated middleware

management and control service. The main elements of the software *meta-control arc* summarized below:

- The Service *Manager* is concerned with managing its service conflicts. Hence for each service there is a manager that looks after that service. The service manager has a hierarchy of control scripts/tasks that are:
    - The monitoring model uses a set of control rules to check monitored behaviour and architectural configuration and hence detects conflicts.
    - The diagnosis model involves the execution of control rules, activated by conflicts that identify and classify the conflict types to provide the basis for the selection of a conflict resolution operator.
    - The repair model is specified using contract-based assertions, pre-conditions and typical operators to provide operations that resolve a service's conflict. These operations are represented as primitive operations integrated into the service manager, such as notify operator, repair operators, or thrown appropriate exception operator. The service manager after that store its service state in a shared space (JavaSpace Service) for monitoring by the system controller.
    - Adaptation Engine: in which the service manager has to adapt the service according to proposed changes.
- The system controller is responsible for establishing and managing the coordination of the overall system application services and ensures that the interrelated system services are maintained and coordinated. The system controller regularly checks the service state that stored previously by the service manager in the distributed shared space (i.e. JavaSpace), whenever the system controller detect any failure in and of its system's service. Then it applies the appropriate resolution strategy; the main models that are included in the system controller are:
    - The system monitor, which has the ability to collect and feedback the information that is required to select and apply the resolution strategies in the case on any conflict/inconsistencies.

- The system repair strategies, which determines when, where, and how the repair/resolution or adaptation should be. Our repair/resolution strategies are formatted in an external markup language (i.e. XML format) and used to evaluate the effect of various alternative solutions based on the proposed EBDI model that mentioned earlier.
- The system reconfiguration, which applies the required reconfiguration operators that are attached to the resolution strategy. For example, if the resolution strategy selects an alternative service to a failed service, the reconfiguration system should establish the required changes that result from the resolution strategy dynamically at runtime. For example, *getNewManager(), notifyClient() and newConnect()* reconfiguration operators.
- The system associated interpreter that is used to dynamically translate the external format (e.g. XML) of repair strategies actions or operators to a lower-level and executable level that is used in the code, therefore this model allows run-time changes within the code without the need to recode or recompile the system again.

- The JavaSpace service, which provides a mechanism to coordinate the relationship of shared resources or services in the distributed application over the network. In addition, it has the ability to store the required information (Sec 7.4).

10.3 Achievements

The main achievements of this research of autonomic lifetime management have focused on the following aspects:

- The *separation of concern* between the management services (meta-service) and the application service (base-service) for developing a dynamic, adjustable, flexible, and generic control service. This required addressing the main requirements for autonomic control service including; conflict detection that examine the current

behaviour against which behaviour is monitored using a set of control rules, conflict identification and classification, that activated as a result of the execution of the control rules in the case of conflict detection to identify, and classify the conflict, which is used as a basis for selecting the appropriate repair strategy, and then interpreted with associated system interpreter.

- Based on the research requirements, the main strength of the proposed approach is that it can support both *decentralized and centralized* control supported by middleware applications' service manager and the system controller respectively. In other words, the service manager facilitates *decentralized* control and management of the service, which provides a separation of concerns and reducing the complexity from the system controller (i.e. the system central control unit). While the system controller facilitates *centralized* management of the interrelated application services, which provide the control service with the ability to coordinate and integrate the interrelated distributed application services, and therefore reconfigures the whole system according to the selected repair and reconfiguration strategies considering such coordination aspects.

- In addition the usage of the distributed shared space (i.e. JavaSpace service) supports the design of the proposed approach with flexible, accessible, and distributed space for assisting the system controller to monitor, repair, coordinate and reconfigure the application services against the detected conflicts/inconsistencies, also the JavaSpace service used as database service for storing the information required in the control process to be checked or notified to the interested parts.

- The developed model and architecture was implemented and evaluated using a number of *case studies*[15] to provide a proof-of-concept or evidence of potential benefits of such autonomic middleware control services (meta-control model) and associated baseline architecture to distributed application life-time management and self-adaptation.

[15] Namely; GridPC, 3in1phone, web-based information service, and sorting algorithms.

10.4 Thesis Summary

This thesis has offered a new vision of lifetime management of distributed application services grounded in a number of related disciplines such as; self-adaptive software, software agent, and advanced software engineering. The detailed description of background theories, methods and the achievement of this project are presented as follows;

Chapter 1 introduced the motivations and technical challenges, and outlined the proposed approach, and main contribution to the work.

Chapter 2 introduced the required background principles, theoretical models and definitions including; (1) self-adaptive software's definition, directions, categories, and examples. (2) autonomic computing that controls computing system's key functions without conscious awareness or user intervention and, increase the productivity while reduce complexity from users [26]. We define the autonomic computing architecture and requirement that adapts and generates the dynamic changes.

Chapter 3 discussed the essential technologies for designing and managing distributed object-oriented systems such as; Service-Oriented Programming (SOP) and object-oriented middleware. These technologies provide a new generation of distributed computing applications development and management, which has the capabilities to reduce management time and integration of obtained packages and components for running system. However, this is not enough *either* for lifetime management of systems that may exist in unpredictable and unexpected environment, *or* for establishing the required changing in requirements with less human intervention and minimize complexity for usage.

Chapter 4 reviewed the state-of-the-art and related work relevant to the control and management aspects. This review was structured a long static management of distributed system, dynamic management of distributed system. Most of such research works have been focused on user-based intervention and management. Though, there are on-going some research works relevant to self-management and self-adaptation aspects including; those focused on policy-based management, architecture-based management, and context, resource and QoS-aware software.

Chapter 5 highlighted the requirements and design aspects *of meta-level control* service, which also outlined the main elements required to allow distributed application services to interact with their middleware to solve runtime arisen conflicts or inconsistencies, and reconfigure to enact a prescribed repair plan. The requirements for solving the conflict of self-adaptive software are (i) conflict detection, which uses of a set of control rules against which behaviour is monitored to detect conflicts, (ii) conflict identification and classification, that activated as result of the execution of the control rules which locate, identify, and classify the failure, (iii) conflicts resolution strategies, which selects the proper strategy for the identified failure, and (iv) system reconfiguration provides capability for our approach to reason about the current state and re/.configuration of an application in order to assess the validity of an coordination/configuration strategy, (v) system interpreter as the reconfiguration model is dynamically attached with system associated interpreter, which is responsible for translating the repair operators from text format to executable format.

Chapter 6 detailed the baseline architecture and design presented in context with a range of current research on software architecture models, such as coordination, autonomic computing, deliberative systems, normative systems and adjustable autonomy. The design of the autonomic middleware control service considered three main service layers, which are middleware core services layer, autonomic middleware control layer (meta-service), and user's application layer (base-service). The Autonomic middleware service (meta-service) includes three main services that are Service Manager, JavaSpace Service, and System Controller for runtime self-control conflicts.

Chapters 7 and 8, presented the three main parts of prototypical system implementation based on Java API's and Jini network. Three applications are namely, GridPc, 3in1Phone, and web-server applications were used to illustrate the implementation of the main three services of our autonomic middleware service (meta-service) that are, Service Manager, JavaSpace service, and System controller service. Service manager dynamically looks after its service by applying sequences processes, which are monitoring, diagnosing, repairing and adapting processes (e.g. the repair operator such as notify (), add_Client (), remove_Client (), and catch (Exception e)). The system controller is responsible for the system control process in the case of conflict in any of its system's service by applying the required resolution and reconfiguration strategy,

which dynamically interpreted from XML document to run-time executable model using its associated interpreter, while the JavaSpace service is used for providing the control service with a distributed shared space which is used for system sharing resources/services, coordination, and used as well as a storage or database for the service information to be available or to be notified to the system controller in the occurrence of conflicts.

Chapter 9 presented a qualitative and quantitative evaluation of the main functionality of the autonomic control services extending current Jini middleware . Our starting point is by applying well-known control theory based metrics to support the evaluation of self-adaptive software. Both the quantitative and qualitative evaluation's results have indicated that the performance profile of the distributed, running software system with the autonomic middleware control service is more efficient than the same software system issues but without the autonomic middleware control service. Though, as shown in the quantitative experiments the additional autonomic middleware control services add further computational overheads[16], which can be balanced against the benefits of software safe self-adaptation including lifetime evolution capabilities without requiring total system shutdown, disturbance of the whole system, or incurring huge costs in the case of outright failure. In addition, the qualitative evaluation has used a set of metrics such as functionality, generality, flexibility, extensibility to assess the autonomic middleware control service throughout the developed case studies namely; GridPC, 3in1Phone, web-server application, and sorting algorithms. This evaluation indicated the following main points; (1) the autonomic control service facilitates self-control mechanism of distributed software system without the need to recod the whole system at run-time. (2) The dynamic, generic and autonomic features that are provided by such control service has been facilitated the control of any distributed software system such as developing associated interpreted to translate the text format resolution strategies to on-demand executable strategies at runtime. (3) The separation of concerns between the service manager and both application service and the system controller reduces the complexity. (4) The continuous evaluation of the control service itself by applying both continuous monitoring and feedback processes to detected any failure could generated in the control process itself. (5) The generic and standard technology that used to

[16] In addition we evaluated the control service itself by measuring the average latency of the models that supports

support the development of our control process by using abstract core middleware services (e.g. JavaSpace service).

Chapter 10 provided the thesis summary, and suggestions for the future works

10.5 Discussion

This Thesis focused on the design and development of control service to support distributed self-adaptive software, in which the main contributions can be summarised as follows:

Design and development of meta-control model to enable safe and adjustable autonomy of distributed applications.

Development of a meta-level control service that includes three main services that required for self-controlling and self-governance the base-services. The control services are service manager entities that look after its services, system controller for controlling and reconfiguring the whole system to accommodates with repair changes, and the associated distributed shared space and in addition control rules or constraint.

Proposal of autonomic middleware control service, which includes a programming model to facilitate the development of autonomic control middleware services that would facilitate lifetime management of distributed application service over the network by detecting, recover any runtime conflicts through applying its control repair strategies, policies and norms.

Use of a shared distributed space to provide a coordination and awareness medium for system's operation management and meta-level services communication and interaction.

Develop of a control strategy markup language and associated interpreter to achieve a level of separation of concerns, and externalisation of control functionality of software and knowledge allowing runtime changes without middleware software system recoding and/or shutdown.

10.6 Future Work

This work proposed and detailed a middleware extension to provide adjustable autonomic software. However, there a number of further works and extensions to be explored including;

- The need to extended the control service with the ability of self-protection, as the security and privacy functions are essential for self-protecting the meta-control functionality, and the system information especially in the untrustworthy environment. The data are one of the main points that could be a target for any attack or failures, so it is essential to provide the system with the ability to protect itself, which could be safely established by backup these data and stored automatically. Backup could be done regularly after certain time or proactively when received interested event, such as detecting an elementary disk failure [1]. There are other self-protection ways of protection, such as randomness attack could be protected by digital signature, password, and signature verification [1].
- Further work are also needed to provided the present model with the machine learning service providing the control service access to history and knowledge related to the previous failures cases to avoid and repair strategies.

- Further work is required to test and evaluate the scalability of this model on large scale or open system, as our approach has been tested on relatively small-scale applications and case studies. The scalability challenges will emerge not only from the number of applications services and meta-services to deploy and manage thus affecting the overall systems response and robustness, but also due to organisational and architectural complexity of having to use as in the multi-domain environment "super-controller" to coordinate the life-cycle management of local "sub-controllers", which in their turn

govern the life-cycle management of their assigned application federation.

Appendix A

Distributed System Development

Introduction

Distributed system can be defined as: "a collection of autonomous hosts that are connected through a computer network with each host executing components and operating a distributed middleware to enable components to coordinate their activities giving the impression of a single, integrated computing facility[139]

The distributed application development is based on dependency relationships between the stages in process shown below[139].

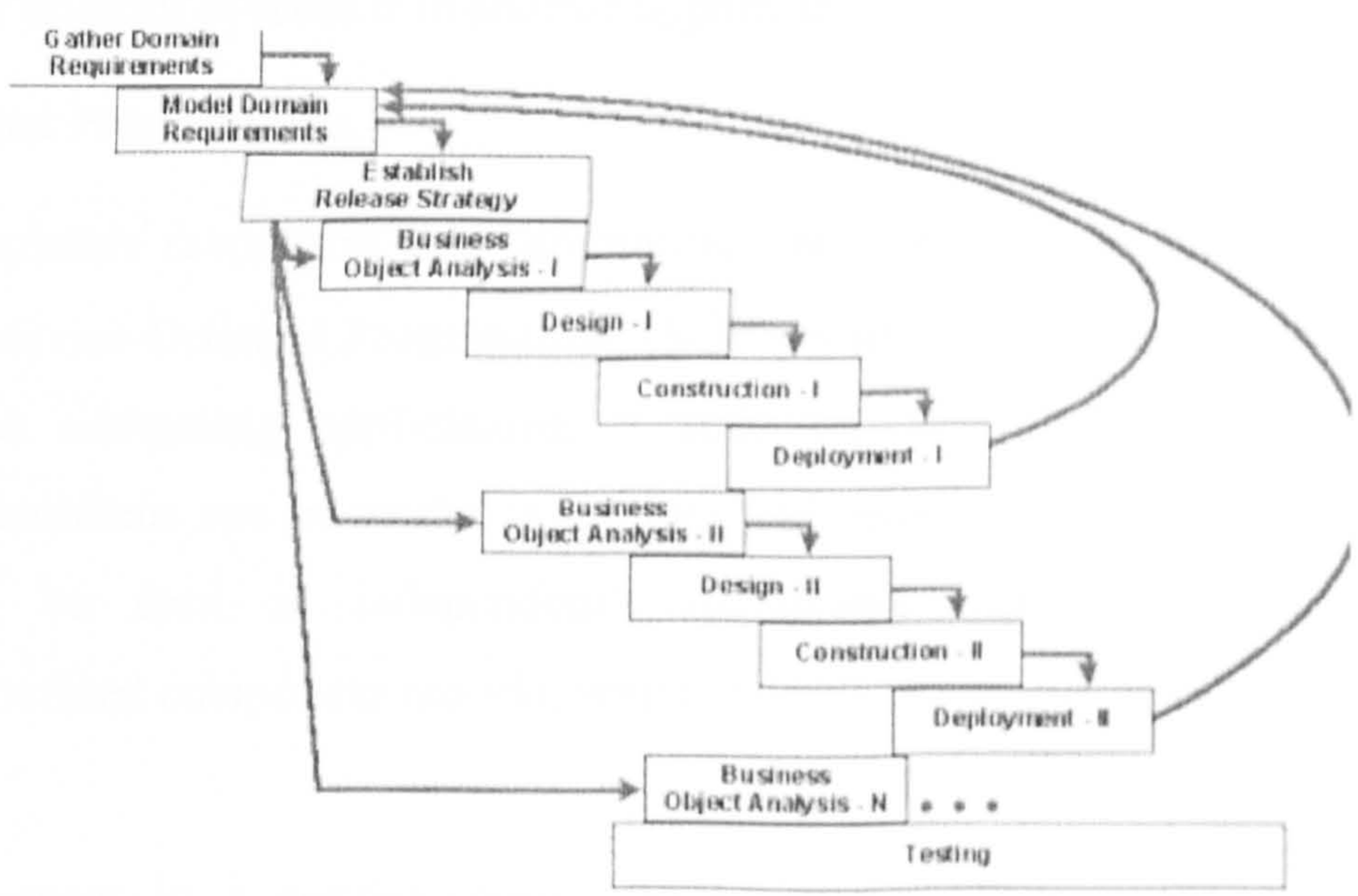


Figure A.1: The Distributed Application Development Process [139].

Distributed system technology has much in common with and is often served by object technology and software components. There are many points of synergy between these technologies, including a focus on real world modelling, as well as a focus on

simplicity, reusability, extensibility, and productivity. Distributed development is based on several key elements shared with the object technology:

- Concurrent development of packages and components.
- Reuse of software components similar to polymorphism.
- Cyclical and incremental development.
- Release strategy.

Object Technology

Object technology provides significant potential value in three areas, all closely related: productivity, maintainability, and paradigm consistency [140].

Object technology can be remarkably effective in allowing the structure of the application to be consistent throughout its development and maintenance phases, where development is object oriented through all phases, it is much easier to do rapid prototyping, to maintain consistency across the life cycle, and even to reuse components. For instance if an object, is represented consistently at each phase, it can be reused. But if its run-time implementation is spread across the application's running code, it is very difficult to reuse it in another application.

Service-Oriented Programming

The most remarkable revolution in programming since Object-Oriented Programming (OOP) is the Service-Oriented Programming (SOP), which enable a new generation of service-oriented computing applications. It adds the idea and the concept that programming problems can be model as services and could be use. The programming problems can be seen as independently deployable black box client/servers independence defined *component* models, which communicate with each other through *contracts*.

An essential aspect in a service-oriented architecture is how components locate services. Service location information is normally hard coded in software components, or saved in a configuration file that reads on start-up by the components. In reality, networked system is dynamic. Software and hardware components are replaced or upgraded, nodes enter and leave the network, which is creating a large management problem for systems configured statically. These systems are simply not built to recover from network errors or failed services, which require partial solution consists of two concepts should considered *a) discovery service* and *b) service lookup*. Components

"discover" the environment in which they are deployed and "lookup" services they need dynamically, which provided by a fully functional frameworks like the Java™, and some middleware.

The distributed middleware, plays a very important role by providing APIs and support functions that effectively bridge the gap between network operating system and distributed application components and services. Middleware is defined as a set of services required for providing connectivity and management services in a distributed computing environment. These services include database connectivity, messaging, remote procedure calls, object request brokers, transaction services, timing services, and naming services.

The main features has been provided by distributed middleware are:

- Improve Productivity
- Improve Efficiency
- Improve Customer Service
- Reduce Costs

The two main categories of distributed middleware, which is essential to describe in here, are [141]:

- *Distributed Object Middleware:* Distributed object middleware provides the abstraction of an object that is remote yet whose methods can be invoked just like those of an object in the same address space as the caller. Distributed objects make all the software engineering benefits of object-oriented techniques - encapsulation, inheritance, and polymorphism available to the distributed application developer. Jini and Corba are examples of the distributed object middleware.
- *Distributed Tuples:* A distributed relational database offers the abstraction of distributed tuples. Its Structured Query Language (SQL) allows programmers to manipulate sets of these tuples (a database) with intuitive semantics and rigorous mathematical foundations based on set theory and predicate calculus. Linda is a framework offering a distributed tuple abstraction called Tuple Space (TS). Linda's API provides associative access to TS, but without any relational semantics. Linda offers spatial decoupling by allowing depositing and withdrawing processes to be unaware of each

other's identities. It offers temporal decoupling by allowing them to have non-overlapping lifetimes. Jini is a Java framework for intelligent devices, especially in the home. Jini is built on top of *JavaSpaces*, which is very closely related to Linda's TS.

The main purpose of middleware services is to help solve many application connectivity and interoperability problems. However, middleware services are not a panacea: [144].

- There is a gap between principles and practice. Many popular middleware services use proprietary implementations (making applications dependent on a single vendor's product).
- The whole number of middleware services is a barrier to using them. To keep their computing environment manageably simple, developers have to select a small number of services that meet their needs for functionality and platform coverage.
- While middleware services raise the level of abstraction of programming distributed applications, they still leave the application developer with hard design choices. For example, the developer must still decide what functionality to put on the client and server sides of a distributed application [115, 141].

To overcoming these three problems is to fully understand both the application problem and the value of middleware services that can enable the distributed application. To determine the types of middleware services required, the developer must identify the functions required, which fall into another problems. So the middleware technology is not enough to solve the distributed problem discussed before but it needs some kind of development to solve that partial of the technical problems such as middleware management services to be continuously monitored and changed to ensure optimum performance of the distributed environment.

Appendix B

Distributed *Middleware*

Distributed Middleware

Distributed middleware, plays a very important role by providing and support APIs with functions that effectively bridge the gap between network operating system and distributed application components and services. Middleware is defined as a set of services required for providing connectivity and management services in a distributed computing environment. These services include database connectivity, messaging, remote procedure calls, object request brokers, transaction services, timing services, and naming services.

Programming with Middleware

Programmers do not have to learn a new programming language to program middleware. Rather, they use an existing one they are familiar with, such as C++ or Java. There are three main ways in which middleware can be programmed with existing languages.

- The first way is where the middleware system provides a library of functions to be called and utilize the middleware; as distributed database systems and Linda do this.
- The second way is through an external interface definition language, as the IDL file describes the interface to the remote component, and a mapping from the IDL to the programming language is used for the programmer to code it.
- The third way is for the language and runtime system to support distribution natively; for example, Java's Remote Method Invocation (RMI).

The main purpose of middleware services is to help solve many application connectivity and interoperability problems. However, middleware services are not a panacea[145]:

- There is a gap between principles and practice. Many popular middleware services use proprietary implementations (making applications dependent on a single vendor's product).
- The whole number of middleware services is barrier to using them. To keep their computing environment manageably simple, developers have to select a small number of services that meet their needs of functionality and platform coverage.
- While middleware services raise the level of abstraction of programming distributed applications, they still leave the application developer with hard design choices. For example, the developer still must decide what functionality to put on the client and server sides of a distributed application [145].

The key for overcoming these three problems is to fully understand both the application problem and the value of middleware services that can enable the distributed application. To determine the types of middleware services required, the developer must identify the functions required, which fall into one of three problems [145]:

1. Distributed system that include critical communications, program-to-program, and data management services. This type of service includes RPCs, MOMs and ORBs.
2. Application enabling services, which give applications access to distributed services and the underlying network. This type of services includes transaction monitors and database services such as Structured Query Language (SQL).
3. Middleware management services, which enable applications and system functions to be continuously monitored to ensure optimum performance of the distributed environment [145]

So the existing middleware services are not enough to solve and control the distributed problem discussed before but it needs some kind of management to solve that practical and technical problem.

Appendix C

Grid Computing

Grid Technology

Grid computing is a form of distributed computing that involves coordinating and sharing computing, application, data, storage, or network resources across dynamic and geographically dispersed organizations [110]. Grid Computing enables the virtualisation of distributed computing resources such as processing networks bandwidth and storage capacity to create a single system's logical view, and when applications need access to distributed computing resources. Just like an Internet user who views a unified instance of content via the Web, a Grid user essentially sees a single, large virtual computer.

Grid computing infrastructures [146] offer a wide rang of distributed resources to applications. However, the heterogeneity of both the network and computing resources, and the dynamic load conditions make adaptation an important requirement for grid applications. For example, the applications must be able to adapt themselves at runtime to handle such things as resource variability (e.g. network bandwidth, server availability, etc.), and system faults (e.g. servers and networks going down, failure of external components, etc.). So if the system is not adaptive, it will provide poor performance. The principals benefits that the Grid will bring are [146]:

- Enabling more effective and seamless collaboration of dispersed communities, both scientific and commercial.
- Enabling large-scale applications comprising of 10,000 computers, large-scale pipelines etc.
- Transparent access to "high-end" resources from your desktop.
- Provide a uniform "look & feel" to a wide range of resources.
- Location independence of computation resources as well as data.

Grid Architecture Description

The establishment, management, and exploitation of dynamic, cross-organizational sharing relationships require new technology. This technology is Grid architecture and supporting software protocols and middleware [110]. Grid architecture, starts from the perspective that effective operation requires to establish sharing relationships among *any potential participants.* In a networked environment, interoperability means common *protocols. Hence, their Grid architecture is first and foremost protocol* architecture, with protocols defining the basic mechanisms *by which users and resources negotiate,* establish, manage, and exploit sharing relationships [110]. This description of Grid architecture identifies requirements for general classes of component and the result is an extensible, open architectural structure which can be placed solutions to key user requirements. The architecture is organized into component layers as shown below, components within each layer share common characteristics but can build on capabilities and behaviours provided by any lower layer. The architectural description is high level and places few constraints on design and implementation, as the layered Grid architecture and its relationship to the Internet protocol architecture shown below [27], and follows a brief description for more detail about each component:

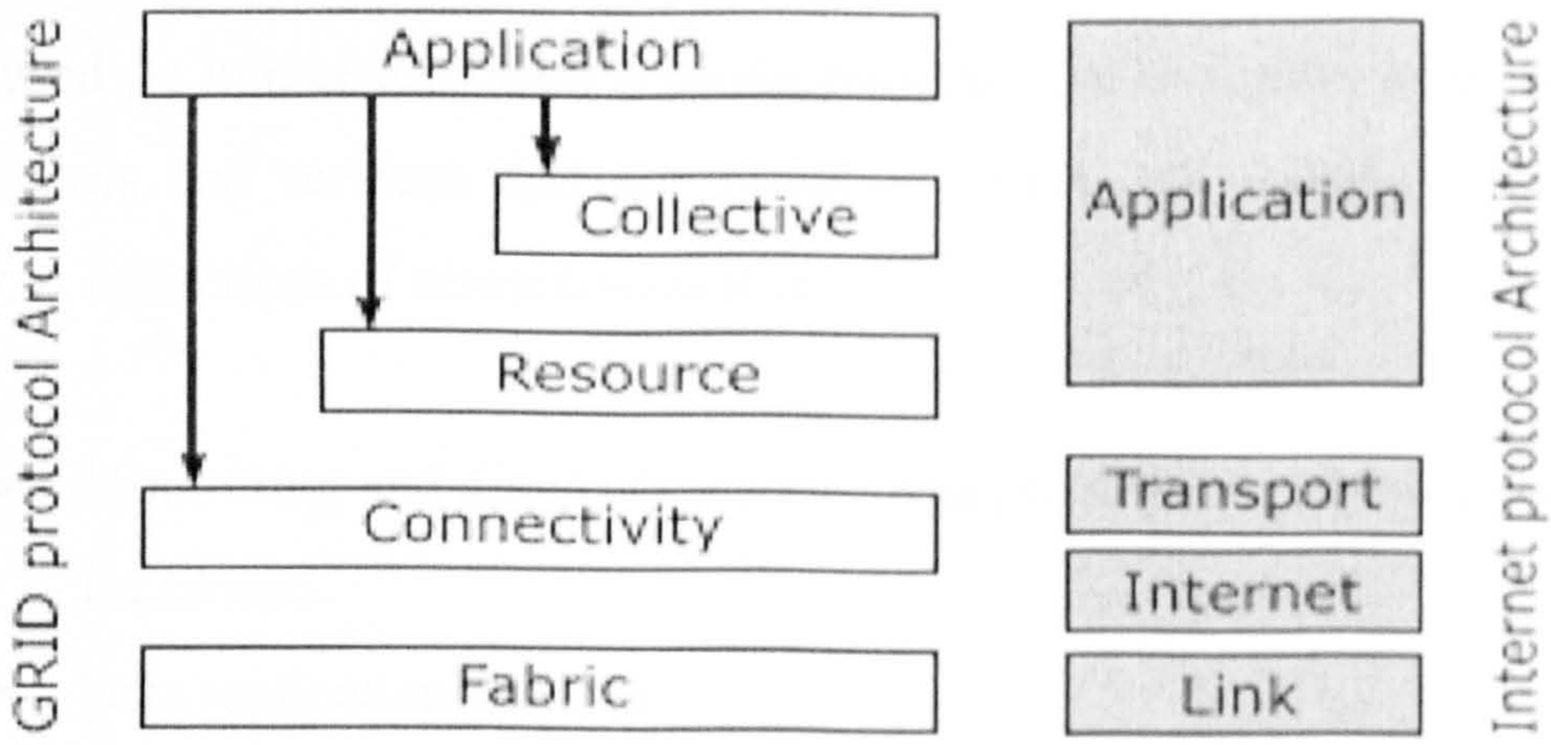


Figure C.1: GRID Protocol Architecture [27].

- *Fabric*: Interfaces to local control, the grid fabric layer contains the resources that are shared. This could include computational power, data storage, sensors etc. This sharing is controlled by grid protocols but the resource could include local networks. In this case the local protocols take over at this point. The grid system is just concerned with access above this point. Systems can be implemented at the fabric level to support resource scheduling and other higher operations

- *Connectivity*: communicating easily and securely, the connectivity layer contains the communication and authentication protocols required for grid-specific network transactions. Communication protocols enable the exchange of data between different fabric layer resources. Authentication protocols build on communication services to provide secure mechanisms for verifying the identity of users and resources.
- *Resource*: sharing single resources, the resource layer uses the communication and security protocols of the connectivity layer for secure control, negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. Resource layer protocols call Fabric layer functions to access and control local resources. Resource layer protocols are concerned entirely with individual resources. There are two classes of Resource layer protocols: (i) Information protocols - used to obtain information about the structure and state of a resource, for example, its configuration, current load, and usage policy such as (ii) Management protocols - used to negotiate access to a shared resource, specifying, for example, resource requirements and the operations to be performed.
- *Collective*: coordinating multiple resources, while the resource layer is focused on interactions with a single resource, the collective layer contains protocols and services that are global in nature and capture interactions across collections of resources such as.

    - Monitoring and diagnostics services support the monitoring resources for failure.
    - Data replication services
    - Grid-enabled programming systems enable familiar programming models to be used in Grid environments.
    - Workload management systems and collaboration frameworks are known as problem solving environments.
    - Software discovery services discover and select the best software implementation and execution platform based on the parameters of the problem being solved.
    - Community authorization servers.

- Community accounting and payment services.

- *Applications*: The final layer in the grid architecture comprises of the user applications. Applications are constructed in terms of services defined at each layer in the grid structure. At each layer well-defined protocols provide access to some useful service such as resource management, data access, resource discovery, and so forth. At each layer protocols and services are used to perform desired actions. Application Program Interfaces are implemented by Software Development Kits (SDKs), which in turn use Grid protocols to interact with network services that provide capabilities to the end user. Higher-level SDKs can provide functionality that is not directly mapped to a specific protocol, but may combine protocol operations with calls to additional applications as well as implement local functionality.

REFERENCES

1. IBM."Autonomic Computing Concepts".accessed May.2003, http://www-3.ibm.com/autonomic/pdfs/AC_Concepts.pdf.
2. P.Robertson, R.laddaga and H.Shrobe. "Introduction: the First International Workshop On Self-Adaptive Software". *in Proceeding of Proceedings First International Workshop on Self-Adaptive Software (IWSAS2000)*. 2000.
3. A.Meng. "On Evaluation Self-Adaptive Software". *in Proceeding of First International Workshop on Self-Adaptive Software (IWSAS2000)*. 2000.
4. P.Robertson. "An Architecture for Self-Adaptive and its Application to Aerial Image Understanding". *in Proceeding of First International Workshop on Self-Adaptive Software (IWSAS2000)*. 2000.
5. D.L.Wells and J.Nagy. "Gauges to Dynamically Deduce Componentware Configurations". *in Proceeding of*. 2002: DARPA (Program Sponsor) http://schafercorp-ballston.com/dasada/projectlist.html..
6. N.Badr, D.Reilly and A.Taleb-Bendiab. "A Conflict Resolution Control Architecture for Self-Adaptive". *in Proceeding of Proceedings of International Workshop on Architecting Dependable Systems WADS 2002 (ICSE 2002)*. May 2002. Orlando, Florida.
7. M.Allen, N.Badr, E.Grishikashvili, and A.Taleb-Bendiab. "Adaptation Engine: an Agent-Based Framework for ad-hoc Service Life-Cycle Management for Meta-Computing". *in Proceeding of Processing of AISP symposium on (IA and Grid Computing)*. 2002. Imperial college London.
8. E.Grishikashvili, N.Badr, D.Reilly, M.Allen, M.Yu, and A.Taleb-Bendiab. "Autonomic computing: A Service-Oriented Framework to Support the Development and Management of Distributed Applications". *in Proceeding of in Processing of 3rd Annual Postgraduate Symposium on The Convergence of Telecommunications, Networking & Broadcasting (PGNet2002)*. 2002. U.K.
9. E.Grishikashvili, N.Badr, D.Reilly, and A.Taleb. "From Component-Based to Service-Based Distributed Applications Assembly and Management". *in Proceeding of Proceedings 29th EUROMICRO CONFERENCE*. 2003. Turkey.
10. D.Reilly, N.Badr, E.Grishikashvili, and A.Taleb-Bendiab. "Service-Oriented Approach for Distributed Application Assembly and Management". *in Proceeding of Processing of 4th Annual Postgraduate Symposium on The Convergence of Telecommunications, Networking & Broadcasting (PGNet2003)*. 2003. U.K.
11. D.Reilly, N.Badr, A.Taleb-Bendiab, and A.Laws. "An Instrumentation and Control-based Approach for Distributed Application Management and Adaptation". *in Proceeding of Proceedings of ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*. 2002. Charleston, USA.
12. S.Cheng, D.Gralan, B.schmerl, J.Sousa, B.Spitzngel, and P.Steenkiste."Using Architectural Style as a Basis for System Self-repair".accessed http://www.cs.cmu.edu/afs/cs/project/able/ftp/wicsa3-arch/WICSA-submitted.pdf..
13. S.Rao and P.Georgeff, "BDI agents: From theory to practice". *the First International Conference on Multi-Agents Systems (ICMAS-95)*, 1995(MIT Press): p. 312-319.

14. A.Laws, A.Taleb-Bendiab, S.Wade, and D.Reilly. "From Wetware to Software: A Cybernetic Perspective of Self-Adaptive Software". *in Proceeding of Self-Adaptive Software: Applications, Second International Workshop on Self-daptive Software*. 2003: R. Laddaga, P. Robertson, and H. Shrobe, Editors.
15. M.Kokar, K.Baslawski and Y.Eracar, "Control Theory-Based Foundation of Self- Controlling Software". *IEEE Intelligent Systems*, 1999: p. 37-45.
16. R.Laddaga. "*Active Software*". *in Proceeding of Proceedings First International Workshop on Self-Adaptive Software (IWSAS2000)*. 2000.
17. G.Karsai and J.Sztipanovits, "A Model-Based Approach to Self-Adaptive Software". *IEEE Intelligent Systems & their Applications*, 1999. 14(3): p. 46-53.
18. Y.Eracar and M.Kokar, "An Architecture for Software that Adapts to Changes in Requirements". *Journal of Systems and Software*, 2000. 50: p. 209-219.
19. P.Oreizy, M.Gorlick, R.Taylor, D.Heimbigner, G.Johnson, N.Medvidovic, A.Quilici, D.Rosenblum, and A.Wolf, "An Architecture-Based Approach to Self-Adaptive". *IEEE Intelligent Systems*, 1999. 14: p. 54-62.
20. L.J.Osterweil and L.A.Clarke. "Continuous Self-Evaluation for the Self-Improvement of Software". *in Proceeding of First International Workshop on Self-Adaptive Software (IWSAS2000)*. 2000.
21. G.Simon, T.Kovacshazy and G.*Peceli*. "Transient Management in Reconfirgurable Systems". *in Proceeding of First International Workshop on Self-Adaptive Software (IWSAS2000)*. 2000.
22. A.LedecZi, A.Bakay and M.Maroti. "Model-Integrated Embedded Systems". *in Proceeding of First International Workshop on Self-Adaptive Software, (IWSAS2000)*. 2000.
23. D.Karuppiah, E.Araujo, Y.Yang, G.Holness, Z.Zhu, B.Lerner, and G.Riseman. "Software Mode Change for Continuous Motion Tracking". *in Proceeding of First International Workshop on Self-Adaptive Software, (IWSAS2000*. 2000.
24. I.Shaul. "Dynamic Self-Adaptation in Distributed systems". *in Proceeding of First International Workshop on Self-Adaptive Software (IWSAS2000)*. 2000.
25. A.Oliva, I.Garcia and L.Buzato. "The Reflective Architecture of Guarana". *in Proceeding of*. 1998 HTTP://www.dcc.unicamp.br/-oliva/guarana..
26. IBM."Autonomic Computing".accessed May.2003, http://www.research.ibm.com/autonomic.
27. I.Foster, C.Kesselman, J.Nick, and S.Tuecke, "Grid Services for Distributed System Integration". *Computer*, 2002. 35(6).
28. G.Bieber and L.Architect."Service-Oriented Programming".accessed August.2003, http://www.cs.wustl.edu/~mk1/AdHocNetworking/submissions/GuyBieber.pdf.
29. G.Bieber and L.Architect."A Service-Oriented Component Architecture for Self-Forming,Self-Healing, Network-Centric Systems".accessed August.2003, http://www.openwings.org/download/specs/openwingswp.pdf.
30. Object Management Group."Introduction to OMG's Specification".accessed August.2003, http://www.omg.org/gettingstarted/specintro.htm#CORBA.
31. OMG."CORBA® BASICS".accessed August.2003, http://www.omg.org/gettingstarted/corbafaq.html.
32. OMG, "Agent Technology Green Paper". *Agent Working Group*, 2000. 91.
33. UPnP™ Forum."Welcome to the UPnP™ Forum".accessed August.2003, http://www.upnp.org/.
34. E.Steinfeld, "Devices that play together,work together". *Automata International Marketing*, 2001. 9/13.

35. UPnP Function's".accessed March.2003, http://www.c-insitc.net.
36. O'Reilly."A Web Services Primer".accessed August.2003, http://webservices.xml.com/pub/a/ws/2001/04/04/webservices/.
37. JavaEnvironment."July.2003, http://java.sun.com.
38. J.Newmarch."Jan Newmarch's Guide to JINI Technologies".accessed http://jan.netcomp.monash.edu.au/java/jini/tutorial.1.03/Jini.xml.
39. Sun MicroSystem."October.2003, http://wwws.sun.com/software/jini/specs/jini1_2.pdf.
40. JiniCommunity."Augest.2003, http://wwws.sun.com/software/jini/specs/jini1.1html/js-spec.html.
41. B.Nuseibeh, S.Easterbrook and A.Russo, "Making Inconsistency Respectable in Software Development". *Journal of Systems and Software*, 2001. 56(11).
42. B.Nuseibeh, J.Kramer and A.Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification". *IEEE Transactions on Software Engineering*, 1994. 20(10): p. 760-773.
43. C.Castelfranchi. "Conflict Ontology". *in Proceeding of ECAI'96 Workshop on Modelling conflicts in AI*. 1996. Budapest.
44. S.Lander and R.Lesser. "Understanding the role of negotiation in distributed search among heterogeneous agents." *in Proceeding of the Thirteenth International Joint Conference on Artificial Intelligence*. 1993. Chambéry, France.
45. S.Cooper and A.Taleb-Bendiab. "CONCENSUS: Supporting Conflict Resolution Through Multi-Parties Negotiation in Concurrent Engineering Design". *in Proceeding of the fourth ISPE International conference on Concurrent Engineering: Research and Application (CE97)*. 1997.
46. K.Sycara, "Cooperative Negotiation in Concurrent Engineering Design". *Computer-Aided Cooperative Product Development*, 1991: p. 269-297.
47. M.Klein. "*An Experimental Evaluation of Domain-Independent Fault Handling Services in Open Multi-Agent systems*". *in Proceeding of The International Conference on Multi- Agent System (ICMAS)*. 2000.
48. S.Lander and V.Lesser. "Customizing Distributed Search Among Agents with Heterogeneous Knowledge". *in Proceeding of Proceedings of the First International Conference on Information and Knowledge Management*. 1992.
49. E.Freeman and S.Hupfer."Make room for JavaSpaces Part 1 (Ease the development of distributed apps with JavaSpaces)".accessed July.2003, http://www.javaworld.com/javaworld/jw-11-1999/jw-11-jiniology.html..
50. J.Rosenschein and G.Zlotkin, "Rules of Encounter: Designing Conventions for automated Negotiation Among Computers". *Cambridge Massachusetts The MIT press*, 1994.
51. S.Cooper and A.Taleb-Bendiab. "*A High Level Control Mechanism For Managing Conflict Resolution In Concurrent Product Design*". *in Proceeding of In Proceedings of the fourth ISPE International conference on Concurrent Engineering :Research and Application (CE97)*. 1997.
52. E.Ephrati and S.Rosenschein. "The Clarke tax as a consensus mechanism among automated agents". *in Proceeding of the proceeding of the ninth conference on Artificial Intelligent*. 1991.
53. E.Ephrati and S.Rosenschein. " Multi-Agents Planning as a Dynamic Search for social Consensus". *in Proceeding of the13th international Joint conferences on Artificail Intelligence*. 1993.

54. M.Adler, B.Davis, R.Weihmayer, and W.Worrest. "Conflict- Resolution Strategies for Nonhierarchical Distributed Agents". *in Proceeding of In Distributed Artificial Intelligent* ||. 1989. London.
55. C.Chang, M.Chung and B.lee, "Collision Avoidance of two general Robot Manipulators by minimum delay time". *IEEE Transaction on system*, 1994: p. 517-522.
56. V.Lesser and S.Lander. "Customizing Distributed Search Among Agents with Heterogeneous knowledge". *in Proceeding of 5th int. Symp. On AI Application in Manufacturing and Robotics*. 1991. Cancun Mexico.
57. K.Barber, T.liu and D.Han. "Strategic Decision-Making for Conflict resolution in Dynamic Organized Multi-Agent Systems". *in Proceeding of GDN 2000 PROGRAM*. 2000.
58. K.Barber, H.Liu and C.Han. "Strategy Selection-Based Meta-Level Reasoning". *in Proceeding of Agent Oriented Software Engineering*. 2000. Limerick England.
59. M.Klien, J.Rodriguez-Aguliar and C.Dellarocas, "Using Domain-Independent Exception Handling Services to Enable Robust Open Multi-Agent Systems". *The Case of Agent Death Journal for Autonomous Agents and Multi-Agent Systems*, 2003. 7.
60. A.Visser, "An exception -handling framework". *International Journal of computer Integral Manufacturing*, 1996(Special Issues on CIM Taxonomies).
61. M.klein., "A Knowledge-Based Approach to Handling Exceptions in Workflow Systems". *Journal of Computer-Supported Collaborative Work*, 2000. 9 3/4(Special Issue on Adaptive Workflow Systems).
62. M.Klein and C.Dellarocas. "Towards a Systematic Repository of Knowledge About Managing Collaborative Design Conflicts". *in Proceeding of Proceedings of the International Conference on AI in Design*. 2000. Boston.
63. P.Oreizy and R.Taylor, "On the role of software architectures in runtime system reconfiguration". *IEE Proceedings-Software*, 1998. 145.
64. M.Dastani, J.Hulstijn and L.van der Torre. "BDI and QDT: a comparison based on classical decision theory". *in Proceeding of Proceedings of AAAI Spring Symposium GTDT'01*. 2000.
65. A.Rao and M.Georgeff. "Modeling Rational Agents Within a BDI-Architecture." *in Proceeding of Knowledge Representation and Reasoning(KR&-R91)*. 1991. San Matteo: Morgan Kaufman publishers.
66. M.Bratman, "Intentions, Plans, and Practical Reason". *Harvard University Press*, 1987.
67. M.E.Bratman, D.J.Irsrael and M.E.Pollack, "Plans and resource-Bounded Practical Reasoning". *Computational Intelligence*, 1988. 4(4): p. 349-355.
68. J.Filipe. "A Normative and Intentional Agent Model for Organisation Modelling". *in Proceeding of Third International Workshop in Engineering Societies in the Agents World*. 2002. Madrid Spain.
69. C.Castelfranchi, F.Dignum, C.Jonker, and J.Treur. "Deliberate Normative Agents: Principles and Architectures". *in Proceeding of ATAL-99*. 1999. Orlando USA.
70. R.Espejo and R.Harnden, "The Viable Systems Model. Interpretations and Applications of Stafford Beer's VSM", ed. John Wiley & Sons. 1989, Chicester.
71. J.Moffett and M.Sloman, "Policy Hierarchies for Distributed Systems Management". *IEEE Journal on Selected Areas in Communications*, 1993. 11:

p. 1404-1414 http://www-dse.doc.ic.ac.uk/dse-papers/management/policyhierarchy.ps.Z.

72. M.Sloman, "Policy Driven Management for Distributed Systems". *Journal of Network and Systems Management*, 1994. 2 ftp://dse.doc.ic.ac.uk/dse-papers/management/pdman.ps.Z.
73. M.Sloman and E.Lupu. "Policy Specification for Programmable Networks". *in Proceeding of First International Working Conference on Active Networks (IWAN'99)*. 1999. Berlin.
74. E.Lupu and M.Sloman, "Conflicts in Policy-based Distributed Systems Management". *IEEE Transactions on Software Engineering*, 1999. 25(Special Issue on Inconsistency Management): p. 852-869.
75. E.Lupu and M.Sloman. "Conflict Analysis for Management Policies". *in Proceeding of Fifth IFIP/IEEE International Symposium on Integrated Network Management IM'97*. 1997. San-Diego: Chapman & Hall http://www-dse.doc.ic.ac.uk/dse-papers/management/IM97.ps.Z..
76. E.Lupu, D.Marriott, M.Sloman, and N.Yialelis. "A Policy Based Role Framework for Access Control". *in Proceeding of First ACM/NIST Workshop on Role-Based Access Control*. 1995. Maryland USA: ACM ftp://dse.doc.ic.ac.uk/dse-papers/management/rbac95.ps.Z..
77. N.Damianou, N.Dulay, E.Lupu, and M.Sloman, "Ponder: A Language for Specifying Security and Management Policies for Distributed Systems", Imperial College Research Report DoC 2001.
78. L.Lymberopoulos, E.Lupu and M.Sloman. "An Adaptive Policy Based Management Framework for Differentiated Services Networks". *in Proceeding of Proc. 3rd IEEE Workshop on Policies for Distributed Systems and Networks (Policy 2002)*. June 2002. Monterey, California.
79. K.Barber, T.Liu, H.Goel, and C.Martin. "Conflict Representation and Classification in a DomainIndependent Conflict Management Framework". *in Proceeding of the Third International Conference on Autonomous Agents*. 1999. Seattle WA.
80. D.Verma and R.Jennings. "Policy Based SLA,Management in Enterprise Networks". *in Proceeding of Proc. Policy 2001,International Workshop on Policies for Distributed Systems and Networks*. 2001. Bristol UK.
81. K.Yoshihara, M.Isomura and H.Horiuchi. "Distributed Policy-based Management Enabling Policy Adaptation on Monitoring using Active Network Technology". *in Proceeding of 12th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*. 2001. Nancy France.
82. M.Brunner and J.Quittek. "MPLS Management using Policies". *in Proceeding of Proc. IM 2001: 2001 IEEE/IFIP International Symposium on Intergrated Network Management*. 2001. Seattle USA.
83. M.Bearden, S.Garg and W.Lee. "Integrating Goal Specification in Policy-Based Management". *in Proceeding of Proc. Policy 2001: International Workshop on Policies for Distributed Systems and Networks*. 2001. Bristol, UK.
84. G.Cugola, E.Nitto and A.Fuggetta, "The JEDI Event-Based Infrastructure and its Applications to the Development of the OPSS WFMS". *IEEE Trans. on Software Engineering*, 1998. 27(9): p. 827-850.
85. G.Banavar, T.Chandra, B.Mukherjee, J.Nagarajarao, R.Strom, and D.Sturman. "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems". *in Proceeding of ICDCS*. 1999.

86. B.Segall and D.Arnold. "Elvin has left the Building:Publish/Subscribe Notification Service with Quenching". *in Proceeding of AUUG Technical Conference '97*. 1997. Brisbane Australia.
87. A.Carzaniga, D.Rosenblum and A.Wolf, "Design and Evaluation of a Wide-Area Event Noti cation Service". *ACM. Trans. on Computer Systems*, 2001. 19(3): p. 332-383.
88. J.Bacon, K.Moody, J.Bates, R.Hayton, C.Ma, A.McNeil, O.Seidel, and M.Spiteri, "Generic Support for Distributed Applications". *IEEE Computer Society Press*, 2000: p. 68-77.
89. C.Ma and J.Bacon. "CORBA: A CORBA-Based Event Architecure". *in Proceeding of the 4th USENIX Conf. On O-O tech. And Systems*. 1998. santa Fe USA.
90. P.Eugster, R.Guerraoui and J.Sventek. "Type-Based Publish/Subscribe". *in Proceeding of Technical report EPFL*. 2000. Lausanne Switzerland.
91. P.Eugster and R.Guerraoui. "Content-Based Publish/Subscribe with Strucutural Reflection". *in Proceeding of Proceedings of the 6th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS01)*. 2001. Texas USA.
92. A.Carzaniga, J.Deng and A.Wolf. "Fast Forwarding for Content-Based Networking". *in Proceeding of Technical report, Dept. of Computer Science*. 2001. University of Colorado.
93. A.Carzaniga, D.Rosenblum and A.Wolf, "Design and Evaluation of a Wide-Area Event Noti cation Service." *ACM. Trans. on Computer Systems*, 2001. 19(3): p. 332-383.
94. G.Cugola and E.Nitto. "Using a Publish/Subscribe Middleware to Support Mobile Computing". *in Proceeding of Middleware for Mobile Computing Workshop*. 2001. Heidelberg Germany.
95. L.Cabrera, M.Jones and M.Theimer. "Herald: Achieving a Global Event Notification Service." *in Proceeding of In Proc. Of the 8th Workshop on Hot Topics in OS (HotOS-VIII)*. 2001.
96. S.Zhuang, B.Zhao and A.Joseph. "Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination". *in Proceeding of the 11th Int. Workshop on Network and OS Support for Digital Audio and Video (NOSS-DAV01)*. 2001.
97. A.Rowstron, A.Kermarrec, M.Castro, and P.Drusche. "Scribe: The Design of a Large-Scale Event Notification Infrastructure". *in Proceeding of the 3rd Int. Workshop on Networked Group Communication (NGC2001)*. 2001.
98. P.Pietzuch and J.Bacon. "Hermes: A distributed event-Based Middleware Architecture". *in Proceeding of Proceedings of the 1st International workshops on Distributed Event-Based Systems (DEBS'02)*. 2002.
99. P.Pietzuch, B.Shand and J.Bacon. "A Framework for Event Composition in Distributed Systems". *in Proceeding of ACM/IFIP/USENIX Int. Middleware Conference*. 2003.
100. M.Shanahan, "The event calculus explained". *Artificial Intelligence Today*, 1999. 1600: p. 409-430.
101. C.Efstratiou, A.Friday, N.Davies, and K.Cheverst. "A Platform Supporting Coordinated Adaptation in Mobile Systems". *in Proceeding of Fourth IEEE Workshop on Mobile Computing Systems and Applications*. 2002. Callicoon New Yorl.
102. D.Garlan, S.Khersonsky and j.Soo Kim, "Model Checking Publish-Subscribe Systems". *SPIN*, 2003: p. 166-180.

103. K.Jones. "The Maturing of software architecture". *in Proceeding of software engineering symposium software Engineering Institute*. 1993. Pittsburgh.
104. D.LeMétayer, "Describing Software Architecture Styles Using Graph Grammars". *IEEE Transactions on Software Engineering*. 24: p. 521-533.
105. M.Wermelinger, A.Lopes and J.Fiadeiro. "A Graph Based Architectural (Re) configuration Language". *in Proceeding of the Joint 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 2001. Vienna Austria.
106. D.Garlan and R.Stratton."Architecture-based Adaptation of Complex Systems".accessed March.2003, http://schafercorp-ballston.com/dasada/projectlist.html.
107. L.Andrade and J.Fiadeiro. "An architectural approach to auto-adaptive systems". *in Proceeding of In Proc. ICDCS 2002 Workshops*. 2002: IEEE Computer Society Press.
108. R.Buyya, S.Chapin and D.DiNucci. "Architectural models for resource management in the Grid." *in Proceeding of 1st IEEE/ACM International Workshop on Grid Computing*. 2000. Bangalore India, .
109. I.Foster, C.Kesselman and S.Tuecke, "The anatomy of the Grid: Enabling scalable virtual organisations". *International Journal of Supercomputer Applications*, 2001. 15.
110. P.Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology". *IBM Corporation*, 2001 http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.
111. D.Chess, C.Palmer and S.White, "Security an autonomic computing environment". *IBM SYSTEMS JOURNAL*, 2003. 42.
112. R.Want, T.Pering and D.Tennenhouse, "Comparing autonomic and proactive computing". *IBM SYSTEMS JOURNAL*, 2003. 42.
113. J.Appavoo, K.Hui, C.Soules, R.Wisniewski, D.Da Silva, O.Krieger, M.Auslander, D.Edelsohn, B.Gamsa, G.Ganger, P.McKenney, M.Ostrowski, B.Rosenburg, M.Stumm, and J.Xenidis, "Enabling autonomic behavior in systems software with hot swapping,". *IBM SYSTEMS JOURNAL*, 2003. 42.
114. N.K.Diakov, H.J.Batteram, H.Zandbelt, and M.J.Sinderen. "Monitoring of Distributed Component Interactions". *in Proceeding of RM'2000 Workshop on Reflective Middleware*. 2000. New York.
115. T.Chandra and S.Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems". *Journal of the ACM*, 1996. 43: p. 225-267.
116. K.S.Barber, T.H.liu and D.C.Han. "Strategic Decision-Making for Conflict resolution in Dynamic Organized Multi-Agent Systems". *in Proceeding of GDN 2000 PROGRAM*. 2000.
117. Jini Community."Jini Specification".accessed August.2003, http://wwws.sun.com/software/jini/specs/jini1.1html/js-spec.html.
118. D.Reilly, A.Taleb-Bendiab, A.Laws, and N.Badr. "An Instrumentation and Control-Based Approach for Distributed Application Management and Adaptation". *in Proceeding of Workshop on Self-Healing Systems (WOSS'02)*. 2002. Charleston SC USA.
119. S.Krishnana, P.Wagstrom and G.V.Laszewski."GSFL: A Workflow Framework for Grid Services".accessed July.2003, www.cs.indiana.edu/~srikrish/talks/gsfl.ppt.

120. D.pautler, S.Woods and A.Quilici. "exploiting domain-specific Knowledge to refine Simulation specification". *in Proceeding of Proc. 12th conf. Auto-mated Software Eng. IEEE CS Press*. 1997.
121. Java Environment, http:// wwww.sun.com.
122. Middleware."http://web.syr.edu/~jshwang/iMint/imint.html.
123. Middleware".accessed http://web.syr.edu/~jshwang/iMint/imint.html.
124. JavaOne, "Designing and Building Distributed Service-Based Architectures with JiniTM Network Technology: a Panel Discussion". *Technical Session*, 2001 http://servlet.java.sun.com/javaone/conf/sessions/2506/google-sf2001.jsp.
125. Jini Community, "Jini Surrogate Architecture Overview". http://surrogate.jini.org/overview.pdf.
126. Jini URL, http://jini.org/.
127. M.Yu, A.Taleb-Bendiab, D.Reilly, and W.Omar. "Ubiquitous Service Interoperation through Polyarchical Middleware". *in Proceeding of Proceedings Web Intelligence (WI 2003)*. 2003. Halifax Canada.
128. D.Garlan and R.Stratton. "Architecture-based Adaptation of Complex Systems". *in Proceeding of.* accessed January 2002 http://schafercorp-ballston.com/dasada/projectlist.html.
129. D.Reilly and A.Taleb-Bendiab. "Dynamic Software Instrumentation for Jini Applications". *in Proceeding of Proceedings of the 3rd International Workshop on Software Engineering Middleware SEM 2002 (ICSE 2002)*. 2002. Orlando Florida USA.
130. Tuple Space, http://www.infolets.com/1006361585/.
131. Jini Community."Augest.2003, http://wwws.sun.com/software/jini/specs/jini1.1html/js-spec.html.
132. S.Sen and H.Durfee. "Unsupervised surrogate agents and search bias change in flexible distributed scheduling". *in Proceeding of the First International Conference on Multi-Agent System (ICMAS-95)*. 1995.
133. Alphaworks."Thin-Client Framework".accessed September.2003, http://www.alphaworks.ibm.com/tech/tcf.
134. D.Reilly and A.Taleb-Bendiab. "A Service-Based Architecture for In-Vehicle Telematics Systems". *in Proceeding of IEEE Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*. 2002. Vienna Austria.
135. IBM Research."Applications of Multi-Agents Learning in E-Commerce and Autonomic Computing".accessed October.2003, http://www.cs.rutgers.edu/~mlittman/topics/nips02/nips02/kephart.ppt.
136. Sun-Microsystems."Trail: The Reflection API".accessed October.2003, Http://java.sun.com/docs/books/tutorial/reflect/.
137. D.Helman, D.Bader and J.JáJá. "Parallel Algorithms for Personalized Communication and Sorting With an Experimental Study." *in Proceeding of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*. 1996. Padua, Italy.
138. Jose Renato Santos G. (John) Janakiraman, and Yoshio Turner,. "Automated Multi-Tier System Design for Service Availability,". *in Proceeding of HPL-2003-109 Technical report*. 2003.
139. Distributed Application Development Process".accessed October.2003, http://www.gantthead.com.
140. B.Meyer, "Object-Oriented Software Construction". 1997. 2nd ed Prentice Hall New Jersey.

141. D.Bakken, "Middleware". *Encyclopedia of Distributed Computing Kluwer Academic Press*, 2001.
142. S.Cheng, D.Garlan and B.Schmerl. "Software Architecture-based Adaptive for Grid Computing". *in Proceeding of Proceedings The11th IEEE Conference on High performance Distributed Computing (HPDC'02)*. 2002. Edinburgh Scotland.